Dynamic Restructuring of Recovery Nets

Rachid Hamadi **Boualem Benatallah**

School of Computer Science and Engineering
The University of New South Wales
Sydney NSW 2052, Australia
{rhamadi,boualem}@cse.unsw.edu.au

Abstract

A Self-Adaptive Recovery Net (SARN) is an extended Petri net model for specifying exceptional behavior in workflow systems. SARN caters for high-level recovery policies that are incorporated either with a single task or a set of tasks, called a recovery region. A recovery region delimits the part of the workflow from which the associated recovery policies take place. In this paper, we assume that SARN is initially partitioned into recovery regions by workflow designers who have a priori expectations for how exceptions will be handled. We propose a pattern-based approach to dynamically restructure SARN partition. The objective is to continuously restructure recovery regions within SARN partition to reflect the dynamic changes in handling exceptions. The restructuring of SARN partition is based on the observation of predefined recovery patterns.

Keywords: Self-Adaptive Recovery Net (SARN), workflow systems, recovery region, predefined recovery patterns, region restructuring operations.

1 Introduction

A workflow management system (WfMS) provides a central control point for defining business processes and orchestrating their execution (Georgakopoulos, Hornick & Sheth 1995). A major limitation of current WfMSs is their lack of support for dynamic adaptations (Casati, Ceri, Pernici & Pozzi 1998). Supporting the handling of exceptions is of great importance in areas such as clinical environment, office automation, flexible manufacturing systems, and design of production lines.

In our previous work, we introduced Self-Adaptive Recovery Net (SARN), an extended Petri net model, for specifying exceptional behavior in workflow systems (Hamadi & Benatallah 2004). This model adapts the structure of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple. It caters for high-level recovery policies that are incorporated either with a single task or a set of tasks called a *recovery region*. A recovery region delimits the part of the workflow from which the associated recovery policies take place. Recovery policies are generic directives that model exceptions at design time together with a set of primitive operations used at run time to handle the occurrence of exceptions.

In this paper, we assume that SARN is initially partitioned into recovery regions by workflow designers who have a priori expectations for how exceptions will be handled. We propose a pattern-based approach to dynamically restructure SARN partition. The objective is to continuously restructure recovery regions within SARN partition by being responsive to the ways past exceptions have been handled. The restructuring of SARN partition is based on the observation of predefined recovery patterns. Our contribution is twofold:

- *Predefined recovery patterns.* These patterns represent pre-identified sequences of recovery procedures. They are used as heuristics for SARN partition restructuring. The discovery of frequent patterns helps workflow administrators decide what kind of restructuring would be desirable to improve SARN partitioning. Transformations of SARN partition over time result in offering improved alternative of its structure based on the observation of recovery patterns.
- *Region restructuring operations.* We propose two groups of region restructuring operations, namely *split* operations and *merge* operations. They are used to dynamically transform the structure of SARN partition.

Region restructuring should only be allowed in a valid way. The system must guarantee the correctness of the modified SARN partition w.r.t. consistency constraints (such as reachability and absence of deadlock), so that constraints that were valid before the dynamic modification of SARN are also valid after it.

The rest of the paper is organized as follows. Section 2 describes the partitioning of the SARN model into recovery regions along with a motivating example. Section 3 introduces the predefined recovery patterns. Their corresponding region restructuring operations are discussed in Section 4. Section 5 describes the *HiWorD* (HIerarchical WORkflow Designer) prototype. Finally, Section 6 reviews some related work and concludes the paper.

2 SARN Partitioning

In this section, we first recall the definition of Self-Adaptive Recovery Net (SARN). Then, we introduce the notions of recovery region and SARN partition. Finally, we illustrate these concepts by an example.

2.1 SARN Definition

SARN extends Petri nets (Murata 1989, Peterson 1981) to model exception handling through *recovery* transitions and *recovery* tokens. In SARN, there are two types of transitions: *standard* transitions representing workflow tasks to be performed and *recovery* transitions that are associated with workflow tasks to

adapt the recovery net in progress when an exception event occurs. There are also two types of tokens: *standard* tokens for the firing of standard transitions and *recovery* tokens associated with recovery policies for the firing of recovery transitions. There is one recovery transition per type of task exception, that is, when a new recovery policy is designed, a new recovery transition is added. When an exception (such as a time out) within a task occurs, an event is raised and a recovery transition will be enabled and fired. The corresponding sequence of basic operations (such as creating a place and deleting an arc) associated with the recovery transition is then executed to adapt the structure of SARN that will handle the exception.

In the following, we recall the formal definition of SARN (Hamadi & Benatallah 2004).

Definition 2.1 (SARN)
A Self-Adaptive Recovery Net (SARN) is a tuple $RN = (P, T, Tr, F, i, o, \ell, M)$ *where:*

- P *is a finite set of places representing the states of the workflow,*
- T *is a finite set of standard transitions* ($P \cap T = \varnothing$) *representing the tasks of the workflow,*
- Tr *is a finite set of recovery transitions* ($T \cap Tr = \varnothing$ *and* $P \cap Tr = \varnothing$) *associated with workflow tasks to adapt the net in-progress when an exception event occurs. There is one recovery transition per type of task exception,*
- $F \subseteq (P \times (T \cup Tr)) \cup ((T \cup Tr) \times P)$ *is a set of directed arcs (representing the control flow),*
- i *is the input place of the workflow with* ${}^\bullet i = \varnothing$,
- o *is the output place of the workflow with* $o^\bullet = \varnothing$,
- $\ell : T \rightarrow \mathcal{A} \cup \{\tau\}$ *is a labeling function where* $\mathcal{A}$ *is a set of task names.* τ *denotes a silent (or an empty) task (represented as a black rectangle), and*
- $M : P \rightarrow \mathbb{N} \times \mathbb{N}$ *represents the mapping from the set of places to the set of integer pairs where the first value is the number of standard tokens (represented as black small circles within places) and the second value the number of recovery tokens (represented as black small rectangles within places).* □

Recall that for $x \in P \cup T$, the *pre-set* of x is defined as ${}^\bullet x = \{y \in P \cup T \ / \ (y, x) \in W\}$ and the *post-set* of x is defined as $x^\bullet = \{y \in P \cup T \ / \ (x, y) \in W\}$.

In the SARN model, there are some primitive operations that can modify the net structure such as adding an arc and disabling a transition (see (Hamadi & Benatallah 2004)). Several of these basic operations are combined in a specific order to handle different exception events. The approach adopted is to use one recovery transition to represent one type of exception. Thus, a recovery transition represents a combination of several basic operations. When an exception occurs, a recovery token is injected into a place and the set of primitive operations associated with the recovery policy are triggered.

2.2 Recovery Region

A *recovery region* is a connected set of places and transitions. A recovery region has one input place and one output place. The output place of a recovery region is typically an input place for the eventual next recovery region(s). To avoid an overlapping of recovery regions, we will separate them so that between the output place of a recovery region and the input place of the eventual subsequent recovery region(s), a silent transition transfers the token from the recovery region to the subsequent recovery region(s).

A *recovery region* is then a subworkflow that can be seen as a unit of work from the business perspective and to which a set of region-based recovery policies may be assigned. A default region-based recovery policy *CompensateRegion* can be associated with any recovery region. *CompensateRegion* removes the effects of all already executed tasks of the completed or running recovery region. For more details about region-based recovery policies see Table 1 and refer to (Hamadi & Benatallah 2004). As such, a recovery region is more than a traditional subworkflow. Formally, a *recovery region* is defined as follows.

Definition 2.2 (Recovery Region)
Let $RN = (P, T, Tr, F, i, o, \ell, M)$ *be a SARN net. A recovery region is a subnet* $R = \langle P_R, T_R, Tr_R, F_R, i_R, o_R, \ell_R, S_R \rangle$ *of* RN *where:*

- $P_R \subseteq P$ *is the set of places of the recovery region,*
- $T_R \subseteq T$ *denotes the set of transitions of the recovery region* R,
- $Tr_R \subseteq Tr$ *denotes the set of task recovery transitions of* R,
- $F_R \subseteq F$ *represents the control flow of* R,
- $i_R \in P_R$ *is the input place of* R,
- $o_R \in P_R$ *is the output place of* R,
- $\ell_R : T_R \rightarrow \mathcal{A} \cup \{\tau\}$ *is a labeling function,*
- S_R *is a finite set of region recovery transitions associated with the recovery region* R *to adapt the net in-progress when a region exception event occurs. There is one recovery transition per type of region exception, and*
- *Let* $T_R = \{t \in T \mid t^\bullet \cap P_R \neq \varnothing \ \wedge \ {}^\bullet t \cap P_R \neq \varnothing\}$. *Then* R *must be connected (i.e., there must not be any isolated places or transitions).* □

R represents the underlying Petri net of the recovery region that is restricted to the set of places P_R and the set of transitions T_R.

2.3 SARN Partition

For a specific SARN net, there can be several different ways to divide it into recovery regions. A *SARN partition* is a division of the set of places into pairwise disjoint recovery regions. Formally, a SARN partition is defined as follows.

Definition 2.3 (SARN Partition)
Let $RN = (P, T, Tr, F, i, o, \ell, M)$ *be a SARN net. A SARN partition* $\Pi = \{R_k \mid k = 1, .., n\}$ *of the set* P *of places is a decomposition into non-empty and disjoint recovery regions* R_k, $k = 1, ..n$, *that is:*

- $\forall \ \ k = 1, .., n \ \ R_k \neq \varnothing$,
- $\forall \ \ k = 1, .., n \ \ R_k$ *contains one* input *place,*
- $\forall \ \ k = 1, .., n \ \ R_k$ *contains one* output *place,*
- $\bigcup_{k=1}^{n} R_k = P$,
- $\forall\ k, j \in \{1, .., n\},\ k \neq j \ \ R_k \cap R_j = \varnothing$,
- *For each* R_k, *let* $T_k = \{t \in T \mid t^\bullet \cap R_k \neq \varnothing \ \wedge \ {}^\bullet t \cap R_k \neq \varnothing\}$. *Then* $\forall\ k, j \in \{1, .., n\},\ k \neq j \ \ T_k \cap T_j = \varnothing$. □

The set T_k denotes the set of transitions of the recovery region R_k. The last condition means that there are no shared transitions between recovery regions.

Table 1: Region-Based Recovery Policies

Recovery Policy	Notation	Region Status	Brief Description
`SkipRegion(Event e, Region R)`	SR^e_R	Running	Skips the running task(s) of the region `R` to the immediate next task(s) of it if the event `e` occurs
`SkipRegionTo(Event e, Region R, TaskSet` $\mathcal{T}$`)`	$SRT^e_{R,\mathcal{T}}$	Running	Skips the running region `R` to the specific next task(s) $\mathcal{T}$ if the event `e` occurs
`CompensateRegion(Event e, Region R)`	CR^e_R	Completed or Running	Removes the effect of all already executed tasks of the completed or running region `R` if the event `e` occurs
`CompensateRegionAfter(Event e, Region R)`	CRA^e_R	Completed	Removes the effect of an already executed region `R` just after completing it if the event `e` occurs
`RedoRegion(Event e, Region R)`	RR^e_R	Completed or Running	Repeats the execution of all already completed tasks of the completed or running region `R` if the event `e` occurs
`RedoRegionAfter(Event e, Region R)`	RRA^e_R	Completed	Repeats the execution of an already completed region `R` just after it ends if the event `e` occurs
`AlternativeRegion(Event e, Region R, Region R')`	$AR^e_{R,R'}$	Running	Allows an alternative execution of a region `R` by another region `R'` if the event `e` occurs
`TimeoutRegion(Region R, Time d)`	TR^d_R	Running	Fails a region `R` if not completed within a time limit `d`. The execution is frozen

2.4 A Motivating Example

We give here an example of a workflow representing ordering flight tickets over the Internet to illustrate the concept of a recovery region (see Figure 1).

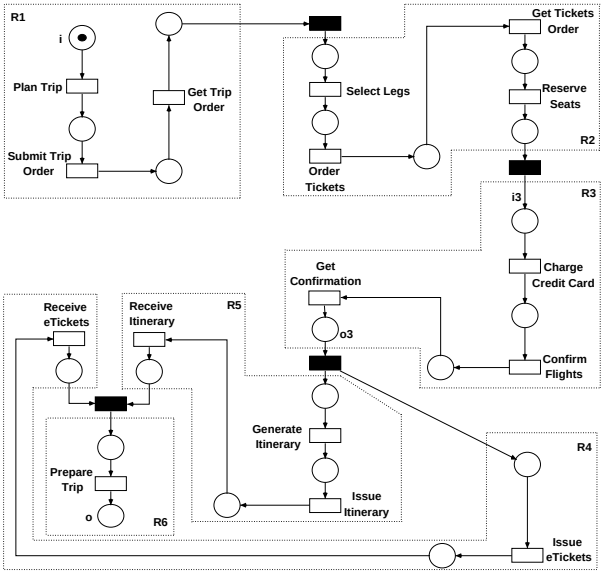

Figure 1: The *Ordering Flight Tickets* Workflow as a SARN Partition

A customer plans her trip by specifying the various stages of the overall journey (the *Plan Trip* task). She sends this information together with the list of all participants of the trip and the information about the credit card to be charged for the ordered tickets to the travel agent (the *Submit Trip Order* task). She then waits for the submission of the electronic tickets (the *Receive eTickets* task) as well as the final itinerary for the trip (the *Receive Itinerary* task) before preparing for the trip by making other arrangements such as hotel booking and car rental (the *Prepare Trip* task). When the travel agent receives the customer's trip order (the *Get Trip Order* task), he will determine the legs for each of the stages (the *Select Legs* task) and submits these legs together with the information about the credit card to be charged to the airline company (the *Order Tickets* task). The agent then waits for the confirmation of the flights (the *Get Confirmation* task), which includes the actual seats reserved for each participant. This information is completed into an itinerary (the *Generate Itinerary* task) and sent to the customer (the *Issue Itinerary* task). When the airline receives the tickets order submitted by the agent (the *Get Tickets Order* task), the requested seats will be checked and, if available, assigned to the corresponding participants (the *Reserve Seats* task). After that, the credit card will be charged (the *Charge Credit Card* task), and the updated leg information sent back to the agent as confirmation of the flights (the *Confirm Flight* task). Finally, once the travel agent receives confirmation, the airline sends the electronic tickets by e-mail to the customer (the *Issue eTickets* task).

We use our *HiWorD* (HIerarchical WORkflow Designer) tool to model workflows (Benatallah, Chrzastowski-Wachtel, Hamadi, O'Dell & Susanto 2003). For the *Ordering Flight Tickets* workflow example (see Figure 1), starting from a single place and using a succession of place and transition refinement rules (Chrzastowski-Wachtel, Benatallah, Hamadi, O'Dell & Susanto 2003), the abstract workflow shown in Figure 2 is obtained. Places represent recovery regions and black rectangles stand for silent (or empty) transitions. Silent activities, called *dummy* activities in (WfMC 1999), have no associated work or re-

sources.

By further refining regions (i.e., places) R1 through R6, we obtain the flat workflow represented in Figure 1 as a SARN partition where recovery regions are surrounded by dotted lines.

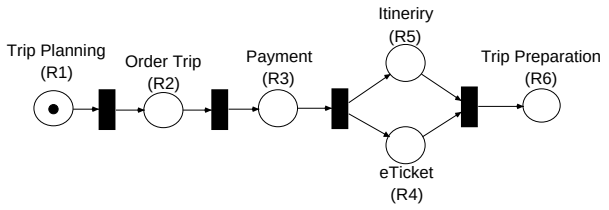

Figure 2: The *Ordering Flight Tickets* Workflow as an Abstract SARN Partition

3 Predefined Recovery Patterns

In this section, we discuss predefined recovery patterns that suggest how to dynamically restructure SARN partition based on the recorded previous executions.

It is possible to *split* a recovery region into two recovery sub-regions or *merge* two consecutive recovery regions. An example where splitting a recovery region into two successive recovery sub-regions is advisable is when exception events and activities involved in their recovery always occur in one specific part of the recovery region. This avoids undoing and redoing a large and expensive amount of work, and the possibility of cascaded compensations. Another example where merging two consecutive recovery regions into one recovery region is suitable is when *RedoRegion* exception events occurring in one recovery region always involve activities of the previous recovery region for their handling.

It should be noted that merging recovery regions is only appropriate when frequent pre-identified sequences of similar recovery procedures have been observed. Furthermore, the approach applies only to *CompensateRegion*, *CompensateRegionAfter*, *RedoRegion*, and *RedoRegionAfter* recovery policies.

We first introduce some definitions and then discuss the predefined recovery patterns.

3.1 Workflow Execution Log

Data mining is the task of discovering valuable information in data (Berry & Linoff 2000). Our type of data is relational databases. For our purpose, we use the following format for the workflow log. Each execution of the workflow, recorded in the log file, is a sequence of events that include start time and completion time of each activity, the resources that executed it, its input data, and its output data.

Definition 3.1 (Execution Log)
The log of one execution of the workflow is a sequence of events $\langle W, I, A, T_s, T_c, AR, D_{in}, D_{out} \rangle$ *where:*

- W *is the identifier of the workflow,*
- I *is the identifier of the workflow instance,*
- A *is the identifier of the activity,*
- T_s *is the start time of the activity,*
- T_c *is the completion time of the activity,*
- AR *is the set of resources used by the activity,*
- D_{in} *is the set of input data of the activity, and*
- D_{out} *is the set of output data of the activity.* □

In order to use workflow log data when searching for recurring recovery patterns, we make some transformations to the workflow log file. First, we regroup all events belonging to the same workflow instance. Then, we put events within the same recovery region together. Finally, we explicitly highlight activities for which an exception was raised, together with the corresponding exception event condition evaluated to True and the region-based recovery policy. Note that since the actions taken to recover from the exception are defined in the recovery policy, it is not necessary to keep them in the processed log.

For example, the following sequence (taken from the processed log file of our example in Figure 1):

$$\langle\ \mathtt{R_1}, \mathtt{R_2}, \mathtt{R_3}(\mathtt{GetConfirmation}, \mathtt{Confirmation-NotReceived}, \mathtt{RedoRegion}), \mathtt{R_3}, \mathtt{R_4}, \mathtt{R_5}, \mathtt{R_6}\ \rangle$$

represents the execution of recovery regions R1, R2, and R3 with an occurrence of an exception during the execution of the activity Get Confirmation and the exception event condition Confirmation Not Received was evaluated to True, a recovery from it using a corresponding *RedoRegion* recovery policy as well as proceeding with the normal execution of R3, followed by the execution of R4, then R5, and, finally, R6.

The execution order of recovery regions in the processed log is partial since recovery regions are allowed to execute in parallel. The following two notions of *precedence* and *dependence* characterize the independence between recovery regions.

Definition 3.2 (Precedence)
Given a processed log of executions of the same workflow, we say that a recovery region R1 precedes *recovery region* R2 *if, in each execution they both appear, either* R2 *starts after* R1 *terminates or there exists a recovery region* R3 *such that* R1 *precedes* R3 *and* R3 *precedes* R2. □

Definition 3.3 (Dependence)
Given a processed log of executions of the same workflow:

- *If a recovery region* R1 *precedes a recovery region* R2 *but* R2 *does not precede* R1 *then* R2 depends *on* R1.
- *If* R2 *precedes* R1 *and* R1 *precedes* R2*, or* R2 *does not precede* R1 *and* R1 *does not precede* R2 *then* R1 *and* R2 *are* independent. □

For instance, if we have the previous sequence together with the following sequence:

$$\langle\ \mathtt{R_1}, \mathtt{R_2}, \mathtt{R_3}(\mathtt{GetConfirmation}, \mathtt{Confirmation-NotReceived}, \mathtt{RedoRegion}), \mathtt{R_3}, \mathtt{R_5}, \mathtt{R_4}, \mathtt{R_6}\ \rangle$$

then the recovery regions R4 and R5 are independent and hence can be executed concurrently (see Figure 1).

3.2 Recovery Patterns

In our approach, we use *predefined recovery patterns* (PRPs) to help identify situations where the structure of SARN partition could be improved through region restructuring operations that will be defined in Section 4. A particular sequence of recovery regions with widespread occurrences should be recognized as a recurring PRP, which we will refer to as a *pattern*. Each identified frequent pattern suggests a restructuring operation. A PRP is formally defined as follows:

Definition 3.4 (PRP)
A `Predefined Recovery Pattern (PRP)` *is a partially ordered sequence of recovery region executions (the same recovery region may appear more than once if exceptions occurred within the recovery region):*

$$\mathtt{PRP} = \langle\ \mathtt{R}_1, ..., \mathtt{R}_{k-1}, \mathtt{R}_k(\mathtt{t}^1_k, \mathtt{e}^1_k, \mathtt{r}^1_k), ...,$$

$$\mathtt{R}_k(\mathtt{t}^p_k, \mathtt{e}^p_k, \mathtt{r}^p_k), \mathtt{R}_k, ..., \mathtt{R}_n\ \rangle$$

where R_k $(k = 1, .., n)$ represents the execution of some activities within the recovery region R_k and, eventually, $\mathtt{t}^j_k$ (respectively $\mathtt{e}^j_k$ and $\mathtt{r}^j_k$) $(j = 1, .., p)$ is the failed task (respectively the corresponding exception event condition and recovery policy) within the recovery region R_k. □

The PRP can also be generated by the following Place/Transition net, called an *abstract SARN partition* obtained from SARN partition by abstracting the content of recovery regions by reducing an entire recovery region into a place:

Definition 3.5 (Abstract SARN Partition)
Let $RN = (P, T, Tr, F, i, o, \ell, M)$ be a SARN net. The `Abstract SARN Partition` *of the SARN partition $\Pi = \{R_k \mid k = 1, .., n\}$, where $R_k = \langle P_k, T_k, Tr_k, F_k, i_k, o_k, \ell_k, S_k \rangle$, is a tuple $RN_a = (P_a, T_a, S_a, F_a, \ell_a)$ where:*

- *$P_a = \Pi$ is the set of places of the abstract SARN partition,*
- *$T_a = T \setminus \bigcup_{k=1}^{n} T_k$ (recall that T_k is the set of transitions of the recovery region R_k) is the set of transitions of the abstract SARN partition,*
- *$S_a = \bigcup_{k=1}^{n} S_k$ is the set of region recovery transitions of RN_a,*
- *Let $F_a = F \setminus \bigcup_{k=1}^{n} F_k$ (recall that F_k is the flow relation within the recovery region R_k). For each $(p, t) \in F_a$ such that $p \in R_k$ do $F_a = (F_a \setminus \{(p, t)\}) \cup \{(R_k, t)\}$. For each $(t, p) \in F_a$ such that $p \in R_k$ do $F_a = (F_a \setminus \{(t, p)\}) \cup \{(t, R_k)\}$. The obtained F_a is the abstract flow relation, and*
- *$\ell_a : T_a \to \mathcal{A} \cup \{\tau\}$ is the abstract labeling function.* □

Figure 2 gives an example of such representation of the SARN partition of Figure 1. This abstract SARN partition clearly shows the recovery regions that are performed concurrently (in our case, `R4` and `R5`).

For a given PRP, there may exist a sequence Σ in the processed workflow log file such that the exact order of recovery regions in PRP can be found in Σ. A PRP is matched against each sequence to check whether the PRP exists. We refer to the number of occurrences of a given PRP in the log file as its *frequency*. In the following, we present the set of predefined recovery patterns that workflow administrators can use to restructure SARN partition.

3.2.1 Split Patterns

We distinguish between three different recovery region split PRPs: splitting a recovery region into (i) two successive recovery regions, (ii) two concurrent recovery regions, and (iii) two alternative recovery regions.

Sequential Split. Given a recovery region and a set of activities in the lower (or upper) part of this recovery region, the objective is to retrieve the frequency of exceptions that occurred within this recovery sub-region and the recovery from them involved only activities of this set (in addition, eventually, of some extra exception handling procedures of the recovery policy). Note that upper part means activities of this part are executed *before* the ones in the lower part. In this case, the recovery region may be split into two successive recovery sub-regions. The main advantage of splitting the recovery region is the reduction in cost of undoing and redoing a large and expensive amount of work, and the delimitation of the possibility of cascaded compensations. For the definition of this pattern, we distinguish between upper sequential split and lower sequential split.

Definition 3.6 (Upper Sequential Split PRP)
Given a recovery region R and a set U of activities in the upper part of R (i.e., the part that is executed first or before the remaining part), the recovery pattern for splitting the recovery region R into two sequential recovery sub-regions is:

$$\mathtt{PRP}_{\mathtt{USeqSplit}} = \langle\ \mathtt{R}(\mathtt{t}_1, \mathtt{c}_1, \mathtt{r}_1), ..., \mathtt{R}(\mathtt{t}_p, \mathtt{c}_p, \mathtt{r}_p), \mathtt{R}\ \rangle$$

where $p \geq 1$, $\mathtt{t}_i \in U$ $(i = 1, .., p)$ is the failed task in the upper part of R, and $\mathtt{c}_i$ (respectively $\mathtt{r}_i$) is the corresponding exception event condition (respectively recovery policy). □

Definition 3.7 (Lower Sequential Split PRP)
Given a recovery region R and a set L of activities in the lower part of R (i.e., the part that is executed last or after the remaining part), the recovery pattern for splitting the recovery region R into two sequential recovery sub-regions is:

$$\mathtt{PRP}_{\mathtt{LSeqSplit}} = \langle\ \mathtt{R}(\mathtt{t}_1, \mathtt{c}_1, \mathtt{r}_1), ..., \mathtt{R}(\mathtt{t}_p, \mathtt{c}_p, \mathtt{r}_1), \mathtt{R}\ \rangle$$

where $p \geq 1$, $\mathtt{t}_i \in L$ $(i = 1, .., p)$ is the failed task in the lower part of R, and $\mathtt{c}_i$ (respectively $\mathtt{r}_i$) is the corresponding exception event condition (respectively recovery policy). □

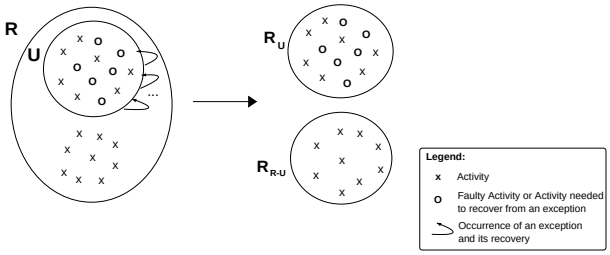

Figure 3: Upper Sequential Split

The predefined pattern $PRP_{USeqSplit}$ (respectively $PRP_{LSeqSplit}$) means that there has been at least one exception within the upper part U (respectively lower part L) of the recovery region R and the recovery from them involved only activities belonging to U (respectively L). If there are frequent occurrences of this PRP in the processed workflow log (w.r.t. a threshold specified by a workflow administrator), then it is recommended that the recovery region R should be split into two successive recovery regions R_U and $R_{R \setminus U}$ (respectively $R_{R \setminus L}$ and R_L) by applying an adequate sequential split operation such that tasks $\mathtt{t}_1, ..., \mathtt{t}_p$ belong to R_U (respectively R_L) as depicted in Figure 3 (respectively Figure 4).

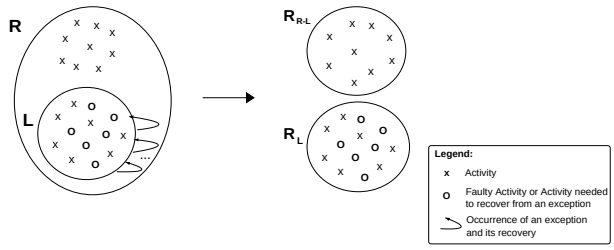

Figure 4: Lower Sequential Split

Parallel Split. Given a recovery region containing at least two parallel parts and a set of activities within one parallel part of this recovery region, the objective is to observe the occurrence of exceptions induced by this set of activities. This recovery region may be split into two parallel recovery sub-regions if frequent exceptions as well as activities involved in their recovery occur within the specified part of this recovery region. The benefit is the reduction of the overhead of focusing on a large recovery region when recovering from an exception as well as increasing concurrency.

Definition 3.8 (Parallel Split PRP)
Given a recovery region R containing at least two parallel parts and a set P of activities in one parallel part of R, the recovery pattern for splitting the recovery region R into two parallel recovery sub-regions is:

$$\mathtt{PRP}_{\mathtt{ParSplit}} = \langle\ \mathtt{R}(\mathtt{t}_1, \mathtt{c}_1, \mathtt{r}_1), ..., \mathtt{R}(\mathtt{t}_\mathtt{p}, \mathtt{c}_\mathtt{p}, \mathtt{r}_\mathtt{p}), \mathtt{R}\ \rangle$$

where $p \geq 1$, $\mathtt{t}_\mathtt{i} \in P$ $(i = 1, .., p)$ is the failed task in the specified parallel part of R, and $\mathtt{c}_\mathtt{i}$ (respectively $\mathtt{r}_\mathtt{i}$) is the corresponding exception event condition (respectively recovery policy). □

This PRP means that there is at least one exception within the specified parallel part P of the recovery region R, and if there are frequent occurrences of this pattern, it is suggested that the recovery region R should be split into two parallel recovery regions R_P and $R_{R \setminus P}$ such that the tasks $\mathtt{t}_1, ..., \mathtt{t}_\mathtt{p}$ belong to R_P (see Figure 5).

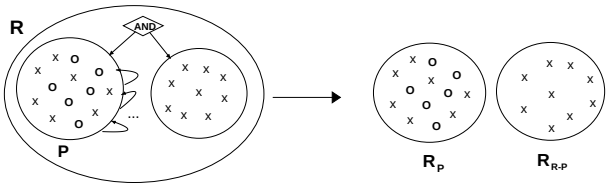

Figure 5: Parallel Split

Alternative Split. Similarly to a parallel split, for a given recovery region containing at least two alternative parts, the observation of repeated occurrences of exceptions as well as activities needed to recover from them within one alternative part of this recovery region suggests that it should be split it into two alternative recovery sub-regions.

Definition 3.9 (Alternative Split PRP)
Given a recovery region R containing at least two alternative parts and a set A of activities in one alternative part of R, the recovery pattern for splitting the recovery region R into two alternative recovery sub-regions is:

$$\mathtt{PRP}_{\mathtt{AltSplit}} = \langle\ \mathtt{R}(\mathtt{t}_1, \mathtt{c}_1, \mathtt{r}_1), ..., \mathtt{R}(\mathtt{t}_\mathtt{p}, \mathtt{c}_\mathtt{p}, \mathtt{r}_\mathtt{p}), \mathtt{R}\ \rangle$$

where $p \geq 1$, $\mathtt{t}_\mathtt{i} \in P$ $(i = 1, .., p)$ is the failed task in the specified alternative part of R, and $\mathtt{c}_\mathtt{i}$ (respectively $\mathtt{r}_\mathtt{i}$) is the corresponding exception event condition (respectively recovery policy). □

This recovery pattern means that there is at least one exception as well as the activities used for its recovery within the specified alternative part A of the recovery region R and if there are repeated occurrences of this PRP, the recovery region R is split into two alternative recovery regions R_A and $R_{R \setminus A}$ such that the tasks $\mathtt{t}_1, ..., \mathtt{t}_\mathtt{p}$ belong to R_A (see Figure 6).

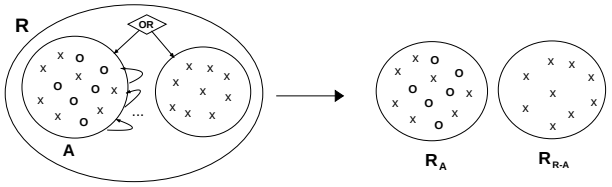

Figure 6: Alternative Split

3.2.2 Merge Patterns

We introduce now the predefined recovery patterns that describe situations where merging recovery regions can be beneficial. We distinguish two different recovery region merge patterns, sequential merge and parallel merge. The main benefit of merging recovery regions is to allow a hierarchical structure of recovery regions. For instance, if management levels are clearly identified then recovery regions can be designed in a way that reflects management levels, with certain privileges assigned to trigger region-based recovery policies at each level.

Sequential Merge. Given two successive recovery regions, the objective is to retrieve the number of occurrences of exceptions in one recovery region for which their handling using similar recovery policies involved activities from the previous recovery region in addition to activities within the faulty recovery region.

Definition 3.10 (Sequential Merge PRP)
Given two sequential recovery regions R_i followed by R_j, the recovery pattern for merging R_i and R_j into one recovery region is:

$$\mathtt{PRP}_{\mathtt{SeqMerge}} = \langle\ \mathtt{R}_\mathtt{i}, \mathtt{R}_\mathtt{j}(\mathtt{t}_1, \mathtt{c}_1, \mathtt{r}), ..., \mathtt{R}_\mathtt{j}(\mathtt{t}_\mathtt{p}, \mathtt{c}_\mathtt{p}, \mathtt{r}), \mathtt{R}_\mathtt{j}\ \rangle$$

where $p \geq 1$, $\mathtt{t}_\mathtt{k} \in R_j$ $(k = 1, .., p)$ is the failed task in the specified recovery region R_j, $\mathtt{c}_\mathtt{k}$ is the corresponding exception event condition, and $\mathtt{r}$ is the similar recovery policy. □

This PRP represents a situation where activities of one recovery region in two successive recovery regions are always involved when handling exceptions that happen in the other one. It is therefore interesting to regroup them into one recovery region to better delimit the parts of the workflow in which recovery policies take place.

Parallel Merge. Given two parallel recovery regions. When frequent occurrences of exceptions in one recovery region involve activities of the other recovery region to handle them using similar recovery policies, it may be better to merge them into one recovery region.

Definition 3.11 (Parallel Merge PRP)
Given two parallel recovery regions R_i and R_j, the recovery patterns for merging R_i and R_j into one recovery region are:

$$\texttt{PRP}_{\texttt{ParMerge1}} = \langle\ \texttt{R}_\texttt{i}, \texttt{R}_\texttt{j}(\texttt{t}_1, \texttt{c}_1, \texttt{r}), ..., \texttt{R}_\texttt{j}(\texttt{t}_\texttt{p}, \texttt{c}_\texttt{p}, \texttt{r}), \texttt{R}_\texttt{j}\ \rangle$$

$$\texttt{PRP}_{\texttt{ParMerge2}} = \langle\ \texttt{R}_\texttt{j}(\texttt{t}_1, \texttt{c}_1, \texttt{r}), ..., \texttt{R}_\texttt{j}(\texttt{t}_\texttt{p}, \texttt{c}_\texttt{p}, \texttt{r}), \texttt{R}_\texttt{j}, \texttt{R}_\texttt{i}\ \rangle$$

where $p \geq 1$, $\texttt{t}_\texttt{k} \in R_j$ $(k = 1, .., p)$ is the failed task in the specified recovery region R_j, $\texttt{c}_\texttt{k}$ is the corresponding exception event condition, and `r` *is the similar recovery policy.* □

These two patterns represent a situation where exceptions in one recovery region of two parallel recovery regions always need activities of the other recovery region to handle them. Regrouping them into one recovery region helps dealing with future occurrences of exceptions of similar recovery policies.

4 Region Restructuring Operations

By default, either the whole SARN net can be seen as one unique recovery region, or each task represents a recovery region by itself. The former means that when an exception occurs all activities of the workflow may be implicated to recover from it. The later implies that only the faulty activity is involved in handling the exception. But both extreme situations are not interesting since in the first case a huge amount of work will be needed and in the second case it is rare that an exception occurring in one activity has no impact or repercussion on other activities. Therefore, the workflow designer must define recovery regions precisely based on the nature of the business process as well as defining explicitly specific treatments for common exceptions (i.e., defining the recovery policies).

In the previous section, we proposed a set of predefined recovery patterns that help workflow administrators decide which operation to perform in order to restructure and adapt the SARN partition. In this section, a region restructuring operation corresponding to each pattern is defined. A region restructuring operation is applied to a valid SARN partition Π and produces a valid SARN partition Π'.

4.1 Split Operations

We distinguish between three different recovery region splitting operations: (i) sequential split, (ii) parallel split, and (iii) alternative split.

4.1.1 Sequential Split

The operation `SeqSplit(Region R, TaskSet V)` splits an existing recovery region `R` into two separate consecutive recovery regions `R1` and `R2` such that the given set of activities `V` (in the upper or lower part of `R`) belongs to `R1`.

Note that this restructuring operation corresponds to both upper sequential split PRP and lower sequential split PRP. As a result of a sequential split, one new recovery region is created out of the existing one. The definition of the existing recovery region is updated to reflect this change.

Definition 4.1 (SeqSplit Operation)
Let $R = \langle P_R, T_R, Tr_R, F_R, i_R, o_R, \ell_R, S_R \rangle$ be a recovery region of the SARN partition Π. The `SeqSplit(Region R, TaskSet V)` *operation transforms Π into Π' where:*

- $\Pi' = (\Pi \setminus \{R\}) \cup \{R1, R2\}$ *is the new SARN partition,*
- $T_{R1} = V$, *and*
- $T_{R2} = T_R \setminus V$. □

Figure 7 gives an example of a sequential split operation `SeqSplit(`R_3`,L)`, where `L` = { `ConfirmFlights`, `GetConfirmation` }, applied to the recovery region `R3` (see SARN partition of Figure 1).

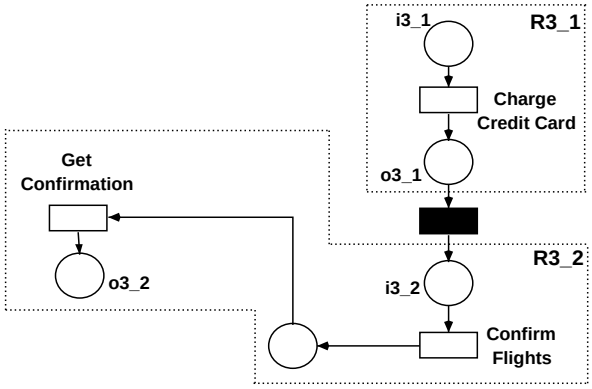

Figure 7: *SeqSplit* Operation Example

4.1.2 Parallel Split

The operation `ParSplit(Region R, TaskSet P)` splits an existing recovery region `R` into two parallel recovery regions `R1` and `R2` such that one of them contains the given set of activities `P`. As a result of a parallel split operation, one new parallel recovery region is created out of the existing one.

Definition 4.2 (ParSplit Operation)
Let $R = \langle P_R, T_R, Tr_R, F_R, i_R, o_R, \ell_R, S_R \rangle$ be a recovery region of the SARN partition Π. The `ParSplit(Region R, TaskSet P)` *operation transforms Pi into Π' where:*

- $\Pi' = (\Pi \setminus \{R\}) \cup \{R1, R2\}$ *is the new SARN partition,*
- $T_{R1} = P$, *and*
- $T_{R2} = T_R \setminus P$. □

4.1.3 Alternative Split

The operation `AltSplit(Region R, TaskSet A)` splits an existing recovery region `R` into two separate alternative recovery regions `R1` and `R2` such that the given set of activiries `A` belongs to one of them.

Definition 4.3 (AltSplit Operation)
Let $R = \langle P_R, T_R, Tr_R, F_R, i_R, o_R, \ell_R, S_R \rangle$ be a recovery region of the SARN partition Π. The `AltSplit(Region R, TaskSet A)` *operation transforms Π to Π' where:*

- $\Pi' = (\Pi \setminus \{R\}) \cup \{R1, R2\}$ *is the new SARN partition,*
- $T_{R1} = A$, *and*
- $T_{R2} = T_R \setminus A$. □

4.2 Merge Operations

Similarly to split operations, we distinguish two different recovery region merge operations: sequential merge and parallel merge.

4.2.1 Sequential Merge

The operation `SeqMerge(Region R1, Region R2)` merges two existing consecutive recovery regions `R1` and `R2` into one recovery region `R`.

Definition 4.4 (SeqMerge Operation) *Let* $R1 = \langle P_{R1}, T_{R1}, Tr_{R1}, F_{R1}, i_{R1}, o_{R1}, \ell_{R1}, S_{R1} \rangle$ *and* $R2 = \langle P_{R2}, T_{R2}, Tr_{R2}, F_{R2}, i_{R2}, o_{R2}, \ell_{R2}, S_{R2} \rangle$ *be two consecutive recovery regions of the SARN partition* Π. *The* `SeqMerge(Region R1, Region R2)` *operation transforms* Π *to* Π' *where:*

- $\Pi' = (\Pi \setminus \{R_1, R_2\}) \cup R$ *is the new SARN partition,*
- $P_R = P_{R1} \cup P_{R2}$, *and*
- $T_R = T_{R1} \cup T_{R2}$. □

4.2.2 Parallel Merge

The operation `ParMerge(Region R1, Region R2)` merges two existing parallel recovery regions `R1` and `R2` into one recovery region `R`.

Definition 4.5 (ParMerge Operation) *Let* $R1 = \langle P_{R1}, T_{R1}, Tr_{R1}, F_{R1}, i_{R1}, o_{R1}, \ell_{R1}, S_{R1} \rangle$ *and* $R2 = \langle P_{R2}, T_{R2}, Tr_{R2}, F_{R2}, i_{R2}, o_{R2}, \ell_{R2}, S_{R2} \rangle$ *be two parallel recovery regions of the SARN partition* Π. *The* `ParMerge(Region R1, Region R2)` *operation transforms* Π *into* Π' *where:*

- $\Pi' = (\Pi \setminus \{R_1, R_2\}) \cup R$ *is the new SARN partition,*
- $P_R = P_{R1} \cup P_{R2}$, *and*
- $T_R = T_{R1} \cup T_{R2}$. □

4.3 Discussion

Split and merge region restructuring operations can be combined. We may need to split a recovery region before merging one part of the result with another recovery region or merge recovery regions before splitting the result.

Note also that, for clarity of presentation, we described only two-recovery region split and merge operations. To split a recovery region into n recovery regions or merge n recovery regions, one can apply the two-recovery region restructuring operations successively $n-1$ times.

Table 2 summarizes the region restructuring operations and their corresponding pattern(s).

Table 2: Region Restructuring Operations

Operation	Patterns
`SeqSplit(Region R, TaskSet V)`	$PRP_{\texttt{USeqSplit}}$ $PRP_{\texttt{LSeqSplit}}$
`ParSplit(Region R, TaskSet P)`	$PRP_{\texttt{ParSplit}}$
`AltSplit(Region R, TaskSet A)`	$PRP_{\texttt{AltSplit}}$
`SeqMerge(Region R1, Region R2)`	$PRP_{\texttt{SeqMerge}}$
`ParMerge(Region R1, Region R2)`	$PRP_{\texttt{ParMerge1}}$ $PRP_{\texttt{ParMerge2}}$

5 *SARN Restructuring* Simulator

To illustrate the viability of our approach presented in this paper, we have developed *SARN Restructuring* simulator as part of *HiWorD* (HIerarchical WORkflow Designer) prototype, a hierarchical Petri net tool using Java (Benatallah et al. 2003).

HiWorD is a hierarchical workflow modeling tool with simulation capability. In addition to traditional workflow modeling, *HiWorD* allows, starting from a single place, the refinement of places and transitions to create hierarchical workflows. The tool also supports the concept of recovery region. A recovery region is considered as a refined place in *HiWorD* for which one or several recovery transitions are added and each recovery transition is associated with a region-based recovery policy (see Figure 8).

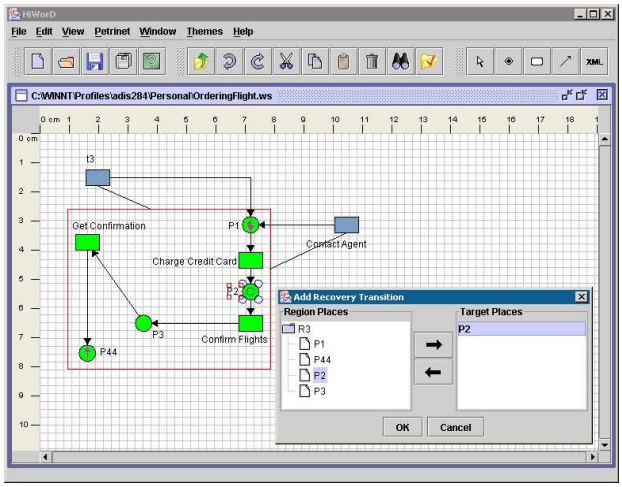

Figure 8: *HiWorD* Prototype

The *HiWorD* tool has been successfully used to hierarchically design workflow scenarios in a safe and effective way (Chrzastowski-Wachtel et al. 2003). It also includes modules that simulate the execution of a workflow, the occurrence of exceptions, and their handling. This information is logged in a relational database.

The *Workflow Engine Simulator* is responsible for the interpretation of the workflow specification and the execution of the workflow instances. It consists of five components (see Figure 9). A *Load Generator* that starts new workflow instances. The starting rate is a variable parameter of the simulation. An *Activity Module* which simulates the workflow activities. It receives input data from the *Workflow Engine Simulator*, waits for some time that corresponds to the duration of the activity, and delivers result back to the *Workflow Engine Simulator*. An *Exception Module* that simulates the occurrence of exceptions. An *Exception Handler* that recovers from exceptions. Finally, an *Audit Module* which logs information about the execution of the activities of the workflow such as start and end times of activities, exception events, and recovery procedures.

The architecture of *SARN Restructuring* simulator consists of four main components (see Figure 9): (i) *SARN Editor*, (ii) *Log Transformer*, (iii) *Pattern Recognition Engine*, and (iv) *Partition Restructuring Module*.

SARN Editor lets workflow designers build workflows as a SARN partition. *Log Transformer* transforms the logged data into a suitable format as discussed in Section 3.1. An administrator can operate this module periodically.

An administrator uses *SARN Admin* to restruc-

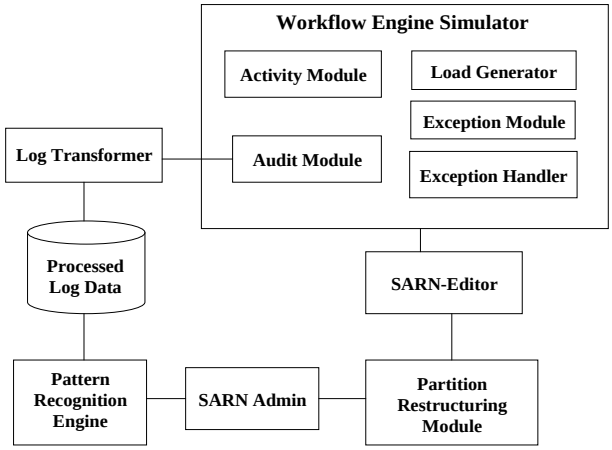

Figure 9: *SARN Restructuring* Simulator Architecture

ture SARN Partition. This component interacts with *Pattern Recognition Engine* and *Partition Restructuring Module*. *Pattern Recognition Engine* processes queries for the predefined recovery patterns. It searches the processed log file for any sequence that matches the predefined recovery patterns and returns the matching sequences with a frequency. Through *Partition Restructuring Module* component, an administrator can invoke any defined restructuring operation to make the desired changes.

6 Related Work and Conclusions

Some studies have considered the problem of exception handling and recovery from activity failures in WfMSs, such as (Ellis, Keddara & Rozenberg 1995, Casati et al. 1998, Joeris & Herzog 1998, Reichert & Dadam 1998, Klingemann 2000). Leymann (Leymann & Roller 2000) introduced the notion of *compensation sphere* which is a subset of activities that either all together have to be executed successfully or all have to be compensated. The fact that spheres do not have to form a connected graph leads to complicated semantics. The model of Hagen and Alonso (Hagen & Alonso 2000) uses a similar notion of sphere to specify atomicity and focuses on the handling of expected exceptions and the integration of exception handling in the execution environment. Grigori et al. (Grigori, Casati, Dayal & Shan 2001) focus on the analysis, prediction, and prevention of exceptions in order to reduce their occurrences. In contrast, our aim is to model the recovery from an exception at design time. In addition, an extensive amount of work on flexible recovery in the context of advanced transaction models has been done, e.g., in (Georgakopoulos, Hornick & Manola 1996, Wächter & Reuter 1992). They particularly show how some of the concepts used in transaction management can be applied to workflow environments.

The idea of applying process mining in the context of workflow management was first introduced by Agrawal et al. in (Agrawal, Gunopulos & Leymann 1998). The authors proposed an approach, based on the induction of directed graphs, for discovering a workflow schema from logs of previous unstructured executions of the workflow. In (Cook & Wolf 1998), Cook and Wolf investigated the issue of process mining in the context of software engineering processes. Herbst and Karagiannis (Herbst & Karagiannis 2000) also addressed the issue of process mining in the context of workflow management. They use ADONIS modeling language and is based on hidden Markov models where models are merged and split in order to discover the underlying process. Weijters and Aalst (Weijters & Aalst 2001) focus on workflow processes with concurrent behavior by combining techniques from machine learning with Petri nets.

In this paper, we used Self-Adaptive Recovery Net (SARN) model to partition a workflow into recovery regions in order to reduce the complexity of workflow recovery due to expected exceptions. We proposed a pattern-based approach to dynamically restructure SARN partition based on the observation of predefined recovery patterns from workflow logs. This method is particularly interesting in designing flexible and adaptable business processes. A simulator has been built to demonstrate the feasibility of SARN restructuring.

References

Agrawal, R., Gunopulos, D. & Leymann, F. (1998), Mining Process Models from Workflow Logs, *in* 'Proceedings of the 6th International Conference on Extending Database Technology (EDBT'98)', Valencia, Spain.

Benatallah, B., Chrzastowski-Wachtel, P., Hamadi, R., O'Dell, M. & Susanto, A. (2003), HiWorD: A Petri Net-based Hierarchical Workflow Designer, *in* 'Proceedings of the 3rd International Conference on Application of Concurrency to System Design (ACSD'03)', IEEE Computer Society Press, Guimaraes, Portugal, pp. 235–236.

Berry, M. & Linoff, G. (2000), *Mastering Data Mining*, Jhon Wiley & Son, Inc.

Casati, F., Ceri, S., Pernici, B. & Pozzi, G. (1998), 'Workflow Evolution', *Data and Knowledge Engineering* **24**(3), 211–238.

Chrzastowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M. & Susanto, A. (2003), A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling, *in* 'Proceedings of the International Conference on Business Process Management (BPM'03)', LNCS 2678, Springer Verlag, Eindhoven, The Netherlands, pp. 336–353.

Cook, J. & Wolf, A. (1998), 'Discovering Models of Software Processes from Event-Based Data', *ACM Transactions on Software Engineering and Methodology* **7**(3), 215–249.

Ellis, C., Keddara, K. & Rozenberg, G. (1995), Dynamic Change within Workflow Systems, *in* 'Proceedings of the Conference on Organizational Computing Systems (COOCS'95)', ACM Press, Milpitas, USA, pp. 10–21.

Georgakopoulos, D., Hornick, M. & Manola, F. (1996), 'Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation', *IEEE Transactions on Knowledge and Data Engineering* **8**(4), 630–649.

Georgakopoulos, D., Hornick, M. & Sheth, A. (1995), 'An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure', *Distributed and Parallel Databases* **3**(2).

Grigori, D., Casati, F., Dayal, U. & Shan, M.-C. (2001), Improving Business Process Quality through Exception Understanding, Prediction, and Prevention, *in* 'Proceedings of the 27th Very Large Data Base Conference (VLDB'01)', Rome, Italy.

Hagen, C. & Alonso, G. (2000), 'Exception Handling in Workflow Management Systems', *IEEE Transactions on Software Engineering (TSE)* **26**(10), 943–958.

Hamadi, R. & Benatallah, B. (2004), Recovery Nets: Towards Self-Adaptive Workfow Systems, *in* 'Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE'04)', LNCS 3306, Springer Verlag, Brisbane, Australia, pp. 439–453.

Herbst, J. & Karagiannis, D. (2000), 'Integrating Machine Learning and Workflow Management to Support Acquisition and Adaption of Workflow Models', *International Journal of intelligent Systems in Accounting, Finance and Management* **9**, 67–92.

Joeris, G. & Herzog, O. (1998), Managing Evolving Workflow Specifications, *in* 'Proceedings of the 3rd Conference on Cooperative Information Systems (CoopIS'98)', New York, USA.

Klingemann, J. (2000), Controlled Flexibility in Workflow Management, *in* 'Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE'00)', Stockholm, Sweden.

Leymann, F. & Roller, D. (2000), *Production Workflow — Concepts and Techniques*, Prentice Hall.

Murata, T. (1989), Petri Nets: Properties, Analysis and Applications, *in* 'Proceedings of the IEEE', Vol. 77(4), pp. 541–580.

Peterson, J. (1981), *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs.

Reichert, M. & Dadam, P. (1998), 'ADEPT flex: Supporting Dynamic Changes of Workflows without Losing Control', *Journal of Intelligent Information Systems* **10**(2), 93–129.

Wächter, H. & Reuter, A. (1992), The ConTract Model, *in* A. Elmagarmid, ed., 'Database Transaction Models for Advanced Applications', Morgan Kaufmann, pp. 219–264.

Weijters, A. & Aalst, W. v. d. (2001), Process Mining. Discovering Workflow Models from Event-Based Data, *in* B. Krse, M. De Rijke, G. Schreiber & M. v. Someren, eds, 'Proceedings of the 13th Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'01)', Maastricht, Netherlands, pp. 283–290.

WfMC (1999), *Workflow Management Coalition, Terminology and Glossary*, Document Number WFMC-TC-1011. http://www.wfmc.org/standards/docs.htm/.