History-Aware, Real-Time Risk Detection in Business Processes

R. Conforti[1], G. Fortino[2], M. La Rosa[1], and A.H.M. ter Hofstede[1,3]

[1] Queensland University of Technology, Australia
{r.conforti@students.,m.larosa@,a.terhofstede@}qut.edu.au
[2] Università della Calabria, Italy
fortino@unical.it
[3] Eindhoven University of Technology, The Netherlands

Abstract. This paper proposes a novel approach for identifying risks in executable business processes and detecting them at run-time. The approach considers risks in all phases of the business process management lifecycle, and is realized via a distributed, sensor-based architecture. At design-time, sensors are defined to specify risk conditions which when fulfilled, are a likely indicator of faults to occur. Both historical and current process execution data can be used to compose such conditions. At run-time, each sensor independently notifies a sensor manager when a risk is detected. In turn, the sensor manager interacts with the monitoring component of a process automation suite to prompt the results to the user who may take remedial actions. The proposed architecture has been implemented in the YAWL system and its performance has been evaluated in practice.

1 Introduction

According to the AS/NZS ISO 31000 standard, a business process risk is the chance of something happening that will have an impact on the process objectives, and is measured in terms of likelihood and consequence [26]. Incidents such as scandals in the finance sector (the 4.9B Euros fraud at Société Générale), in the health sector (Patel Inquiry) and in the aviation industry (failed terrorist attacks) have shown that business processes are constantly exposed to a wide range of risks.

Failures of process-driven risk management can result in substantial financial and reputational consequences, potentially threatening an organization's existence. Legislative initiatives such as the Sarbanes-Oxley Act[4] and Basel II [2] in the finance sector have highlighted the pressing need to better manage business process risks. As a consequence of these mandates, organizations are now seeking new ways to *control* process-related risk and are attempting to incorporate it as a distinct view in their operational management. However, whilst conceptually appealing, to date there is little guidance as to how this can best be done. Currently the disciplines of process management and risk management are largely disjoint and operate independently of one another. In industry they are usually handled by different organizational units. Within academia, recent research has centered on the identification of process-related risks. However the incidents described above demonstrate that a focus on risk analysis alone is no longer adequate, and an active, real-time risk detection and controlling is required.

[4] www.gpo.gov/fdsys/pkg/PLAW-107publ204

We propose a novel approach for operationalizing risk management in Business Process Management automation Suites (BPMSs). The aim of this approach is to provide a concrete mechanism for identifying risks in executable business process models and detecting them during process execution. This is achieved by considering risks throughout the BPM lifecycle, from process model design where risk conditions are defined, through to process diagnosis, where risks are monitored. By automating risk detection, the interested users (e.g. a process administrator) can be notified as early as a risk is detected, such that remedial actions can be taken to rectify the current process instance, and prevent an undesired state of the process (*fault* for short), from occurring. Based on historical data, we can also compute the probability of a risk at run-time, and compare it to a threshold, so as to notify the user only when the risk's criticality is no longer tolerable. To the best of our knowledge, this is the first attempt to incorporate risks into executable business processes and enable their automatic detection at run-time.

The proposed approach is realized via a distributed, sensor-based architecture. Each sensor is coupled with a risk condition capturing the situation upon which the risk of a given fault may occur. Sensors are defined at design-time on the process model. Conditions can be determined via a query language that can fetch both historical and current execution data from the logs of the BPMS. At run-time sensors are registered with a central sensor manager. At a given sampling rate, or based on the occurrence of a specific event, the sensor manager retrieves and filters all data relevant for the various sensors (as it is logged by the BPMS engine), and distributes it to the relevant sensors. If a sensor condition holds, i.e. if the probability of the associated risk is above a given threshold, the sensor alerts the sensor manager which in turn notifies the monitoring component of the BPMS. The distributed nature of the architecture guarantees that there is no performance overhead on the BPMS engine, and thus on the execution of the various process instances. We implemented this architecture on top of the YAWL system. We extended the YAWL Editor to cater for the design of risk sensors, and equipped the run-time environment with a sensor manager service that interacts with YAWL's monitoring service and execution engine.

To prove the feasibility of the proposed approach, we used fault tree analysis [4] (a well-established risk analysis method) to identify risk conditions in a reference process model for logistics, in collaboration with an Australian risk consultant. These risks embrace different process aspects such as tasks' order dependencies, involved resources and business data, and relate to historical data where needed, to compute risk probabilities. We expressed these conditions via sensors in the YAWL environment, and measured the time needed to compute these conditions at run-time. The tests showed that the sensor conditions can be computed in a matter of milliseconds without impacting on the performance of the running process instances.

This paper is organized as follows. Section 2 illustrates the running example in the logistics domain. Section 3 describes our risk-aware BPM approach while Sect. 4 presents the sensor-based architecture to implement this approach. The architecture is evaluated in Sect. 5. Section 6 covers related work while Sect. 7 concludes the paper.

2 Running Example

In this section we use an example to illustrate how the risk of possible faults to occur during a business process execution can be identified as early as possible. In particular, we show how risks can be expressed in terms of process-specific aspects such as tasks occurrence, data or available resources. Figure 1 describes the payment subprocess of an order fulfillment business process which is inspired by the VICS industry standard for logistics [30]. The notation used to represent this example is that of YAWL [13], although a deep knowledge of this language is not required.

This process starts after the freight has been picked up by a carrier and deals with the shipment payment. The first task is the production of a Shipment Invoice containing the shipment costs related to a specific order for a specific customer. If shipments have been paid in advance, all that is required is for a Finance Officer to issue a Shipment Remittance Advice specifying the amount being debited to the customer. Otherwise, the Finance Officer issues a Shipment Payment Order that needs to be approved by a Senior Finance Officer (who is the superior of this Finance Officer). At this point, a number of updates may be made to the Shipment Payment Order by the Finance Officer that issued it, but each of these needs to be approved by the Senior Finance Officer. After the document is finalized and the customer has paid, an Account Manager can process the shipment payment by specifying the balance. If the customer underpaid, the Account Manager needs to issue a Debit Adjustment, the customer needs to pay the balance and the payment needs to be reprocessed. A customer may also overpay. In this case the Account Manager needs to issue a Credit Adjustment. In the latter case and in case of a correct payment, the shipment payment process is completed.

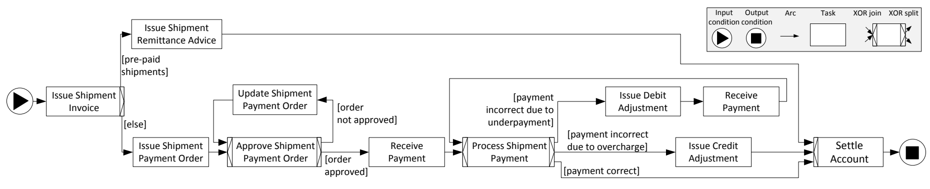


Fig. 1. Order-Fulfillment: Payment subprocess.

In collaboration with a risk analyst of an Australian consulting company, we identified four faults that can occur during the execution of this payment subprocess. In order to prevent the occurrence of these faults, for each of them we also defined an associated risk condition by using fault tree analysis [4]. Accordingly, each risk condition is expressed as a set of lower-level boolean events which are organized in a tree via logical connectives such as ORs, ANDs and XORs.

The first fault is an *overtime process* fault. A Service Level Agreement (SLA) for a process or for a given task within a process, may establish that the process (or task) may not last longer than a Maximum Cycle Time MCT, otherwise the organization running the process may incur a pecuniary penalty. In our case, an overtime fault occurs if an instance of the payment subprocess is not completed within an MCT of five days.

To detect the risk of overtime fault at run-time, we should check the likelihood that the running instance does not exceed the MCT based on the amount of time T_c expired at the current stage of the execution. Let us consider T_e as the remaining cycle time, i.e. the amount of time estimated to complete the current instance given T_c. Then the probability of exceeding MCT can be computed as $1 - MCT/(T_e + T_c)$ if $T_e + T_c > MCT$ and is equal to 0 if $T_e + T_c \leq MCT$. If this probability is greater than a tolerance value (e.g. 60%), we notify the risk to the user. The estimation of the remaining cycle time is based on past executions of the same process and can be computed using the approach in [29] (see Section 5 for more details).

The second fault is related to the resources participating in the process. The Senior Finance Officer who has approved a Shipment Payment Order for a given customer, must have not approved another order by the same customer in the last d days, otherwise there is an *approval fraud*. This fault is thus generated by the violation of a four-eye principle across different instances of the Payment subprocess.

To detect the risk of this fault we first have to check that there is an order, say order o of customer c, to be approved. This means checking that an instance of task Approve Shipment Payment Order is being executed. Moreover, we need to check that either of the following conditions holds: i) o has been allocated to a Senior Finance Officer who has already approved another order for the same customer in the last d days; or ii) at least one Senior Finance Officer is available who approved an order for customer c in the last d days and all other Senior Finance Officers who never approved an order for c during the last d days are available. The corresponding fault tree is shown in Fig. 2.

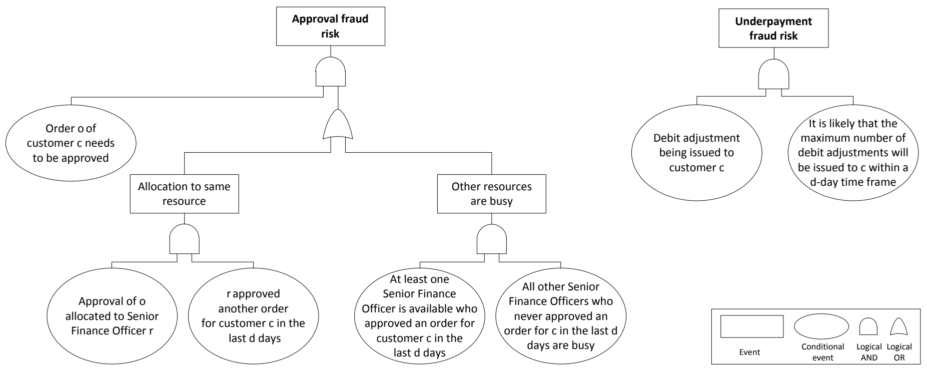

Approval fraud risk
Order o of customer c needs to be approved
Allocation to same resource
Other resources are busy
Approval of o allocated to Senior Finance Officer r
r approved another order for customer c in the last d days
At least one Senior Finance Officer is available who approved an order for customer c in the last d days
All other Senior Finance Officers who never approved an order for c in the last d days are busy
Underpayment fraud risk
Debit adjustment being issued to customer c
It is likely that the maximum number of debit adjustments will be issued to c within a d-day time frame
Event
Conditional event
Logical AND
Logical OR

Fig. 2. The fault trees for Approval Fraud and Underpayment Fraud.

The third fault relates to a situation where a process instance executes a given task too many times. This situation typically occurs in the context of loops. Not only could this lead to a process slowdown but also to a "livelock" if the task is in a loop whose exit condition is purposefully never met. In general, given a task t a maximum number of allowable executions of t per process instance $MAE^i(t)$ can be fixed as part of the SLA for t. With reference to the Payment subprocess, this can occur for example if task Update Shipment Payment Order is re-executed five times within the same process instance. We call this an *order unfulfillment* fault.

To detect the risk of this fault at run-time, we need to check if: i) an order o is been updated (i.e. task Update Shipment Payment Order is currently being performed for order o); and ii) it is likely that this order will be updated again (i.e. task Update Shipment Payment Order will be repeated within the same process instance). The probability that the number of times a task will be repeated within the same instance of the Payment subprocess is computed by dividing the number of instances where the MAE^i for task Update Shipment Payment Order has been reached, over the number of instances that have executed this task at least as many times as it has been executed by the current instance, and have completed. The tolerance value indicates a threshold above which the risk should be notified to the user. For example, if this threshold is 60% for task t, a risk should be raised if the probability of $MAE^i(t)$ is greater than 0.6.

The fourth fault is an *underpayment fraud*. It relates to a situation in which a given task is executed too many times across multiple process instances. Similar to the previous fault, given a task t we can define a maximum number of allowable executions of t per process $MAE^p(t)$ as part of the SLA for p. In our example, this type of fault occurs when a customer underpays more than three times within the last five days.

To detect the risk of underpayment fraud, we need to check if: i) a debit adjustment is currently being issued to a customer c (i.e. task Issue Debit Adjustment is currently being performed for customer c); and ii) it is likely that the maximum number of debit adjustments will be issued to the same customer in a d-day time frame. The probability that MAE^p is reached for task Issue Debit Adjustment of customer c in d days is computed by dividing the number of customers for which the MAE^p for task Issue Debit Adjustment has been reached within d days, over the number of customers for which this task has been executed at least as many times as it has been executed for c within d days. If this probability is above a tolerance value, the risk should be raised and the user notified. Similar to the previous risk, the tolerance value indicates a threshold above which this risk should be notified to the user. The corresponding fault-tree is shown in Fig. 2.

3 Risk-aware Business Process Management

As we have seen in the context of the payment example, a fault in a business process is an undesired state of a process instance which may lead to a process failure (e.g. the violation of a policy may lead to a process instance being interrupted). Identifying a fault in a process requires determining the condition upon which the fault occurs. For example, in the payment subprocess, we have an underpayment fraud if a customer underpays more than three times within a five-day time frame.

However, a *fault condition* holds only when the associated fault has occurred, which is typically too late to avoid a process failure. Indeed, we need to be able to estimate the risk of a process fault, i.e. if, and possibly with what likelihood, the fault will occur in the future. Early risk detection allows process users to promptly react with countermeasures, if any, to prevent the related fault from occurring at all.

We use the notion of *risk condition*, as opposed to fault condition, to describe the set of events that lead to the possibility of a fault to occur in the future. In order to evaluate risk conditions "on-line", i.e. while a process instance is being executed, we need to consider the current state of the BPMS. This means knowing the state of all running instances of any process (and not only the state of the instance for which we are computing the risk condition), the resources that are busy and those that are available, and the values of the data variables being created and consumed. Moreover, we need to know the historical data, i.e. the execution data of all instances that have been completed. In particular, we can use historical data to estimate the probability of a given fault to occur, i.e. the *risk probability*. For example, for the underpayment fraud, we can estimate the likelihood that another debit adjustment is being issued for a given combination of customer/order (historical data), given that one such debit adjustment has just been issued (current data). To obtain a boolean risk condition, we compare the risk probability that we obtain with a tolerance value, such that the condition holds if the risk probability exceeds the given threshold. For example, we raise the risk of underpayment fraud if the risk probability is greater then 60%.

In other cases, we may avoid to embed a risk probability in the risk condition, if we are able to determine the occurrence of a set of events which directly leads to a high risk. This is the case of the approval fraud, where both the events "Allocation to same resource" and "Other resources are busy" already signal a high risk of approval fraud.

Based on these considerations, we present a novel approach for on-line risk detection in business processes. The focal idea of this approach, shown in Fig. 3, is to embed elements of risk into all four phases of the traditional BPM lifecycle [7].

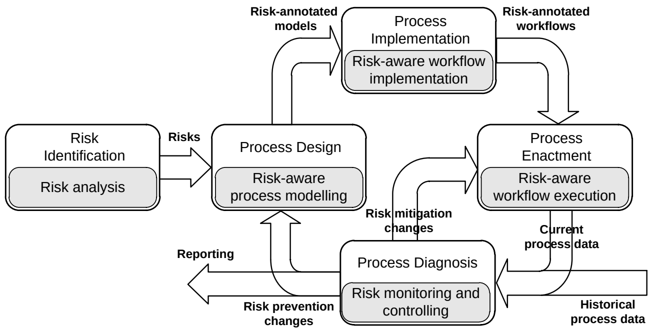


Fig. 3. Risk-aware Business Process Management lifecycle.

Input to this "risk-aware" BPM lifecycle is a *Risk Identification* phase, where risk analysis is carried out to identify risks in the process model to be designed. Traditional risk analysis methods such as FTA (as seen in the previous section), Root Cause Analysis [17] or CORAS [25], can be employed in this phase. The output of this phase is a set of risks, each expressed as a risk condition.

Next, in the *Process Design* phase, these high-level risk conditions are mapped down to process model-specific aspects. For example, the condition "debit adjustment being issued to customer c for order o" is mapped to the occurrence of a specific task, namely "Issue Debit Adjustment" in the Payment process model. The result of this second phase is a risk-annotated process model. In the next phase, *Process Implementation*, these conditions are linked to workflow-specific aspects, such as content of variables, and resource allocation states. For example, "customer c" is linked to the `Customer` element of the XML representation of the `Debit Adjustment` document. Process Implementation may be integrated with Process Design if the language used at design-time is executable (e.g. BPMN 2.0 or YAWL).

The risk-annotated workflow model resulting from Process Implementation is then executed by a risk-aware process engine during *Process Enactment*. Historical data stored in process logs, and current execution data coming from process enactment, are filtered, aggregated and analyzed in the *Process diagnosis* phase, in order to evaluate the various risk conditions. When a risk condition evaluates to true, the interested users (e.g. a process administrator) are notified and reports can also be produced during this phase for auditing purposes. Finally, this phase can trigger changes in the current process instance, to mitigate the likelihood of a fault to occur, or in the underlying process model, to prevent a given risk from occurring ever again.

In the next section we describe a sensor-based architecture to operationalize this enhanced BPM lifecycle.

4 Sensor-based realization

In order to realize our risk-aware BPM lifecycle, we devised an approach based on sensors. In a nutshell, the idea is to capture risk and fault conditions via sensors, and then monitor these sensors during process execution. An overview of this approach is shown in Fig. 4 using the BPMN 2.0 notation [21].

Sensors are defined during the *Process Design* and *Process Implementation* phases of our risk-aware BPM lifecycle (see Fig. 3), for each process model for which the presence of risks and/or faults need to be monitored. If the process model is specified via an executable language, then these two phases coincide.

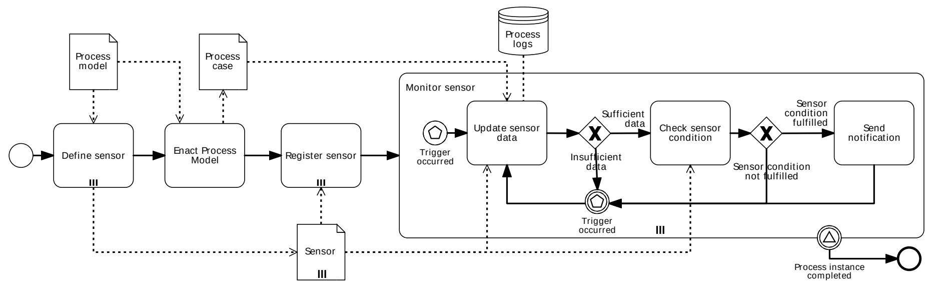


Fig. 4. Realization of risk-aware BPM lifecycle via sensors.

A sensor is defined through a boolean *sensor condition*, constructed on a set of process variables, and a *sensor activation trigger*. Process variables are used to retrieve information from the specific instance in which the sensor condition will be evaluated as well as from other instances, either completed or still running. For example, we can use variables to retrieve the resource allocated to a given task, the value of a task variable, or the status of a task. Process instances can either be identified based on the current instance (e.g. the last five instances that have been completed before the current one), or based on the fulfillment of a *case condition* (e.g. "all instances where a given resource has executed a given task"). The sensor condition can represent either a *risk condition* associated with a fault, or a *fault condition*, or both. If both conditions are specified, the fault condition is evaluated only if the risk condition evaluates to true. For example, the sensor will check if an overtime process fault has occurred in a process instance only if first the risk of such fault has first been detected, based on the estimation of the remaining cycle time for this instance. Finally, the sensor activation trigger can be either a timer periodically fired according to a sampling rate (e.g. every 5 minutes), or an event emitted by the process engine (e.g. the completion of a task). Figure 5 shows a simplified version of the sensor definition language by using an abstract syntax [19]; the complete definition of this language is provided in the technical report [5].

$$
\begin{aligned}
\mathit{Sensor} &\triangleq \mathit{v} : \mathit{Variables}; \mathit{c} : \mathit{Condition};\\
&\quad \mathit{t} : \mathit{Trigger}\\
\mathit{Variables} &\triangleq \mathit{Assignment}^{+}\\
\mathit{Condition} &\triangleq \mathit{riskCond}, \mathit{faultCond} : \mathit{boolExpr}\\
\mathit{Trigger} &\triangleq \mathit{timer} \mid \mathit{event}\\
\mathit{Assignment} &\triangleq \mathit{CaseExpr} \mid \mathit{CaseElemExpr} \mid\\
&\quad \mathit{VarFunc} \mid \mathit{Definition}\\
\mathit{CaseExpr} &\triangleq \mathit{result} : \mathit{varName};\\
&\quad \mathit{e} : \mathit{CaseIDStat}; \mathit{a} : \mathit{Action}\\
\mathit{CaseElemExpr} &\triangleq \mathit{ce} : \mathit{CaseExpr}; \mathit{x} : \mathit{TaskOrNet}\\
\mathit{VarFunc} &\triangleq \mathit{result}, \mathit{input} : \mathit{varName};\\
&\quad \mathit{va} : \mathit{varAction}\\
\mathit{Definition} &\triangleq \mathit{result} : \mathit{varName}; \mathit{c} : \mathit{constant}\\
\mathit{CaseIDStat} &\triangleq \mathit{absoluteExpr} \mid \mathit{relExpr} \mid\\
&\quad \mathit{CaseCondSet}\\
\mathit{CaseCondSet} &\triangleq \mathit{CaseCondExpr} \mid \mathit{CaseCond} \mid\\
&\quad \mathit{CaseParam}\\
\mathit{CaseCondExpr} &\triangleq \mathit{pes}_{1}, \mathit{pes}_{2} : \mathit{CaseCondSet};\\
&\quad \mathit{bo} : \mathit{booleanOp}\\
\mathit{CaseCond} &\triangleq \mathit{x} : \mathit{TaskOrNet}; \mathit{a} : \mathit{Action};\\
&\quad \mathit{c} : \mathit{compOp}; \mathit{r} : \mathit{rightHandExpr}\\
\mathit{CaseParam} &\triangleq \mathit{i} : \mathit{idFunc}; \mathit{c} : \mathit{compOp};\\
&\quad \mathit{r} : \mathit{rightHandExpr}\\
\mathit{TaskOrNet} &\triangleq \mathit{taskLabel} \mid \mathit{netName}\\
\mathit{Action} &\triangleq \mathit{predFunc} \mid \mathit{taskOrNetVar} \mid\\
&\quad \mathit{SubVarExpr} \mid \mathit{inputPredFunc}
\end{aligned}
$$

(a)

Abstract element	**Description**
Sensor	is composed by $\mathit{Variables}$, $\mathit{Condition}$, $\mathit{Trigger}$
$\mathit{Variables}$	identifies a set of $\mathit{Assignment}$
$\mathit{Condition}$	identifies the sensor condition composed by a risk condition and by a fault condition
$\mathit{Trigger}$	specifies the type of trigger desired
$\mathit{Assignment}$	defines a mapping between a variable and a piece of information
$\mathit{CaseExpr}$	identifies information belonging to the process instance
$\mathit{CaseElemExpr}$	identifies information belonging to an element of the process instance
$\mathit{VarFunc}$	returns the result of a function executed on a variable
$\mathit{Definition}$	sets the value of a variable to a predefined value
$\mathit{CaseIDStat}$	identifies a process instance or a set of process instances
$\mathit{CaseCondSet}$	describes how a process instance can be identified
$\mathit{CaseCondExpr}$	is a boolean conjunction of $\mathit{CaseCondSet}$
$\mathit{CaseCond}$	specifies the condition that the process instance must satisfy
$\mathit{CaseParam}$	specifies the parameter related to the process instance id
$\mathit{TaskOrNet}$	identifies an element of the process model using taskLabel or netName
Action	identifies the type of information desired

(b)

Fig. 5. Abstract syntax of sensor definition language (a); Description of its elements (b).

During *Process Enactment*, the defined sensors are registered with a *sensor manager*, which activates them. In the *Process Diagnosis* phase, which starts as soon as the process is enacted, the activated sensors receive updates on the variables of their sensor conditions according to their trigger (timer or event). When a sensor receives an update, it checks its sensor condition. If the condition holds, a notification is sent from the sensor to the monitor service of the BPMS.

The sensor manager relies on three interfaces to interact with the BPMS (see Fig. 6(a)):

– *Engine interface*, used to register a sensor with a particular event raised by the BPMS engine. When the event occurs the sensor is notified by the sensor manager.
– *Database interface*, used to query the BPMS database in order to collect current and historical information.
– *Monitor interface*, used to notify the detection of risks and faults to the monitor service of the BPMS.

These interfaces can be implemented by the vendor or user of the BPMS where the sensor manager needs to be installed. In this way, our sensor manager can virtually be interfaced with any BPMS. As an example, the conceptual model of the database interface is showed in Fig. 6(b), where methods have been omitted for space reasons. This conceptual model is inspired by the reference process meta-model of the WfMC [14], in order to cover as many aspects as possible of a workflow model, and meantime, to

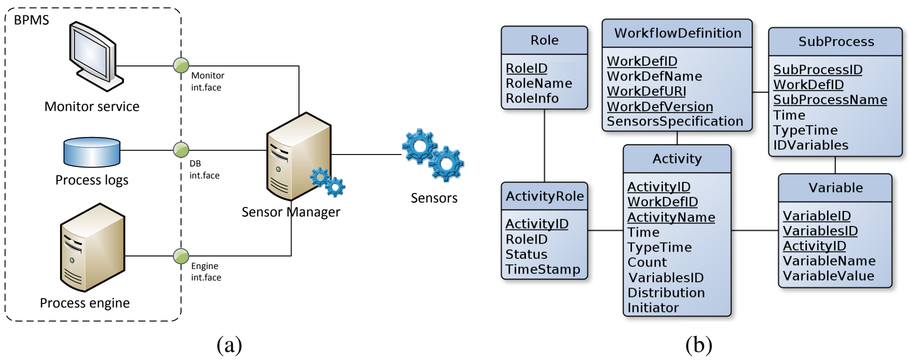


Fig. 6. Sensor-based architecture (a); Database Interface schema model (b).

remain as generic as possible. For example, class *WorkFlowDefinition* allows one to retrieve information about the process model where the sensor is defined, such as process identifier and name, while class *SubProcess* allows one to retrieve information about a specific subprocess, and so on. This interface should be implemented according to the characteristics of the specific database used in the BPMS at hand. For an efficient use of the interface, one should also define indexes on the attributes of the BPMS database that map the underlined attributes in Fig. 6(b). These indexes have been determined based on the types of queries that can be defined in our sensor condition language.

An alternative approach to achieve the portability of the sensor manager, would be to read the BPMS logs from a standard serialization format such as OpenXES. However, as we will show in Sect. 5, this solution is rather inefficient.

The advantages of using sensors are twofold. First, their conditions can be monitored while the process model is being executed, i.e. in real-time. Second, according to a distributed architecture, each sensor takes care of checking its own condition after being activated by the sensor manager. In this way, potential execution slowdowns are avoided (e.g., the process engine and the sensor manager could be deployed to two different machines).

We now have all ingredients to show how the risks that we identified for the Payment subprocess can be captured via sensor conditions, using the language defined in Fig. 5. For space reasons we only focus on the approval fraud and underpayment fraud risks. A description of the other sensor conditions is provided in the technical report [5].

We recall that there is an approval fraud whenever a Senior Finance Officer approves two orders for the same customer within five days. Accordingly, the corresponding risk can be detected if given an order o of customer c to be approved, either of the following conditions holds: i) o has been allocated to a Senior Finance Officer who has already approved another order for the same customer in the last five days; or ii) at least one Senior Finance Officer is available who approved an order for customer c in the last five days and all other Senior Finance Officers who never approved an order for c during the five days are available.

This risk condition is triggered by an event, i.e. the spawning of a new instance of task Approve Shipment Payment Order. This is checked by using a variable to retrieve the status of this task in the current instance. The risk condition itself is given by the disjunction of the two conditions described above. The first such condition is checked by using a variable to retrieve which resources were allocated to task Approve Shipment Payment Order, and another variable to retrieve the number of times this task was completed for customer c. This latter variable is defined via a case condition over customer c, the completion time of this task (that must be greater than the start time of the current task Approve Shipment Payment Order minus five days in milliseconds), and the identifier of the instance (that must be different from the identifier of the current instance).

The second condition is checked by using two variables and invoking two functions. A variable to retrieve which resources completed task Approve Shipment Payment Order, and another variable to retrieve all resources that can be offered this task (i.e. the current task). The first variable is defined via a case condition over customer c and the completion time of this task (that must be greater than the start time of the current task Approve Shipment Payment Order minus five days). The two invoked functions return the number of tasks started on the resources that completed task Approve Shipment Payment Order, and the number of tasks in the execution queue of the resources who have been offered this task, and did not complete it for customer c in the last five days.

The definition of the above variables in our sensor language is provided below, while the $Action$ elements used in these definitions are described in Table 1.

$$\begin{aligned}
\mathsf{sfo}_1 &= \mathsf{case(current).Approve\_Shipment\_Payment\_Order\_593(allocateResource)}\\
\mathsf{c} &= \mathsf{case(current).Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company}\\
\mathsf{d} &= \mathsf{5}\\
\mathsf{ASPO}_{\mathrm{AllocateTime}} &= \mathsf{case(current).Approve\_Shipment\_Payment\_Order\_593(OfferTimeInMillis)}\\
\mathsf{ASPO}_{\mathrm{\#App}} &= \mathsf{case(Approve\_Shipment\_Payment\_Order\_593(completeResource)=sfo_1} \wedge\\
&\ \mathsf{Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company=c} \wedge\\
&\ \mathsf{Approve\_Shipment\_Payment\_Order\_593(CompleteTimeInMillis)}>\\
&\ \mathsf{(ASPO_{\mathrm{AllocateTime}}\text{-}(d*24*60*60*1000))} \wedge\\
&\ \mathsf{(ID)!=[IDCurr]).Approve\_Shipment\_Payment\_Order\_593(CountElements)}\\
\mathsf{sfo}_2 &= \mathsf{case(Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company=c} \wedge\\
&\ \mathsf{Approve\_Shipment\_Payment\_Order\_593(isCompleted)=\text{“true”}} \wedge\\
&\ \mathsf{Approve\_Shipment\_Payment\_Order\_593(CompleteTimeInMillis)}>\\
&\ \mathsf{(ASPO_{\mathrm{StartTime}}\text{-}(d*24*60*60*1000))} \wedge\\
&\ \mathsf{(ID)!=[IDCurr]).Approve\_Shipment\_Payment\_Order\_593(completeResource)}\\
\mathsf{sfo} &= \mathsf{case(current).Approve\_Shipment\_Payment\_Order\_593(offerDistribution)}
\end{aligned}$$

After the definition of the variables, the risk condition is specified as follows: $\mathsf{(ASPO_{\mathrm{\#App}}{>}0)} \vee \mathsf{((sfo_2.startMinNumber{=}0)} \wedge \mathsf{(sfo.startMinNumberExcept.sfo_2{>}{=}1))}$.

We recall that an underpayment fraud occurs whenever a customer underpays more than three times in a five-day time frame. Accordingly, the respective risk can be detected if i) task Issue Debit Adjustment is being performed for a given customer and order (this is the trigger for this risk); and ii) the probability that the maximum number of allowable executions for this task will be reached in a five-day time frame, is above the fixed tolerance value for this risk, say 60% (this is the risk condition itself). This condition can be checked by using two variables: one to retrieve the number of times the task Issue Debit Adjustment has been completed for this customer, the other to retrieve the probability that an attempted fraud will take place. For this second variable, we use the $Action$ "`FraudProbabilityFunc`" to compute the specific probability (see Table 1).

The defined variables are implemented through the sensor language as follows:

$$\begin{aligned}
\text{IDA}_{\text{StartTime}} &= \text{case(current).Issue\_Debit\_Adjustment\_605(StartTimeInMillis)}\\
\text{c} &= \text{case(current).Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company}\\
\text{d} &= 5\\
\text{IDA}_{\text{\#Issue}} &= \text{case(Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company=c} \wedge\\
&\ \text{Issue\_Debit\_Adjustment\_605(Count)}>0 \wedge\\
&\ \text{Issue\_Debit\_Adjustment\_605(CompleteTimeInMillis)}>(\text{IDA}_{\text{StartTime}}\text{-d*24*60*60*1000))}\\
&\ \text{.Issue\_Debit\_Adjustment\_605(CountElements)}\\
\text{GroupingElement} &= \text{Issue\_Shipment\_Invoice\_594.ShipmentInvoice.Company}\\
\text{WindowElement} &= \text{Issue\_Debit\_Adjustment\_605(CompleteTimeInMillis)}\\
\text{Threshold} &= 0.6\\
\text{Probability} &= \text{case(Issue\_Debit\_Adjustment\_605(Count)}>0 \wedge \text{(ID)!=[IDcurr]).Issue\_Debit\_Adjustment\_605}\\
&\ \text{(FraudProbabilityFunc, IDA}_{\text{\#Issue}}\text{, 3, GroupingElement, WindowElement, (d*24*60*60*1000))}
\end{aligned}$$

These variables are used to compose the following risk condition: $\text{Probability}>0.6$.

Action	Description
(ID)	returns the ID of the generic instance that is being analyzed
[IDCurr]	returns the ID of the instance that the sensor is monitoring
Count	returns the number of times a task has been completed
allocateResource	returns the resources to which the task has been allocated
completeResource	returns the resource that completed the task
isStarted	returns "*true*" if the task has been started
isCompleted	returns "*true*" if the task has been completed
OfferTimeInMillis	returns the time (in millisecond) when the task has been offered
StartTimeInMillis	returns the time (in millisecond) when the task has been started
CompleteTimeInMillis	returns the time (in millisecond) when the task has been completed
ShipmentInvoice.Company	returns the value of the subvariable `Company` belonging to the variable `ShipmentInvoice`
offerDistribution	returns list of resources to which the task is offered by default
CountElements	returns the number of instances that satisfy the parameters required
FraudProbabilityFunc	returns the probability of a fraud using as parameters: the current number of executions, the maximum number of executions allowed, the parameter used to group the instances, the parameter used to identify a temporal window, the dimension of the temporal window

Table 1. Description of the *Action* elements used in the example sensor conditions.

5 Evaluation

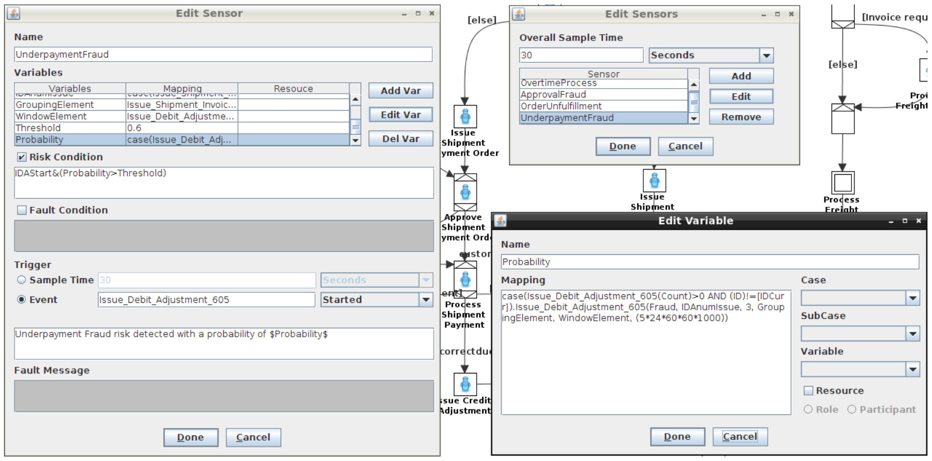
In this section we discuss the implementation of the sensor-based architecture in the YAWL system and then evaluate its performance.

5.1 Implementation

In order to prove the feasibility of our approach, we implemented the sensor-based architecture in the YAWL system.[5] We decided to extend the YAWL system for the following reasons. First, this system is based on a service-oriented architecture, which facilitates the seamless addition of new services. Second, the system is open-source, which facilitates its distribution among academics and practitioners, and widely used in practice (the system has been downloaded over 100,000 times since its first inception in the open-source community). Finally, the underlying YAWL language is very expressive as it provides wide support for the workflow patterns [13].

As part of this implementation, we extended the YAWL Editor version 2.2beta with a new component, namely the Sensor Editor, for the specification of sensors within YAWL process models. Such graphical component, shown in Fig. 7, fully supports the specification of sensor conditions as defined in Sect. 4.

[5] Available at `www.yawlfoundation.org`



Name: UnderpaymentFraud

Variables

Variables	Mapping	Resouce
GroupingElement	Issue_Shipment_Invoic...	
WindowElement	Issue_Debit_Adjustme...	
Threshold	0.6	
Probability	case(Issue_Debit_Adj...	

☑ Risk Condition: IDAStart&(Probability>Threshold)

☐ Fault Condition

Trigger
○ Sample Time 30 Seconds
● Event Issue_Debit_Adjustment_605 Started

Underpayment Fraud risk detected with a probability of $Probability$

Fault Message

Overall Sample Time: 30 Seconds

Sensor
OvertimeProcess
ApprovalFraud
OrderUnfulfillment
UnderpaymentFraud

Name: Probability

Mapping:
```
case(Issue_Debit_Adjustment_605(Count)>0 AND (ID)!=[IDCur
r]).Issue_Debit_Adjustment_605(Fraud, IDAnumIssue, 3, Group
ingElement, WindowElement, (5*24*60*60*1000))
```

Case
SubCase
Variable
☐ Resource
○ Role ○ Participant

Fig. 7. The Sensor Editor within the YAWL Editor.

Moreover, we implemented the Sensor Manager as a generic component which exposes three interfaces (engine, database and monitor) as described in Sect. 4. We then wrapped this component into a Web service which implements the three interfaces for the YAWL system, allowing the component to interact with the YAWL Engine, the Monitor service and the YAWL database. While there is a straightforward mapping between the YAWL Engine and our engine interface, and between the YAWL Monitor service and our monitor interface, we had to join several YAWL tables to implement our database interface. This is because in the YAWL system, event logs are scattered across different database tables. For example, to retrieve all identifiers of the process instances for a specific process model, given the model identifier, we need to perform a join among the following YAWL tables: `logspecification`, `lognetinstance`, `lognet` and `logevent`.

The complete mapping is illustrated in Tab. 2. As an example, this table also shows the mapping between our database interface and the relational schema used by Oracle BPEL 10g to store BPEL process logs. Also in this case, the database can be fully mapped by joining several tables.

Finally, we implemented a separate service to estimate the remaining cycle time T_e for a process or task instance. This service uses ProM's prediction miner [29] to compute the estimations, and provides the results to the Sensor Manager on demand. While the estimation of T_e could be done on-line, i.e. while evaluating a particular sensor condition at run-time, parsing the full logset each time would be inefficient. Rather, we compute this estimation off-line, whenever a new process model is deployed to the YAWL Engine, by using the logset available at that time. Periodically, we update the logset with the new instances being executed meantime, and invoke this service to refresh the estimations for each process model currently deployed.

5.2 Performance Analysis

We used our implementation to evaluate the scalability of the approach. First, we measured the time needed to evaluate the basic functions (e.g. counting the number of instances of a task or retrieving the resource allocated to a task). Next, we measured the

Database table	**Tables that need to be joined**	
	YAWL	**Oracle BPEL 10g**
WorkFlowDefinition	logspecification, lognet, lognetinstance, logevent	cube_instance and cube_scope
SubProcess	logspecification, lognet, lognetinstance, logevent	cube_instance and cube_scope
Activity	lognetinstance, logtask, logtaskinstance, lognet, logevent, logspecification, rs_eventlog	wftask and work_item
Variables	logtask, lognet, lognetinstance, logtaskinstance, logevent, logdataitem, logspecification	audit_trail, audit_detail and xml_document
Role	rs_participant	wftask
ActivityRole	rs_eventlog, logtaskinstance	wftask

Table 2. Database interface mapping for YAWL 2.2beta and Oracle BPEL 10g.

time needed to evaluate the sensor conditions for the risks defined in the Payment subprocess. The tests were run on an Intel Core I5 M560 2.67GHz processor with 4GB RAM running Linux Ubuntu 11.4. The YAWL logs were stored on the PostGres 9.0 DBMS. These logs contained 318 completed process instances from 36 difference process models, accounting for a total of 9,399 process events (e.g. task instance started and completed, variable's value change). Specifically, there were 100 instances from the Payment subprocess yielding a total of 5,904 process events. The results were averaged over 10 runs.

Basic function	**Description**	**OpenXES time [ms]**	**Database time [ms]**	**Reduction rate [%]**
net status	functions checking if a net status has been reached (isStarted, isCompleted)	6,535	18.9	99.71
net time	functions returning the time when a net status has been reached (startTime, completeTime, startTimeInMillis, completeTimeInMillis)	6,781	18.8	99.72
net variable	returns the value of a net variable	6,489	432.6	93.33
task count	number of times a task has been completed	803	19.8	97.53
task resource	functions that return the resources associated with a task (offerResource, allocateResource, startResource, completeResource)	850	20.9	97.54
task status	functions checking if a task status has been reached (isOffered, isAllocated, isStarted, isCompleted)	792	30.5	96.14
task time	functions returning the time when a task status has been reached (offerTime, allocateTime, startTime, completeTime, offerTimeInMillis, allocateTimeInMillis, startTimeInMillis, completeTimeInMillis)	824	22.3	97.29
task variable	returns the value of a task variable	787	96.7	87.71
task distribution	functions returning the resources associated with a task by default (offerDistribution, allocateDistribution, startDistribution, completeDistribution)	243		-
task initiator	functions returning the allocation strategy for a resource association (offerInitiator, allocateInitiator, startInitiator, completeInitiator)	249.6		-

Table 3. Performance of basic functions.

Table 3 shows the results of the evaluation of the basic functions provided by our language. In particular, in this table we compare the evaluation times obtained by accessing the YAWL logs via our database interface, with those obtained by accessing a serialization of the logs, e.g. in the OpenXES format. While OpenXES provides a simple and unique representation of a generic set of process logs, accessing an OpenXES file in real-time, i.e. during the execution of a process instance, is not feasible, due to the long access times (e.g. 6.5 sec. on average for evaluating a net variable). On the other hand, accessing the logs via our database interface, despite it requires the creation of a specific implementation for each BPMS database, provides considerably faster times than accessing OpenXES files (at least 87% gain w.r.t. OpenXES access). In fact, as we can see from Tab. 3, the evaluation times for all the basic functions are below 30 ms, apart from function `task variable`, which takes 100 ms and function `net variable`, which takes 430 ms.

The last two basic functions reported in Tab. 3, namely `task distribution` and `task initiator`, are evaluated in less than 250 milliseconds. These functions are not computed by accessing the logs, but rather by accessing information that is contained directly in an executable process model, e.g. the resources that are associated with a specific task. However, in our implementation we still use the database interface to access this information, in order to provide the developer with a single access point to all process-related data.

Table 4 reports the results of the evaluation of the sensor conditions defined for our running example. While the sensor conditions for the overtime process and order unfulfillment faults are very low (below 150 ms), longer times are obtained for evaluating the conditions for the two faults related to fraud. This is because both these conditions require to evaluate "complex queries", i.e. queries over the entire process logs: In the approval fraud, we need to retrieve all resources that approved an order for a specific customer, while in the underpayment fraud we need to retrieve all process instances where a debit adjustment was issued and aggregate these instances per customer. These queries are different than those needed to evaluate the basic functions, as the latter are performed on the events in the logs that are relative to a single known process instance, e.g. the instance for which the sensor condition is being evaluated.

The worst-case complexity of evaluating one such a complex query is still linear on the number of parameters that need be evaluated in the query (corresponding to the language element *CondExprSet* in Sect. 4) multiplied by the total number of instances present in the logs (corresponding to the size of table WorkflowDefinition addressed by our database interface).

Sensor	**Min** [ms]	**Max** [ms]	**Average** [ms]	**St.Dev.**
Overtime process	121	137	131.8	4.66
Approval fraud	6,483	7,036	6,766.4	183.06
Order unfulfillment	69	91	77.4	7.18
Underpayment fraud	3,385	3,678	3,523	89.98

Table 4. Performance of sensors.

In conclusion, the performances of evaluating sensor conditions should always be considered w.r.t. the specific process for which the risks are defined, and the type of trigger used. For example, let us assume an average duration of 24 hours for the Payment subprocess, with a new task being executed every 30 minutes. This means we have up to 30 minutes to detect an overtime process risk before a new task is executed, and we need to compute this sensor condition again. If we choose a rate of 5 minutes to sample this condition, we are well below the 6 minute-threshold, so we can check this sensor's condition up to 6 times during the execution of a task. Since we do this in less than 150 ms, this time is acceptable. For an event-driven risk we also need to consider the frequency of the specific event used as trigger. For example, the approval fraud risk is triggered every time an instance of task Approve Shipment Payment Order is offered to a Senior Financial Officer for execution. Since we take up to 7 seconds to compute this sensor condition, we are able to cope with a system where there is a request for approval every 7 seconds. So also for this sensor, the performance is quite acceptable.

6 Related Work

Risk measurement and mitigation techniques have been widely explored in various fields. At the strategic level risk management, standards prescribe generic procedures for identifying, analyzing, evaluating and treating risks (see e.g. [26]). Although helpful, such general guidelines are inevitably vague and fail to provide any specific guidance for operationalizing risk management strategies in business processes. At the other extreme, there are many techniques for identifying risks in specific areas such as employee fraud [1], conflict of interest [18] and in the engineering field more generally [12, 3]. Other approaches, such as fault-tree analysis [4], are general enough to be applied to multiple domains. However, none of these approaches provides insights on how to define and operationalize the detection of process-related risks.

Previous process-based research recognizes the importance of explicitly linking elements of risk to business process models. zur Muehlen et al. [23, 32] propose a taxonomy of process-related risks and describe its application in the analysis and documentation of business processes. This taxonomy includes five process-related risk types (goals, structure, information technology, data and organization) which can be captured by four interrelated model types: i) risk structure model describing the relationships between risks; ii) risk/goal matrix; iii) risk state model describing the dynamic aspects of a risk; and iv) an extension to the EPC notation to assign risks to individual process steps. An extension of the work in [23] is proposed in [20], where the authors describe a four-step approach to integrate risks in business processes at the operational and strategic levels via *value-focused process engineering*.

A different perspective is offered by the ROPE (Risk-Oriented Process Evaluation) methodology [10, 28]. ROPE is based on the observation that process activities require resources to be adequately executed. If faults occur (here called "threats"), they impact the functionality of resources until one or more affected resources are no longer available. In the worst case a resource represents a single point of failure and consequently hinders the execution of the related process activity. If a threat is detected, an appropriate countermeasure process is invoked to counteract the threat. However, if this cannot be done, a recovery process can be invoked to re-establish the functionality of the affected resources until they are available again for the respective business process activity. The aim of the ROPE methodology is to incorporate all these aspects in a single model that can be simulated to determine a company's critical business processes and single points of failure. Finally, on the basis of the ROPE methodology, a reference model for risk-aware BPM is proposed in [15, 16].

With respect to the risk-aware BPM lifecycle shown in Fig. 3, all the above proposals only cover the phases of risk analysis and risk-aware process modeling. None of them specifies how risk conditions can be concretely linked to run-time aspects of process models such as resource allocation, data variables and control-flow conditions, for the sake of detecting risks during process execution. Thus, none of these approaches operationalizes risk detection into workflow management systems. Moreover, they neglect historical process data for risk estimation. As such, these approaches are complementary to our work, i.e. they can be used at a conceptual level for the identification of process-related risks, which can then be implemented via our sensor-based technology.

Our sensor-based architecture is also related to real-time monitoring of business process execution. Similarly to our approach, Oracle Business Activity Monitoring (BAM) [22] relies on sensors to monitor the execution of BPEL processes. Three types of sensors can be defined: *activity sensors*, to grab timings and variable contents of a specific activity; *variable sensors*, to grab the content of the variables defined for whole BPEL process (e.g. the inputs to the process); and *fault sensors*, to monitor BPEL faults. These sensors can be triggered by a predefined set of events (e.g. task activation, task completion). For each sensor, one can specify the endpoints where the sensor will publish its data at run-time (e.g. a database or a JMSQueue). We allow the specification of more sophisticated sensor (and fault) conditions, where different process-related aspects can be incorporated such as data, resource allocation strategies, order dependencies, as well as historical data and information from other running process instances. Moreover, our sensors can be triggered by process events or sampled at a given rate. Nonetheless, our sensor-based architecture is exposed as a service and as such it could be integrated with other process monitoring systems, such as Oracle BAM.

Real-time monitoring of process models can also be achieved via Complex Event Processing (CEP) systems. In this context, CEP systems have been integrated into commercial BPMSs, e.g. webMethods Business Events[6], ARIS Process Event Monitor [6] and SAP Sybase [27], as well as explored in academia [9, 11]. A CEP system allows the analysis of aggregated events from different sources (e.g. databases, email accounts as well as process engines). Using predefined rules, generally defined with a specific SQL-like language [31], a CEP system can verify the presence of a specific pattern among a stream of simple events processed in a given time window. Our approach differs from CEP systems in the following aspects: i) strong business process orientation vs general purpose system; ii) ability to aggregate and analyze complex XML-based events (e.g. process variables) vs simple events; iii) time-driven and event-driven triggers vs event-driven trigger only. Moreover, CEP systems typically suffer from performance overheads [11, 31] which limit their applicability to real-time risk detection [31].

7 Conclusion

The contribution of this paper is twofold. First, it provides a concrete mechanism for identifying risks in executable business process models and for detecting them during process execution. This is achieved by embedding elements of risk within each phase of the BPM lifecycle: from process design, where high-level risks are mapped down to specific process model elements, to process diagnosis, where risk conditions are monitored in real-time. The second contribution is an operationalization of the proposed risk-awareness approach in the context of BPMSs. This is achieved via a distributed, sensor-based architecture that is interfaced with a BPMS via a set of interfaces. Each risk is associated with a sensor condition. Conditions can relate to any process aspect, such as control-flow dependencies, resource allocations, the content of data elements, both from the current process instance and from instances of any process that have already been completed. At design-time, these conditions are expressed via a Java-like query language within a process model. At run-time, each sensor independently alerts a sensor manager when the associated risk condition evaluates to true during the execution of a specific process instance. When this occurs, the sensor manager notifies a process administrator about the given risk by interfacing with the monitoring service

[6] `http://www.softwareag.com/au/products/wm/events/overview/default.asp`

of the BPMS. This allows early risk detection which in turn enables proper remedial actions to be taken in order to avoid potentially costly process faults.

The sensor-based architecture was implemented in the YAWL system and its performance evaluated in practice. The tests show that the sensor conditions can be computed efficiently and that no performance overhead is induced to the BPMS engine. To the best of our knowledge, this is the first attempt to embed risks into executable business processes and enable their automatic detection at run-time.

This work suffers from several limitations, which provide opportunities for future work. First, it does not support the actual risk mitigation but only risk detection. We plan to devise a mechanism for automatically generating remedial actions that can be applied once a risk has been detected at run-time. The idea is to use genetic algorithms such as simulated annealing [24] to create perturbations on the current process instance in order to rectify its execution and thus avoid a fault from eventually occurring. Our previous application of simulated annealing to the problem of automatically correcting business process models [8] has shown that such perturbations can be obtained very efficiently. The challenge stands in properly defining the objective functions so as to create meaningful perturbations. Second, the approach's usefulness and ease of use have not been evaluated in practice. In this regard, we plan to interview a pool of risk analysts drawn from our business contacts in Australia. Finally, we plan to equip the risk modeling component with a set of predefined risks, categorized by type and domain (e.g. approval fraud), which can be used as templates to generate skeletons of risk conditions.

Acknowledgments We thank Colin Fidge for his valuable comments on an early version of this paper, and Michael Adams for his help with the YAWL implementation. This research is partly funded by the ARC Discovery Project "Risk-aware Business Process Management" (DP110100091).

References

1. W.S. Albrecht, C.C. Albrecht, and C.O. Albrecht. *Fraud Examination*. South-Western Publishing, 3rd edition, 2008.
2. Basel Committee on Bankin Supervision. *Basel II - International Convergence of Capital Measurement and Capital Standards*, 2006.
3. N. Bhushan and K. Rai. *Strategic Decision Making: Applying the Analytic Hierarchy Process*. Springer, 3rd edition, 2004.
4. International Electrotechnical Commission. *IEC 61025 Fault Tree Analysis (FTA)*, 1990.
5. R. Conforti, G. Fortino, M. La Rosa, and A.H.M. ter Hofstede Hofstede. History-aware, real-time risk detection in business processes (extended version). QUT ePrints 42222, Queensland University of Technology, http://eprints.qut.edu.au/42222, 2011.
6. R.B. Davis and E. Brabander. *ARIS Design Platform: Getting Started with BPM*. Springer, 2007.
7. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
8. M. Gambini, M. La Rosa, S. Migliorini, and A.H.M. ter Hofstede. Automatic error correction of business process models. In *BPM*. Springer, 2011.
9. P. Gay, A. Pla, B. López, J. Meléndez, and R. Meunier. Service workflow monitoring through complex event processing. In *ETFA*. IEEE, 2010.
10. G. Goluch, S. Tjoa, S. Jakoubi, and G. Quirchmayr. Deriving resource requirements applying risk-aware business process modeling and simulation. In *ECIS*. AISeL, 2008.

11. G. Hermosillo, L. Seinturier, and L. Duchien. Using Complex Event Processing for Dynamic Business Process Adaptation. In *SCC*. IEEE, 2010.
12. R. Hespos and P. Strassmann. Stochastic Decision Trees for the Analysis of Investment Decisions. *Management Science*, 11(10), 1965.
13. A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2010.
14. D. Hollingsworth. The Workflow Reference Model. Workflow Management Coalition, 1995.
15. S. Jakoubi and S. Tjoa. A reference model for risk-aware business process management. In *CRiSIS*. IEEE, 2009.
16. S. Jakoubi, S. Tjoa, S. Goluch, and G. Kitzler. Risk-aware business process management: Establishing the link between business and security. In *Complex Intelligent Systems and Their Applications*, volume 41 of *Optimization and its Applications*. Springer, 2010.
17. W.G. Johnson. *MORT - The Management Oversight and Risk Tree*. U.S. Atomic Energy Commission, 1973.
18. A. Little and P. Best. A framework for separation of duties in an sap r/3 environment. *Managerial Auditing Journal*, 18(5):419–430, 2003.
19. B. Meyer. *Introduction to the theory of programming languages*. Prentice-Hall, 1990.
20. D. Neiger, L. Churilov, M. zur Muehlen, and M. Rosemann. Integrating risks in business process models with value focused process engineering. In *ECIS*. AISeL, 2006.
21. OMG. *Business Process Model and Notation (BPMN) ver. 2.0*, January 2011. http://www.omg.org/spec/BPMN/2.0.
22. Oracle. *BPEL Process Manager Developer's Guide*, http://download.oracle.com/docs/cd/E15523_01/integration.1111/e10224/bp_sensors.htm. Accesssed: June 2011.
23. M. Rosemann and M. zur Muehlen. Integrating risks in business process models. In *ACIS*. AISeL, 2005.
24. K.I. Smith, R.M. Everson, J.E. Fieldsend, C. Murphy, and R. Misra. Dominance-based multiobjective simulated annealing. *IEEE Trans. on Evolutionary Computation*, 12(3), 2008.
25. M. Soldal Lund, B. Solhaug, and K. Stolen. *Model-Driven Risk Analysis*. Springer, 2011.
26. Standards Australia and Standards New Zealand. *Standard AS/NZS ISO 31000*, 2009.
27. Sybase. *Sybase CEP Implementation Methodology for Continuous Intelligence*, http://www.sybase.com.au/files/White_Papers/Sybase_CEP_Implementation_Methodology_wp.pdf. Accessed: June 2011.
28. S. Tjoa, S. Jakoubi, and G. Quirchmayr. Enhancing business impact analysis and risk assessment applying a risk-aware business process modeling and simulation methodology. In *ARES*, pages 179–186. IEEE Computer Society, 2008.
29. B.F. van Dongen, R.A. Crooy, and W.M.P. van der Aalst. Cycle time prediction: When will this case finally be finished? In *CoopIS*, pages 319–336. Springer, 2008.
30. Voluntary Interindustry Commerce Solutions Association. *Voluntary Inter-industry Commerce Standard (VICS)*. http://www.vics.org. Accessed: June 2011.
31. D. Wang, E.A. Rundensteiner, R.T. Ellison, and H. Wang. Active complex event processing infrastructure: Monitoring and reacting to event streams. In *ICDEW*. IEEE, 2011.
32. M. zur Muehlen and D.T.-Y. Ho. Risk management in the bpm lifecycle. In *BPM Workshops*. Springer, 2006.