Policy-Based Exception Handling in Business Processes

Rachid Hamadi and Boualem Benatallah

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

Technical Report
UNSW-CSE-TR0428
August 2004

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

Abstract

A workflow management system (WfMS) provides a central control point for defining business processes and orchestrating their execution. A major limitation of current WfMSs is their lack of support for dynamic workflow adaptations. This functionality is an important requirement in order to provide sufficient flexibility to cope with expected but unusual situations and failures. In this report, we propose Self-Adaptive Recovery Net (SARN), an extended Petri net model for specifying exceptional behavior in workflow systems at design time. SARN can adapt the structure of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple and easy. The proposed framework also caters for high-level recovery policies that are incorporated either with a single task or a set of tasks, called a recovery region. These recovery policies are generic constructs that model exceptions at design time together with a set of primitive operations that can be used at run time to handle the occurrence of exceptions.

1 Introduction

A workflow management system (WfMS) provides a central control point for defining business processes and orchestrating their execution [GHS95, CCPP98]. A major limitation of current WfMSs is their lack of support for dynamic workflow adaptations. This functionality is an important requirement in order to provide sufficient flexibility to cope with expected but unusual situations and failures.

In a workflow process, each task can fail, e.g., its execution produces unexpected results. Therefore, exception handling must be part of the workflow process. If we want to model a workflow that takes into account and reacts to failures adequately, then the exception handling part ends up dominating the normal behaviour part [HA00]. Existing workflow modeling languages such as traditional state machines, statecharts, and Petri nets [Pet81] are not suitable when the exceptional behaviour exceeds the normal behaviour since they do not, as such, have the ability to reduce modeling size, improve design flexibility, and support exception handling at design time.

There are two main approaches that deal with exceptions in existing WfMSs: (i) the *Ad-Hoc* approach and (ii) the *Run Time* approach. The former integrates all expected exceptions within the normal behavior of the business process. This makes the design of the workflow complicated and inconvenient. The latter deals with exceptions at run time, meaning that there must be a workflow expert who decides which changes have to be made to the workflow process in order to handle exceptions (see, e.g., [RD98]).

One solution, which is discussed in this report, is to use high-level recovery policies that are incorporated either with a *single* task or a set of tasks that we will call hereafter a *recovery region*. These recovery policies are *generic constructs* that model exceptions at design time together with a set of primitive operations that can be used at run time to handle the occurrence of exceptions. Note that our proposed approach concentrates on handling exceptions at the instance level and not on modifying the workflow schema such as in [CCPP98].

We identified a set of recovery policies that are useful and commonly needed in many practical situations. The problem, often overlooked, is that adding recovery policies to workflow modeling languages is a delicate issue. In fact, while in general new policies may provide the support and flexibility described above, they also make the workflow model more complex. Complexity severely compromises the usability (and therefore the adoption) of such models. Therefore, the hard part lies in striking a balance between expressive power and simplicity. As a consequence, the goal that guided our work is exactly that of striking this balance and "right-sizing" the model, while providing room for it to evolve as the need arises. This required an analysis of existing workflow applications and of their exceptional behaviors so that we could:

- determine a set of recovery policies that could adequately model most of them and
- avoid the artificial introduction of complex recovery policies that we could have thought useful but that are rarely used in practice.

In this report, we propose Self-Adaptive Recovery Net (SARN), an extended Petri net model for specifying exceptional behavior in workflow systems at design time. The proposed framework also caters for high-level recovery policies that are incorporated either with a single task or a recovery region, i.e., a set of tasks. Our contribution is twofold:

- *Self-Adaptive Recovery Net (SARN).* An extended Petri net model for specifying exceptional behavior in workflow systems at design time. This model can adapt the structure of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple and easy.

- *High-level recovery policies.* These recovery policies are generic constructs that model exceptions at design time. They are incorporated with either a single task or a set of tasks called a recovery region. Note that this list of recovery policies is not exhaustive. Indeed, new (customized) recovery policies can be added.

Handling exceptions introduces two major aspects. First, the system should provide support (i.e., a set of recovery policies) to enable the flexibility to deal with expected exceptions. Second, expected exceptions should only be allowed in a valid way. The system must ensure the correctness of the modified SARN w.r.t. consistency constraints (such as reachability and absence of deadlock), so that constraints that were valid before the dynamic change of SARN are also valid after the modification.

The rest of the report is organized as follows. Section 2 introduces the proposed model SARN along with a motivating example. Section 3 presents the task-based recovery policies. Their extension to a set of tasks or region is given in Section 4. Finally, Section 5 reviews some related work and concludes the report.

2 Self-Adaptive Recovery Net

In this section, we introduce the model proposed, i.e., Self-Adaptive Recovery Net (SARN). SARN is an extended Petri net model. We will first informally give the definition of Petri nets.

Petri nets [Pet62, Pet81] are a well-founded process modeling technique that have formal semantics. They have been used to model and analyze several types of processes including protocols, manufacturing systems, and business processes. A Petri net is a directed, connected, and bipartite graph in which each node is either a *place* or a *transition*. Tokens occupy places. When there is at least one token in every place connected to a transition, we say that the transition is *enabled*. Any enabled transition may *fire* removing one token from every input place, and depositing one token in each output place. For a more elaborate introduction to Petri nets, the reader is referred to [Mur89, Rei85, Pet81].

SARN extends Petri nets 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 a failure event occurs. There are also two types of tokens: *standard* tokens for the firing of standard transitions and *recovery* tokens associated with recovery policies. There is one recovery transition per type of task failure, that is, when a new recovery policy is designed, a new recovery transition is added. When a failure (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.

2.1 A Motivating Example

A simplified *Travel Planning* workflow specified as an SARN model is depicted in Figure 1.

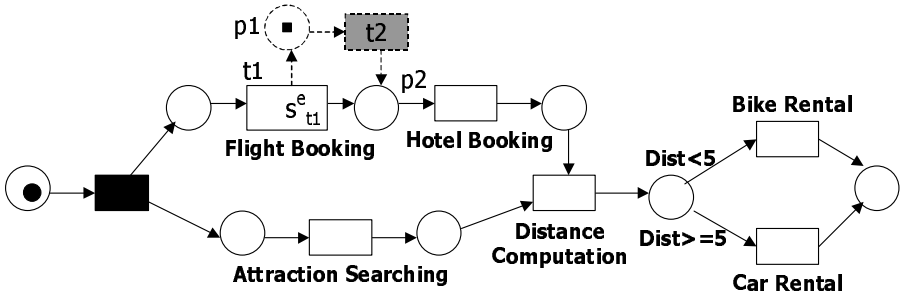


Figure 1: *Travel Planning* Workflow as a SARN

In this workflow process, a sequence of a *Flight Booking* task followed by a *Hotel Booking* task is performed in parallel with an *Attraction Searching* task. After these booking and searching tasks are completed, the distance from the attraction location to the accommodation is computed, and either a *Car Rental* task or a *Bike Rental* task is invoked. The symbol S^e_{t1} within the *Flight Booking* task means that a *Skip*[1] recovery policy is associated with it.

The part drawn in dotted lines is created after the sequence of primitive operations associated with the *Skip* recovery transition has been executed. When a *Skip* failure event `e` (e.g., `e`="no response after one day") occurs during the execution of the task *Flight Booking*, the *Skip* recovery transition appears and the operations associated with it are executed.

Note that, in this report, exceptions constitute events which may occur during workflow execution and which require deviations from the normal business process. These events can be *generic* such as the unavailability of resources, a time-out, and incorrect input types, or *application specific* (i.e., *dependent*) such as the unavailability of seats for the *Flight Booking* task in the *Travel Planning* workflow. In this latter case, events must be defined by the workflow designer.

2.2 Definition of SARN

In the following, we will give the formal definition of SARN.

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,*

[1] Details about the *Skip* recovery policy and other task-based recovery policies will be discussed in Section 3.

- *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 a failure event occurs. There is one recovery transition per type of task failure such as Skip and Compensate,*
- *$F \subseteq (P \times (T \cup Tr)) \cup ((T \cup Tr) \times P)$ 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).* □

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 Table 1). Several of these basic operations are combined in a specific order to handle different task failure events. The approach adopted is to use one recovery transition to represent one type of task failure. Thus a recovery transition represents a combination of several basic operations.

Table 1: Primitive Operations

Basic Operation	Effect
`CreateArc(x,y)`	An arc linking the place (or transition) `x` to the transition (or place) `y` is created
`DeleteArc(x,y)`	An arc linking the place (or transition) `x` to the transition (or place) `y` is deleted
`DisableArc(x,y)`	An arc linking the place (or transition) `x` to the transition (or place) `y` is disabled
`ResumeArc(x,y)`	A disabled arc linking the place (or transition) `x` to the transition (or place) `y` is resumed, i.e., will participate in firing transitions
`CreatePlace(p)`	A place `p` is created
`DeletePlace(p)`	A place `p` is deleted
`CreateTransition(t)`	A transition `t` is created
`SilentTransition(t)`	An existing transition `t` is replaced by a silent transition
`ReplaceTransition(t,t')`	An existing transition `t` is replaced by another transition `t'`
`DeleteTransition(t)`	A transition `t` is deleted
`DisableTransition(t)`	A transition `t` will not be able to fire
`ResumeTransition(t)`	A disabled transition `t` is resumed, i.e,
`AddToken(p)`	A standard token is added to a place `p`
`RemoveToken(p)`	A standard token is removed from a place `p`
`AddRecoveryToken(p)`	A recovery token is added to a place `p`
`RemoveRecoveryToken(p)`	A recovery token is removed from a place `p`

In the example depicted in Figure 1, the *Skip* recovery policy S^e_{t1} is associated with eight basic operations: (1) `DisableTransition(t1)`, (2) `CreatePlace(p1)`, (3) `AddRecoveryToken(p1)`, (4) `CreateArc(t1,p1)`, (5) `CreateTransition(t2)`, (6) `SilentTransition(t2)`, (7) `CreateArc(p1,t2)`, and (8) `CreateArc(t2,p2)`.

It should be noted that this sequence of operations must be executed as an atomic transaction.

In SARN, when a task failure occurs, a recovery token is injected into a place and the set of primitive operations associated with the recovery policy are triggered.

2.3 Enabling and Firing Rules in SARN

In ordinary Petri nets, a transition is enabled if all its input places contain at least one token. An enabled transition can be fired and one token is removed from each of its input places and one token is deposited in each of its output places. The tokens are deposited in the output places of the transition once the corresponding task finishes its execution, that is, the tokens remain hidden when the task is still active. Upon the completion of the task, the tokens appear in the outgoing places. If no task failure events occur, SARN will obey this enabling rule.

Besides supporting the conventional rules of Petri nets, some new rules are needed in SARN as a result of the new mechanisms added (recovery transitions and tokens). Here is what will happen when a task failure event occurs:

1. The recovery transition corresponding to this failure event will be created and a recovery token is injected in the newly created input place of the recovery transition.

2. Once a recovery token is injected, the execution of the faulty task will be paused and the system will not be able to take any further task failure event.

3. Once a recovery transition is created, the basic operations associated with it will be executed in the order specified. The net structure will be modified to handle the corresponding failure. Note that this sequence of basic operations does not introduce inconsistencies such as deadlocks.

4. When all operations associated with the recovery transition are executed, all the modifications made to the net structure in order to handle the failure will be removed. The net structure will be restored to the configuration before the occurrence of the failure event.

5. The recovery tokens generated by the execution of the operations will become standard tokens and the normal execution is resumed. The enabled transitions will then fire according to the standard Petri net firing rules.

2.4 Valid SARN Models

A key issue in supporting dynamic SARN structural changes is that the system should guarantee that the modified SARN is valid w.r.t. consistency constraints such as reachability and liveness. For instance, the system must not allow to skip a running task to a non-subsequent task (in the control flow). Consistency rules must be established in order

to control any invalid use of the recovery policies or restrict their use (to a specific part(s) of the SARN model).

SARN must meet certain *constraints* in order to ensure the correct execution of the underlying workflow at run time. Each transition t must be *reachable* from the initial marking of the SARN net. That is, there is a valid sequence of transitions leading from the initial marking to the firing of t. Furthermore, we require that from every reachable marking of the SARN net, a final marking (where there is only one standard token in the output place o of the SARN net) can be reached, i.e., there is a valid sequence of transitions leading from the current marking of the net to its final marking.

To verify the correctness of our SARN model, we utilize some key definitions for Petri net behavior properties adapted from [Mur89].

Definition 2.2 (Reachability)
In a SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ *with initial marking* $M_0 = i$*, a marking* M_n *is said to be reachable from* M_0 *if there exists a sequence of firings that transforms* M_0 *to* M_n*. A firing or occurrence sequence is denoted by* $\sigma = M_0 t_1 M_1 t_2 M_2 \dots t_n M_n$ *or simply* $\sigma = t_1 t_2 \dots t_n$*. In this case,* M_n *is reachable from* M_0 *by* σ *and we write* $M_0[\sigma\rangle M_n$*.* □

Definition 2.3 (Liveness)
A SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ *is said to be live, if for any marking* M_n *that is reachable from* $M_0 = i$*, it is possible to ultimately fire any transition of the net by progressing some further firing sequence.* □

Definition 2.4 (Boundedness)
A SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ *is said to be* k*-bounded or simply bounded, if the number of tokens in each place does not exceed a finite number* k *for any marking reachable from* $M_0 = i$*. A SARN net* $RN = (P, T, Tr, F, i, o, \ell, M)$ *is said to be safe if it is* 1*-bounded.* □

The *boundedness* property is useful in checking, for instance, that a place stands for a status or a condition if the number of tokens it contains is either zero or one, or if a place stands for a number of resources then boundedness can be used to check if the number of resources overflows.

A SARN model that satisfies the above properties is said to be *valid*.

2.5 Task States

Each task in a workflow process contains a task state variable that is associated with a task state type which determines the possible task states [GSCB99]. A transition from one task state to another constitutes a primitive task event.

Figure 2 shows the generic task states. It is consistent with the proposed standard of the Workflow Management Coalition [WfM99]. At any given time, a task can be in one of the following states: `NotReady`, `Ready`, `Running`, `Completed`, `Aborted`, or `Frozen`.

The task state `NotReady` corresponds to not yet enabled task. When a task becomes enabled, i.e., all its incoming places contains at least one token, the task state changes into `Ready`. A firing of a task causes a transition to the `Running` state. Depending

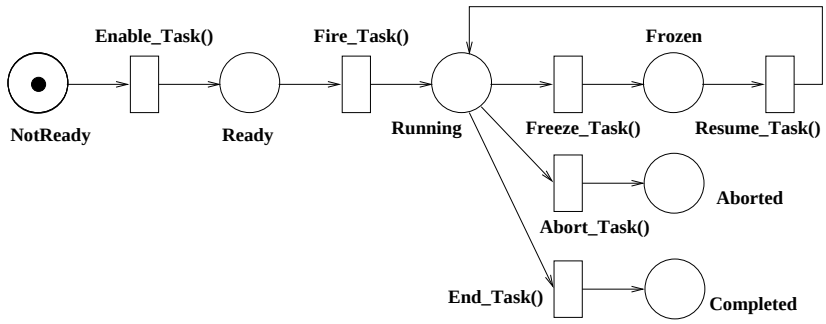


Figure 2: Generic Task States

upon whether an activity ends normally or is forced to abort, the end state of a task is either `Completed` or `Aborted`. The `Frozen` state indicates that the execution of the task is temporarily suspended. No operations may be performed on a frozen task until its execution is resumed.

3 Task-Based Recovery Policies

In what follows, we will discuss the identified task-based recovery policies. We will first informally describe the recovery policy, then give a formal definition, and finally, give an example to illustrate how the workflow system modelled as a SARN behaves at run time. We will distinguish between nine recovery policies, namely, *Skip*, *SkipTo*, *Compensate*, *CompensateAfter*, *Redo*, *RedoAfter*, *AlternativeTask*, *AlternativeProvider*, and *Timeout*. Note that this list is not exhaustive. Indeed, new (customized) task-based recovery policies can be added.

3.1 Skip

The *Skip* recovery policy will, once the corresponding failure event occurs during the execution of the corresponding task `T`: (i) disable the execution of the task `T` and (ii) skip to the immediate next task(s) in the control flow. This recovery policy applies to running tasks only.

Formally, in the context of SARN, a `Skip(Event e, Transition T)` recovery policy, when executing a task `T` and the corresponding failure event `e` occurs, means (see Figure 3):

Precondition: $state(\texttt{T}) = \texttt{Running}$.

Effect:

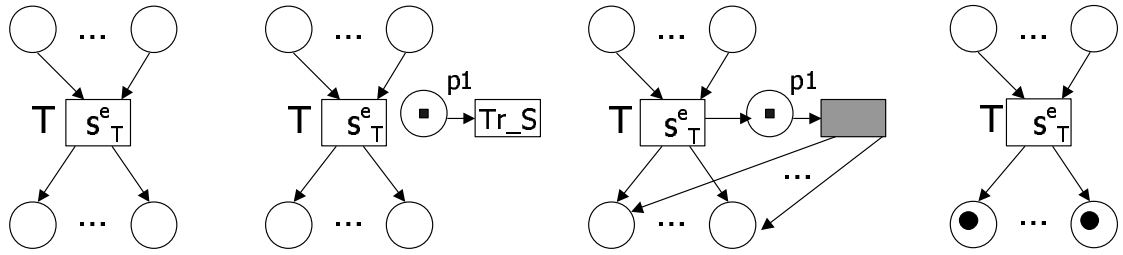
1. `DisableTransition(T)`, i.e., disable the transition of the faulty task,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_S)`: create a *Skip* recovery transition,
4. `CreateArc(p1,Tr_S)`: `p1` is the input place of the *Skip* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *Skip* recovery transition (see Figure 3(b)),

6. execute the basic operations associated with the *Skip* recovery transition to modify the net structure in order to handle the exception (see Figure 3(c)),

7. execute the added exceptional part of the SARN net,

8. once the exceptional part finishes its execution, i.e., there is no *recovery* token within the added structure, the modifications made for the task failure event are removed, and

9. resume the normal execution by transforming the recovery tokens on the output places of the skipped task into standard tokens (see Figure 3(d)).



(a) Task T running (b) A *Skip* failure event occurs (c) Handling *Skip* failure event (d) Resume normal execution

Figure 3: *Skip* Recovery Policy

The operations associated with a *Skip* recovery transition, in order to complete step (7) above, are as follows (see Figure 3(c)):

(a) `CreateArc(T,p1)`: add an incoming arc from the skipped task to the input place of the recovery transition,

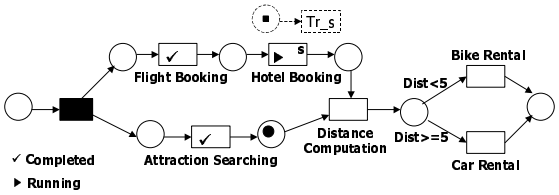
(b) `SilentTransition(Tr_S)`: replace the *Skip* recovery transition with a silent transition (i.e., a transition with no associated task or an empty task), and

(c) $\forall$ `p` $\in$ `T`$^\bullet$ `CreateArc(Tr_S,p)`: add an outgoing arc from the silent transition to each output place of the skipped task `T`.

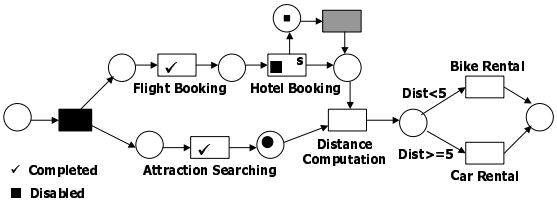
For instance, in the *Travel Planning* workflow example (see Figure 1), we associate with the task *Hotel Booking* a *Skip* recovery policy at design time. Upon the occurrence of a skip failure event (for instance, "no response after one day") while executing the *Hotel Booking* task, the resulting SARN net will look like Figure 4(a). After executing the set of basic operations corresponding to the *Skip* recovery policy, the SARN net will become like Figure 4(b). Once the skip failure is handled, the SARN net will look like Figure 4(c).

3.2 SkipTo

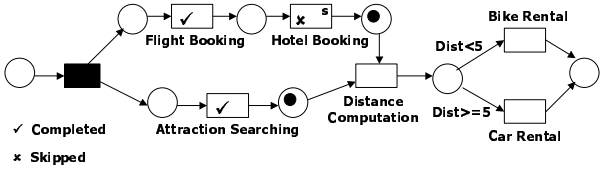
The *Skip* recovery policy defined previously is generic in the sense that there is no need to ask the designer at which step of the execution she wants to resume the execution from. Indeed, when skipping the running task, the immediate next task(s) with respect to the



(a) Occurrence of a skip failure event "no response after one day" while executing *Hotel Booking*



(b) The SARN net after executing the *Skip* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 4: *Skip* Recovery Policy Example

control flow will be executed. An interesting variant of the *Skip* recovery policy could be to skip to certain task(s) and not necessarily to the immediate next task(s). We will call this derived recovery policy *SkipTo*. It should be noted that the tasks of the set of tasks $\mathcal{T}$ to skip to must be pairwise independent and each task of $\mathcal{T}$ must be a subsequent task of the skipped task `T1`.

Formally, a `SkipTo(Event e, Task T1, TaskSet` $\mathcal{T}$`)` recovery policy, when its corresponding failure event `e` occurs when executing a task `T1`, is defined as follows (see Figure 5):

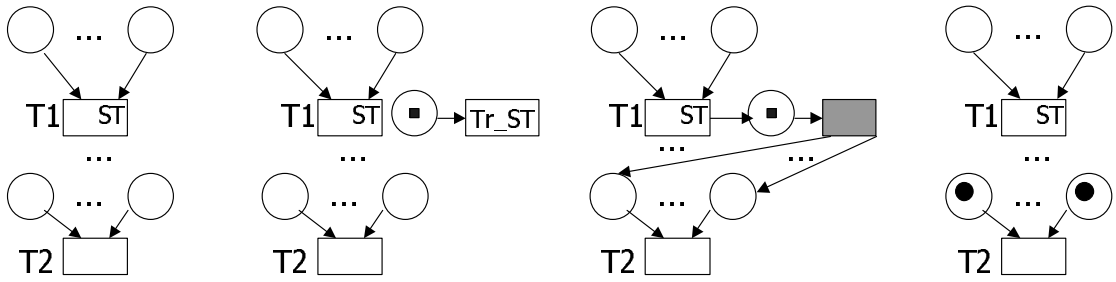
Precondition:

- $state($`T1`$) =$ `Running`,
- $\forall$ `T2`, `T2`$' \in \mathcal{T}$ $(\texttt{T2}, \texttt{T2}') \notin F^+ \wedge (\texttt{T2}', \texttt{T2}) \notin F^+$, that is, the tasks of $\mathcal{T}$ are pairwise independent with respect to the flow relation F (see Definition 2.1). F^+ denotes the transitive closure of F, and
- $\forall$ `T2` $\in \mathcal{T}$ $(\texttt{T1}, \texttt{T2}) \in F^+$, i.e., there is a path, with respect to the flow relation F, from the skipped task `T1` to the task(s) of $\mathcal{T}$ to skip to.

Effect:

1. `DisableTransition(T1)`: disable the transition of the faulty task,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_ST)`: create a *SkipTo* recovery transition,

4. `CreateArc(p1,Tr_ST)`: `p1` is the input place of the *SkipTo* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *SkipTo* recovery transition (see Figure 5(b)),
6. execute the elementary operations associated with the *SkipTo* recovery transition (see Figure 5(c)),
7. execute the added exceptional part of the SARN net to handle the exception,
8. remove the modifications made for the skip to failure event, and
9. resume the regular execution by transforming the recovery tokens on the input places of the task(s) to skip to into standard tokens (see Figure 5(d)).



(a) Task T1 running Task T2 to skip to

(b) A *SkipTo* failure event occurs

(c) Handling *SkipTo* failure event

(d) Resume normal execution

Figure 5: *SkipTo* Recovery Policy

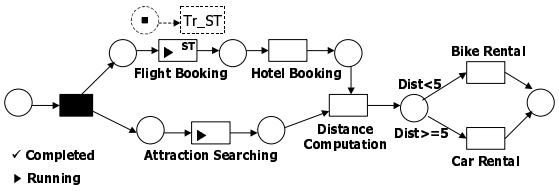
The basic operations associated with a *SkipTo* recovery transition are as follows (see Figure 5(c)):

(a) `CreateArc(T1,p1)`: add an incoming arc from the skipped task to the input place of the recovery transition,

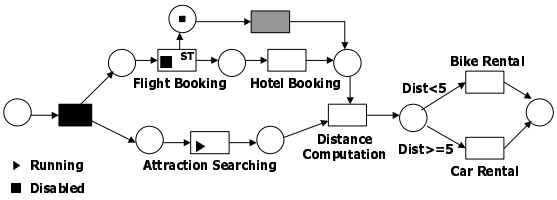
(b) `SilentTransition(Tr_ST)`: replace the *SkipTo* recovery transition with a silent transition, and

(c) $\forall$ `T2` $\in \mathcal{T}$ $\forall$ `p` $\in$ $^\bullet$`T2` `CreateArc(Tr_ST,p)`: add an outgoing arc from the silent transition to each input place of the task(s) to skip to.

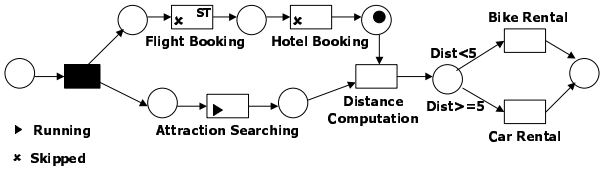
Figure 6 gives an example of the *SkipTo* recovery policy where the running *Flight Booking* task is skipped to the *Distance Computation* task. We associate with the task *Flight Booking* a *SkipTo* recovery policy at design time. When a skip to failure event, e.g., "unavailability of seats", occurs while executing the *Flight Booking* task, the resulting SARN net will look like Figure 6(a). After executing the set of basic operations associated with the *SkipTo* recovery policy, the SARN net will become like Figure 6(b). Finally, once the *SkipTo* exception handled, the *Travel Planner* workflow will look like Figure 6(c).



(a) Occurrence of a skip to failure event "unavailability of seats" while executing *Flight Booking*



(b) The SARN net after executing the *SkipTo* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 6: *SkipTo* Recovery Policy Example

3.3 Compensate

The *Compensate* recovery policy removes all the effects of a completed task. The task must be compensatable, i.e., there is a `compensate-T` task that removes the effect of the task `T` (see [BCTH03] for more details about compensatable tasks). Note that the event of compensating a task can occur any time after the completion of the task and before the workflow execution terminates. Furthermore, we assume that there is no data flow dependencies between the task to be compensated and the subsequent completed task(s).

Formally, in the context of our model, a `Compensate(Event e, Task T)` recovery policy of a task `T` when its corresponding failure event `e` occurs means:

Precondition:

- $state(\texttt{T}) = \texttt{Completed}$ and
- `T` is *compensatable*, i.e., there is a `compensate-T` task of `T`.

Effect:

1. $\forall\ \texttt{t} \in T \mid (\texttt{T},\texttt{t}) \in F^{+} \wedge state(\texttt{t}) = \texttt{Running}$ *do* `DisableTransition(t)`, so that $state(\texttt{t}) = \texttt{Frozen}$, hence all running subsequent task(s) of the task to be compensated are disabled (recall that T is the set of transitions, i.e., tasks of the workflow),
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_C)`: create a *Compensate* recovery transition,
4. `CreateArc(p1,Tr_C)`: `p1` is the input place of the *Compensate* recovery transition,

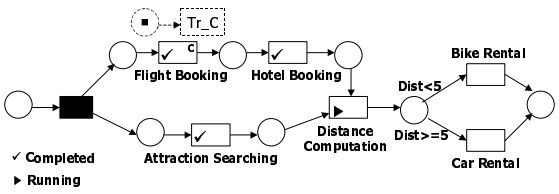
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *Compensate* recovery transition,
6. execute the primitive operations associated with the *Compensate* recovery transition,
7. execute the exceptional part of the SARN net,
8. remove the modifications made for the task failure event, and
9. resume the execution of the workflow.

The operations associated with a *Compensate* recovery transition are as follows:

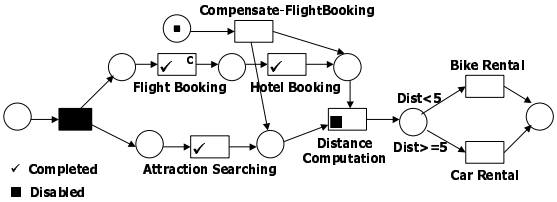
(a) `ReplaceTransition(Tr_C,compensate-T)`: associate to the *Compensate* recovery transition the task `compensate-T` that removes the effects of the task `T` and

(b) $\forall\, \mathtt{t} \in T \mid state(\mathtt{t}) = \mathtt{Frozen}\ \forall\, \mathtt{p} \in {}^{\bullet}\mathtt{t}$ *do* $\mathtt{CreateArc}(\mathtt{compensate} - \mathtt{T}, \mathtt{p})$: add an outgoing arc from the `compensate-T` transition to each input place of the suspended running task(s).

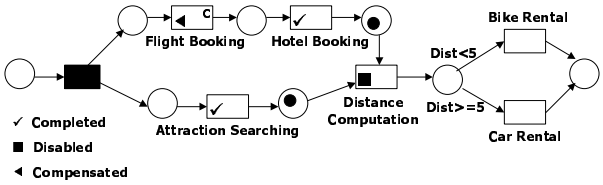
Figure 7 gives an example of the *Compensate* recovery policy where the *Flight Booking* task was compensated while the system was executing the *Distance Computation* task. At design time, we associate with the task *Flight Booking* a *Compensate* recovery policy. When a compensate failure event, for instance, "cancel flight", occurs while executing the task *Distance Computation*, the resulting SARN net will look like Figure 7(a). After executing the set of basic operations associated with the *Compensate* recovery policy, the SARN net will appear like Figure 7(b). Finally, once the *Compensate* exception is handled, the *Travel Planner* workflow will look like Figure 7(c).



(a) Occurrence of a compensate failure event "cancel flight" while executing *Distance Computation*



(b) The SARN net after executing the *Compensate* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 7: *Compensate* Recovery Policy Example

3.4 CompensateAfter

The *Compensate* recovery policy removes the effects of a completed task any time after the completion of the task and before the workflow execution terminates. An interesting case that will not have effects on the subsequent dependant tasks is when compensating a task just after finishing its execution and before initiating any subsequent dependant task. We will call this particular *Compensate* recovery policy *CompensateAfter*.

Formally, a `CompensateAfter(Event e, Task T)` recovery policy of a task `T` when its corresponding failure event `e` occurs means:

Precondition:

- $state(\texttt{T}) = \texttt{Completed}$ and
- `T` is *compensatable*, i.e., there is a `compensate-T` task of `T`.

Effect:

1. `CreatePlace(p1)`: create a new place `p1`,
2. `CreateTransition(Tr_CA)`: create a *CompensateAfter* recovery transition,
3. `CreateArc(p1,Tr_CA)`: `p1` is the input place of the *CompensateAfter* recovery transition,
4. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *CompensateAfter* recovery transition,
5. execute the basic operations associated with the *CompensateAfter* recovery transition,
6. execute the added exceptional part of the SARN net to handle the exception,
7. remove the modifications made for the task failure event, and
8. resume the execution of the workflow.

The operations associated with a *CompensateAfter* recovery transition are as follows:

(a) `ReplaceTransition(Tr_CA,compensate-T)`: associate with the *CompensateAfter* recovery transition the task `compensate-T` that removes the effects of the task `T`,

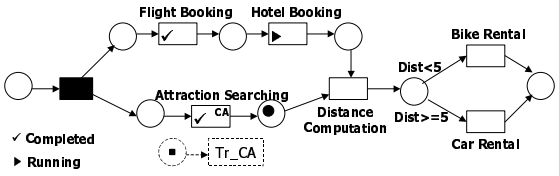
(b) $\forall$ `p` $\in$ $\texttt{T}^\bullet$ $\texttt{CreateArc}(\texttt{compensate} - \texttt{T}, \texttt{p})$: add an outgoing arc from the `compensate-T` transition to each output place of the compensated task,

(c) disable all outgoing arcs of the compensated task, and

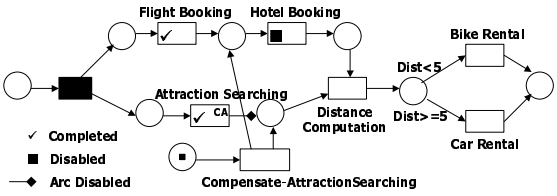
(d) remove one (standard) token from each output place of the compensated task.

In Figure 8, an example of the *CompensateAfter* recovery policy is given where the *Attraction Searching* task was compensated just after it finishes its execution and while *Hotel Booking* was running. The task *Attraction Searching* was associated with a *CompensateAfter* recovery policy at built time. When a compensate after failure event occurs just

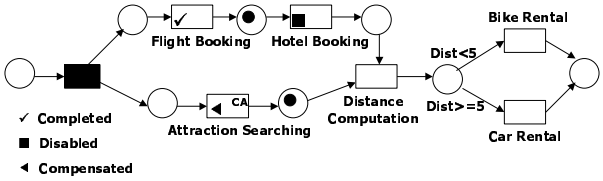
after completing the execution of the task *Attraction Searching*, the resulting SARN net will look like Figure 8(a). Once the set of basic operations associated with the *CompensateAfter* recovery policy are executed, the SARN net will look like Figure 8(b). Finally, after handling the *CompensateAfter* exception, the *Travel Planner* workflow will appear like Figure 8(c).



(a) Occurrence of a compensate after failure event just after completing *Attraction Searching*



(b) The SARN net after executing the *CompensateAfter* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 8: *CompensateAfter* Recovery Policy Example

3.5 Redo

The *Redo* recovery policy repeats (once) a finished task `T` because, for instance, the results obtained are not delivered (or produced) as expected. Note that, like the *Compensate* recovery policy, the event of redoing a task can occur any time after the completion of the task and before the workflow execution terminates. Furthermore, we assume that there is no data flow dependencies between the task to be repeated and the subsequent completed task(s).

Formally, a `Redo(Event e, Task T)` recovery policy of the task `T` when its corresponding failure event `e` occurs means:

Precondition: $state(\texttt{T}) = \texttt{Completed}$.

Effect:

1. $\forall\ \texttt{t} \in T\ \mid\ (\texttt{T}, \texttt{t}) \in F^{+}\ \wedge\ state(\texttt{t}) = \texttt{Running}$ *do* $\texttt{DisableTransition}(\texttt{t})$, so that $state(\texttt{t}) = \texttt{Frozen}$, hence all running subsequent task(s) of the task to be repeated are disabled,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_R)`: create a *Redo* recovery transition,

4. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *Redo* recovery transition,
5. execute the elementary operations associated with the *Redo* recovery transition,
6. execute the exceptional part of the SARN net,
7. remove the modifications made for the task failure event, and
8. resume the execution of the workflow.

The operations associated with a *Redo* recovery transition (to complete step (5) above) are as follows:

(a) disable all incoming arcs of the task to be repeated,

(b) disable all outgoing arcs from each output place of the task to be repeated,

(c) `CreateArc(p1,T)`: add an outgoing arc from the created place `p1` (that contains a recovery token) to the task `T` to be repeated,

(d) `SilentTransition(Tr_R)`: replace the *Redo* recovery transition with an empty task,

(e) add an outgoing arc from each output place of the task to be repeated to the empty *Redo* recovery transition, and

(f) add an outgoing arc from the empty *Redo* recovery transition to each input place of the disabled transitions.

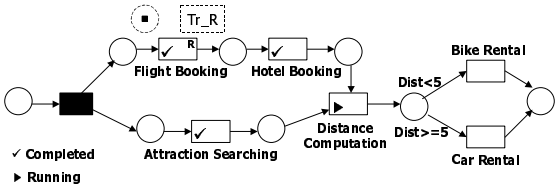
Figure 9 gives an example of the *Redo* recovery policy where the *Flight Booking* task was repeated while the system was executing the *Distance Computation* task. At design time, we associate with the task *Flight Booking* a *Redo* recovery policy. When a redo failure event, e.g., "change flight date", occurs while executing the task *Distance Computation*, the resulting SARN net will look like Figure 9(a). After executing the set of primitive operations associated with the *Redo* recovery policy, the SARN net will become like Figure 9(b). Finally, once the redo exception is handled, the *Travel Planner* workflow will look like Figure 9(c).

3.6 RedoAfter

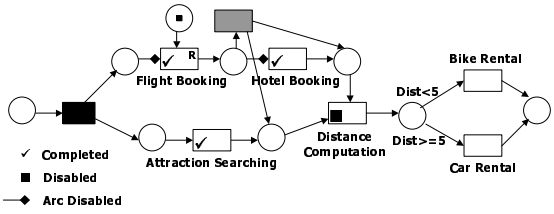
The *Redo* recovery policy repeats the execution of a completed task any time after the completion of the task and before the workflow execution terminates. An interesting case that will not have effects on subsequent dependant tasks is when redoing a task just after finishing its execution and before initiating any subsequent dependant task. We will call this particular *Redo* recovery policy *RedoAfter*.

Formally, a `RedoAfter(Event e, Task T)` recovery policy of a task `T` when its corresponding failure event `e` occurs means:

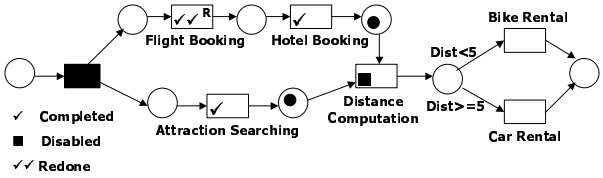
Precondition: $state(\texttt{T}) = \texttt{Completed}$.



(a) Occurrence of a redo failure event "change flight date" while executing *Distance Computation*



(b) The SARN net after executing the *Redo* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 9: *Redo* Recovery Policy Example

Effect:

1. `CreatePlace(p1)`: create a new place `p1`,
2. `CreateTransition(Tr_RA)`: create a *RedoAfter* recovery transition,
3. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *RedoAfter* recovery transition,
4. execute the primitive operations associated with the *RedoAfter* recovery transition,
5. execute the added exceptional part of the SARN net to handle the exception,
6. remove the modifications made for the task failure event, and
7. resume the execution of the workflow.

The operations associated with a *RedoAfter* recovery transition (to complete step (4) above) are as follows:

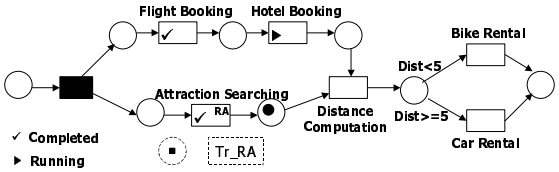
(a) disable all incoming arcs of the task to be repeated,

(b) `CreateArc(p1,T)`: add an outgoing arc from the created place `p1` to the task `T` to be repeated,

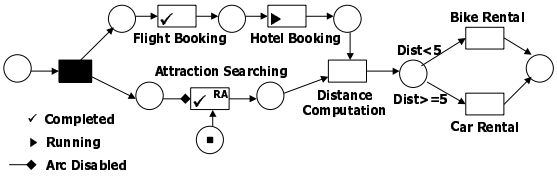
(c) `DeleteTransition(Tr_RA)`: delete the *RedoAfter* recovery transition, and

(d) remove one (standard) token from each output place of the task to be repeated.

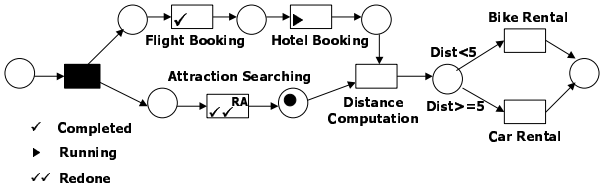
In Figure 10, an example of the *RedoAfter* recovery policy is given where the *Attraction Searching* task was repeated just after it finishes its execution and while *Hotel Booking* was running. The task *Attraction Searching* was associated with a *RedoAfter* recovery policy at built time. When a redo after failure event "modify attraction time", for instance, occurs just after completing the execution of the task *Attraction Searching*, the resulting SARN net will look like Figure 10(a). Once the set of basic operations associated with the *RedoAfter* recovery policy are executed, the SARN net will look like Figure 10(b). Finally, after handling the *RedoAfter* exception, the *Travel Planner* workflow will look like Figure 10(c).



(a) Occurrence of a redo after failure event "modify attraction time" just after completing *Attraction Searching*



(b) The SARN net after executing the *RedoAfter* recovery policy operations



(c) The SARN net just before resuming the normal execution

Figure 10: *RedoAfter* Recovery Policy Example

3.7 AlternativeTask

The *AlternativeTask* recovery policy allows another task `T'` to be executed in place of a running task `T` in case the later fails.

Formally, in the context of SARN, an `AlternativeTask(Event e, Task T, Task T')` recovery policy of a task `T` by another task `T'` when its corresponding failure event `e` occurs means (see Figure 11):

Precondition: $state(\texttt{T}) = \texttt{Running}$.

Effect:

1. `DisableTransition(T)`: disable the running task `T`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_AT)`: create an *AlternativeTask* recovery transition,

4. `CreateArc(p1,Tr_AT)`: `p1` is the input place of the *AlternativeTask* recovery transition,

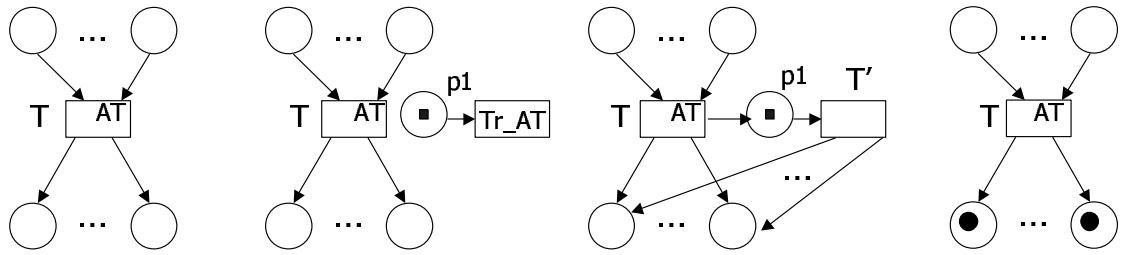
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *AlternativeTask* recovery transition (see Figure 11(b)),

6. execute the basic operations associated with the *AlternativeTask* recovery transition to modify the SARN structure(see Figure 11(c)),

7. run the added exceptional part of the SARN net,

8. remove the modifications made for the net once the exceptional part finishes its execution, and

9. resume the normal execution by transforming the recovery tokens on the output places of the substituted task into standard tokens (see Figure 11(d)).



(a) Task T running (b) An *AlternativeTask* failure event occurs (c) Handling *AlternativeTask* failure event (d) Resume normal execution

Figure 11: *AlternativeTask* Recovery Policy

The basic operations associated with an *AlternativeTask* recovery transition are as follows (see Figure 11(c)):

(a) `CreateArc(T,p1)`: add an incoming arc from the replaced task `T` to the input place `p1` of the recovery transition `Tr_AT`,

(b) `ReplaceTransition(Tr_AT,T')`: replace the *AlternativeTask* recovery transition with the alternative task `T'`, and

(c) $\forall$ `p` $\in$ `T`$^\bullet$ `CreateArc(`T'`,p)`: add an outgoing arc from `T'` to each output place of the substituted task.

3.8 AlternativeProvider

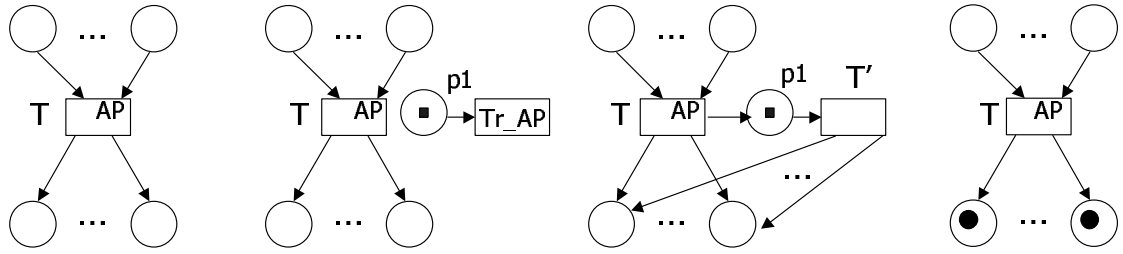
The *AlternativeProvider* recovery policy allows an alternative execution of a task `T` by another provider `P` in case the current provider fails to execute the task `T`. This is especially interesting in the context of Web services since each transition represents a service community from which a service is chosen at run time to execute the corresponding task.

Formally, in the context of SARN, an `AlternativeProvider(Event e, Task T, Provider P)` recovery policy of a task `T` by another provider `P` when its corresponding failure event `e` occurs means (see Figure 12):

Precondition: $state(\texttt{T}) = \texttt{Running}$.

Effect:

1. `DisableTransition(T)`: disable the running task `T`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_AP)`: create an *AlternativeProvider* recovery transition,
4. `CreateArc(p1,Tr_AP)`: `p1` is the input place of the *AlternativeProvider* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *AlternativeProvider* recovery transition (see Figure 12(b)),
6. modify the SARN structure by executing the basic operations associated with the *AlternativeProvider* recovery transition (see Figure 12(c)),
7. run the added exceptional part of the SARN net,
8. remove the modifications made for the net once the exceptional part finishes its execution, and
9. resume the normal execution by transforming the recovery tokens on the output places of the substituted task to standard tokens (see Figure 12(d)).



(a) Task T running (b) An *AlternativeProvider* failure event occurs (c) Handling *AlternativeProvider* failure event (d) Resume normal execution

Figure 12: *AlternativeProvider* Recovery Policy

The basic operations associated with an *AlternativeProvider* recovery policy are as follows (see Figure 12(c)):

(a) `CreateArc(T,p1)`: add an incoming arc from the replaced task `T` to the input place of the recovery transition,

(b) `ReplaceTransition(Tr_AP,T')`, that is, replace the *AlternativeProvider* recovery transition with the task `T'` of the alternative provider `P`, and

(c) $\forall\ \texttt{p} \in \texttt{T}^{\bullet}$ $\texttt{CreateArc}(\texttt{T}', \texttt{p})$: add an outgoing arc from `T'` to each output place of the replaced task.

3.9 Timeout

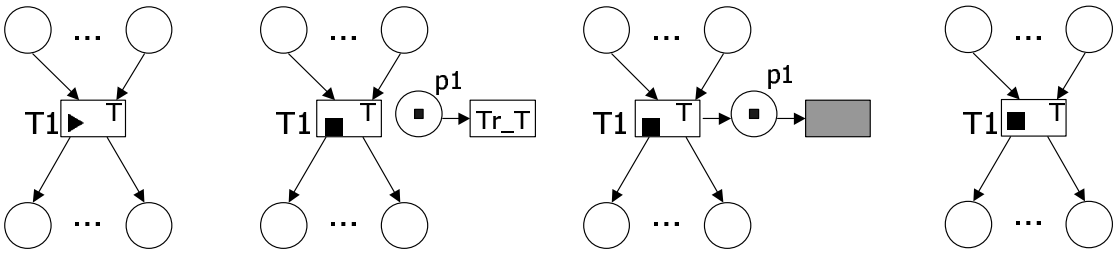
The *Timeout* recovery policy allows a time limit `d` to be associated with a task. The task fails after `d` units of time if it has not completed within that time.

In terms of our SARN model, a `Timeout(Task T1, Time d)` recovery policy of a task `T1` when its corresponding *Timeout* failure event occurs after `d` units of time means (see Figure 13):

Precondition: $state($`T1`$) =$ `Running`.

Effect:

1. `DisableTransition(T1)`: suspend the execution of the task `T1`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_T)`: create a *Timeout* recovery transition,
4. `CreateArc(p1,Tr_T)`: `p1` is the input place of the *Timeout* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *Timeout* recovery transition,
6. execute the elementary operations associated with the *Timeout* recovery transition,
7. execute the added exceptional part of the SARN net,
8. remove the modifications made, and
9. freeze the normal execution of the workflow.



(a) Task T1 running

(b) A *Timeout* failure event occurs

(c) Handling *Timeout* failure event

(d) Freeze normal execution

Figure 13: *Timeout* Recovery Policy

The operations associated with a *Timeout* recovery transition (to complete step (6) above) are as follows (see Figure 13(c)):

(a) `CreateArc(T1,p1)`: add an incoming arc from the suspended task `T1` to the input place of the recovery transition and

(b) `SilentTransition(Tr_T)`: replace the *Timeout* recovery transition with an empty task.

Table 2 gives a summary of the identified task-based recovery policies described previously.

Table 2: Task-Based Recovery Policies

Recovery Policy	Notation	Task Status	Brief Description
`Skip(Event e, Task T)`	S^e_T	Running	Skips the running task `T` to the immediate next task(s) if the event `e` occurs
`SkipTo(Event e, Task T, TaskSet` $\mathcal{T}$`)`	$ST^e_{T,\mathcal{T}}$	Running	Skips the running task `T` to the specific next task(s) $\mathcal{T}$ if the event `e` occurs
`Compensate(Event e, Task T)`	C^e_T	Completed	Removes the effect of an already executed task `T` if the event `e` occurs
`CompensateAfter(Event e, Task T)`	CA^e_T	Completed	Removes the effect of an already executed task `T` just after completing it if the event `e` occurs
`Redo(Event e, Task T)`	R^e_T	Completed	Repeats the execution of a completed task `T` if the event `e` occurs
`RedoAfter(Event e, Task T)`	RA^e_T	Completed	Repeats the execution of a completed task `T` just after finishing it if the event `e` occurs
`AlternativeTask(Event e, Task T, Task T')`	$AT^e_{T,T'}$	Running	Allows an alternative execution of a task `T` by another task `T'` if the event `e` occurs
`AlternativeProvider(Event e, Task T, Provider P)`	$AP^e_{T,P}$	Running	Allows an alternative execution of a task `T` by another provider `P` if the event `e` occurs
`Timeout(Task T, Time d)`	T^d_T	Running	Fails a task `T` if not completed within a time limit `d`. The execution is frozen

4 Region-Based Recovery Policies

The recovery policies defined in the previous section apply to a single task only. In this section, we will extend them to a *recovery region*, i.e., a set of tasks. We will first define the notion of recovery region and then extend some of the single task-based recovery policies to be applied to a recovery region.

4.1 Recovery Region

A *recovery region* is a connected set of places and transitions. Recovery regions are required to have a certain structure. We require that 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 make the definition of recovery regions clear, we

will separate them in such a way that, between the output place of a recovery region and the input place of the eventual subsequent recovery region(s), a silent transition will occur transferring the token from a recovery region to its subsequent recovery region(s). This will cause a non-overlapping of recovery regions, hence making them clearly separated.

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 recovery policies may be assigned. As such, a recovery region is more than a traditional subworkflow. Formally, a *recovery region* is defined as follows.

Definition 4.1 (Recovery Region)
Let $RN = (P, T, Tr, F, i, o, \ell, M)$ *be a SARN net. A recovery region is a subnet* $R = \langle P_R, T_R, F_R, i_R, o_R, \ell_R \rangle$ *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*,*
- $i_R \in P_R$ *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, and*
- *Let* $T_R = \{t \in T \mid t^\bullet \cap P_R \neq \varnothing \quad \wedge \quad {}^\bullet t \cap P_R \neq \varnothing\}$*. Then* R *must be connected (i.e., there is no isolated place or transition).* □

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.

4.2 Region-Based Recovery Policies

In what follows, we will discuss the identified region-based recovery policies. They are mainly based on an extension of their corresponding task-based recovery policies.

We will distinguish between eight region-based recovery policies, namely, *SkipRegion*, *SkipRegionTo*, *CompensateRegion*, *CompensateRegionAfter*, *RedoRegion*, *RedoRegionAfter*, *AlternativeRegion*, and *TimeoutRegion*.

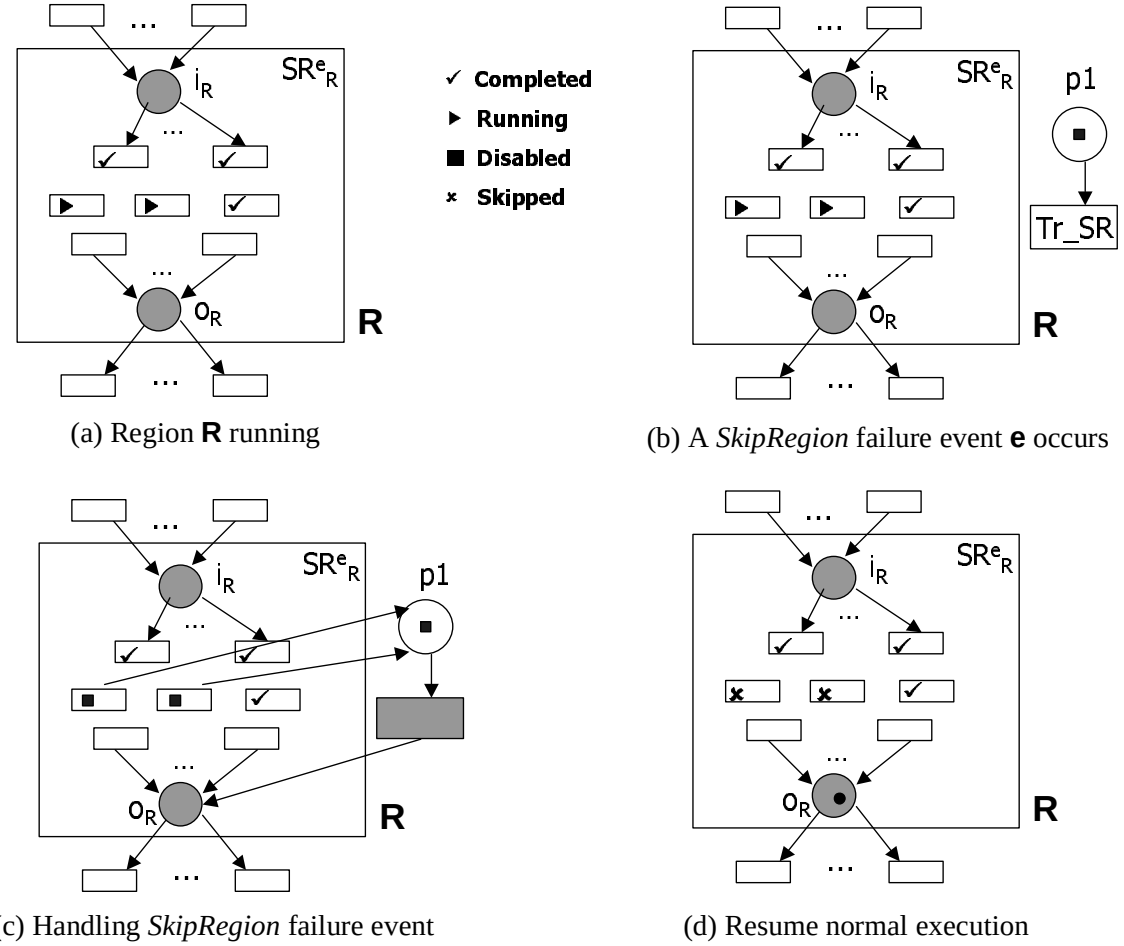
Note also that region-based recovery policies should only be allowed in a valid way. The system must ensure correctness of the modified SARN with respect to consistency constraints (reachability, liveness, and boundedness), so that these behavior properties that were valid before the dynamic change of SARN are also preserved after the handling of the exception.

4.2.1 SkipRegion

The *SkipRegion* recovery policy will, when the corresponding failure event occurs during the execution of the corresponding recovery region `R`: (i) disable the execution of the running tasks within the recovery region `R` and (ii) skip the non-completed tasks of the recovery region `R` to the immediate next task(s) of `R`. This recovery policy applies to

running recovery regions only, i.e., there are tasks within the recovery region `R` that are still running. This means that, eventually, some tasks within the recovery region are completed while others have not yet executed.

Formally, in the context of SARN, a `SkipRegion(Event e, Region R)` recovery policy when executing tasks of the recovery region `R` and the corresponding failure event `e` occurs means (see Figure 14):



(a) Region **R** running

(b) A *SkipRegion* failure event **e** occurs

(c) Handling *SkipRegion* failure event

(d) Resume normal execution

Figure 14: *SkipRegion* Recovery Policy

Precondition: $\exists$ `T` $\in T_R \mid state($`T`$) =$ `Running`.

Effect:

1. $\forall$ `T` $\in T_R \mid state($`T`$) =$ `Running` *do* `DisableTransition(T)`: disable all running tasks of the recovery region `R`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_SR)`: create a *SkipRegion* recovery transition,
4. `CreateArc(p1,Tr_SR)`: `p1` is the input place of the *Skip* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *SkipRegion* recovery transition (see Figure 14(b)),

6. execute the basic operations associated with the *SkipRegion* recovery transition to modify the net structure in order to handle the exception (see Figure 14(c)),
7. execute the added exceptional part of the SARN net,
8. once the exceptional part finishes its execution, i.e., there is no *recovery* token within the added net structure part, the modifications made for the task failure event are removed, and
9. resume the normal execution by transforming the recovery tokens on the output places of the skipped task into standard tokens (see Figure 14(d)).

The operations associated with a *SkipRegion* recovery transition, in order to complete step (7) above, are as follows (see Figure 14(c)):

(a) $\forall$ T $\in T_R \mid$ *state*(T) = `Running` *do* `CreateArc(T,p1)`: add an incoming arc from the running tasks of the skipped recovery region to the input place of the recovery transition,

(b) `SilentTransition(Tr_SR)`: replace the *SkipRegion* recovery transition with a silent transition, and

(c) `CreateArc(Tr_SR,`o_R`)`: add an outgoing arc from the silent transition to the output place o_R of the recovery region `R` to skip.

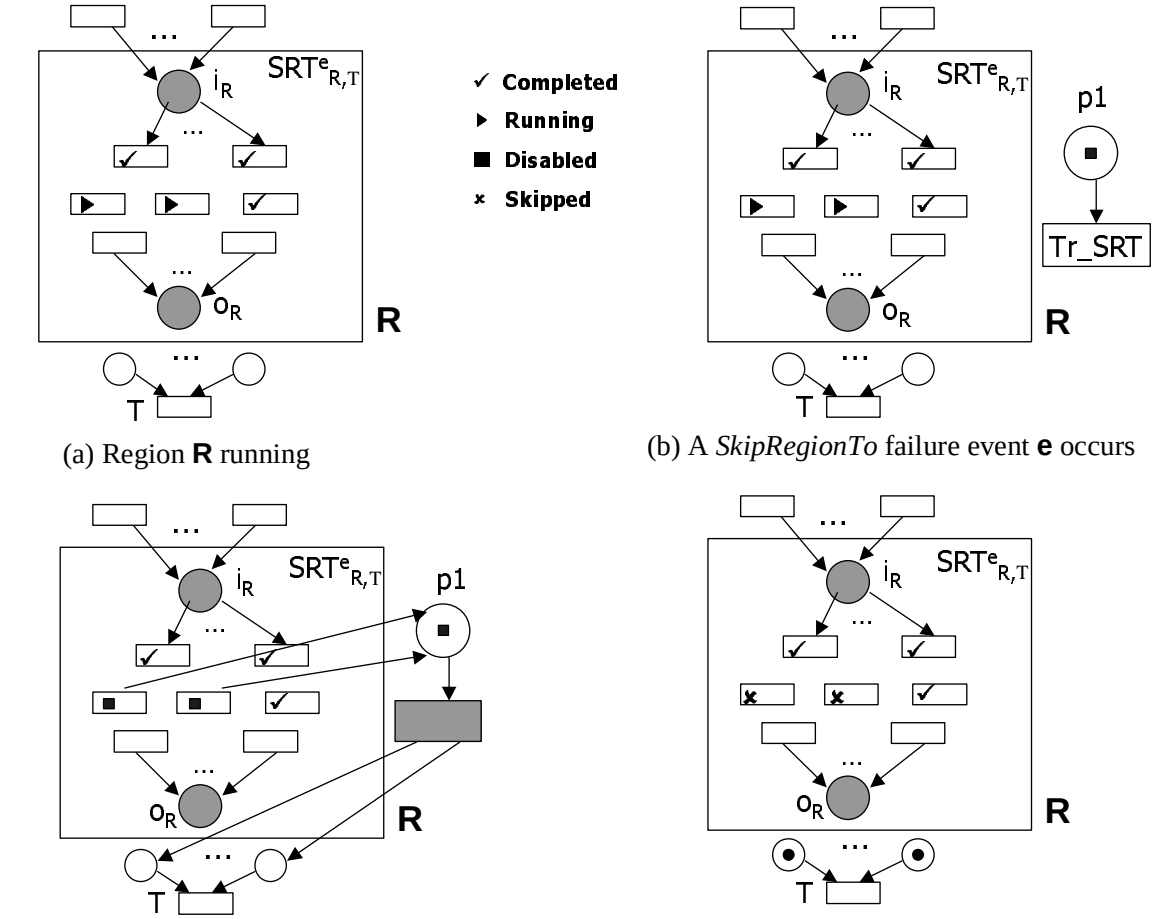
4.2.2 SkipRegionTo

The *SkipRegion* recovery policy defined previously is generic in the sense that there is no need to ask the workflow designer to which step of the execution she wants to resume the execution from. Indeed, when skipping the running recovery region, the immediate next task(s) of the recovery region with respect to the control flow will be executed. An interesting variant of the *SkipRegion* recovery policy could be to skip to certain task(s) and not necessarily to the immediate next task(s) of the recovery region. We will call this derived recovery policy *SkipRegionTo*. It should be noted that the tasks of the set of tasks $\mathcal{T}$ to skip to must be pairwise independent and each task of $\mathcal{T}$ must be a subsequent task of the skipped recovery region `R`.

Formally, a `SkipRegionTo(Event e, Region R, TaskSet` $\mathcal{T}$`)` recovery policy , when its corresponding failure event `e` occurs when executing tasks of the recovery region `R`, is defined as follows (see Figure 15):

Precondition:

- $\exists$ T $\in T_R \mid$ *state*(T) = `Running`,
- $\forall$ T1, T2 $\in \mathcal{T}$ $(\texttt{T1},\texttt{T2}) \notin F^+ \wedge (\texttt{T2},\texttt{T1}) \notin F^+$, that is, the tasks of $\mathcal{T}$ are pairwise independent with respect to the flow relation F, and
- $\forall$ T $\in \mathcal{T}$ $(o_R,\texttt{T}) \in F^+$, i.e., there is a path, with respect to the flow relation F, from the skipped recovery region `R` (that is, its output place o_R) to the task(s) of $\mathcal{T}$ to skip to (recall that F^+ denotes the transitive closure of F).



(a) Region **R** running

(b) A *SkipRegionTo* failure event **e** occurs

(c) Handling *SkipRegionTo* failure event

(d) Resume normal execution

Figure 15: *SkipRegionTo* Recovery Policy

Effect:

1. $\forall$ `T` $\in T_R$ | *state*(`T`) = `Running` *do* `DisableTransition`(`T`): disable all running tasks of the recovery region `R`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_SRT)`: create a *SkipRegionTo* recovery transition,
4. `CreateArc(p1,Tr_SRT)`: `p1` is the input place of the *SkipRegionTo* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *SkipRegionTo* recovery transition (see Figure 15(b)),
6. execute the elementary operations associated with the *SkipRegionTo* recovery transition (see Figure 15(c)),
7. execute the added exceptional part of the SARN net to handle the exception,
8. remove the modifications made for the skip region to failure event, and

9. resume the usual execution by transforming the recovery tokens on the input places of the task(s) to skip to into standard tokens (see Figure 15(d)).

The operations associated with a *SkipRegionTo* recovery transition are as follows (see Figure 6(c)):

(a) $\forall$ T $\in T_R \mid$ *state*(T) = `Frozen` *do* `CreateArc(T, p1)`: add an incoming arc from the running tasks of the skipped recovery region to the input place of the recovery transition,

(b) `SilentTransition(Tr_SRT)`: replace the *SkipRegionTo* recovery transition with a silent transition, and

(c) $\forall$ T $\in \mathcal{T}$ $\forall$ p $\in$ $^\bullet$T `CreateArc(Tr_SRT, p)`: add an outgoing arc from the silent transition to each input place of the task(s) to skip to.

4.2.3 CompensateRegion

The *CompensateRegion* recovery policy removes the effect of all already executed tasks of the completed or running recovery region. The tasks of the recovery region `R` must be compensatable, i.e., there is a `compensate-T` task that removes the effect of each task `T` of `R`. Note that the event of compensating a recovery region can occur any time after the completion of at least one task of the recovery region and before the business process execution terminates. Furthermore, we assume that there is no data flow dependencies between the tasks of the recovery region to be compensated and the subsequent completed task(s).

Formally, in the context of our model, a `CompensateRegion(Event e, Region R)` recovery policy of a recovery region `R` when its corresponding failure event `e` occurs means:

Precondition:

- $\exists$ T $\in T_R \mid$ *state*(T) = `Completed` and
- $\forall$ T $\in T_R$ T is *compensatable*, i.e., there is a `compensate-T` task of each task `T` of the recovery region `R` to be compensated.

Effect:

1. $\forall$ T $\in T_R \mid$ *state*(T) = `Running` *do* `DisableTransition(T)`: disable possibly (in case of compensating a running recovery region) all running transitions of the recovery region `R`,
2. $\forall$ t $\in T \mid (o_R, \mathtt{t}) \in F^+ \wedge$ *state*(t) = `Running` do `DisableTransition(t)`, hence possibly (in case of compensating a completed recovery region) all running subsequent task(s) of the recovery region `R` to be compensated are disabled,
3. `CreatePlace(p1)`: create a new place `p1`,
4. `CreateTransition(Tr_CR)`: create a *CompensateRegion* recovery transition,

5. `CreateArc(p1,Tr_CR)`: `p1` is the input place of the *CompensateRegion* recovery transition,
6. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *CompensateRegion* recovery transition,
7. execute the primitive operations associated with the *CompensateRegion* recovery transition,
8. execute the exceptional part of the SARN net,
9. remove the modifications made for the compensate region failure event, and
10. resume the execution of the workflow.

The operations associated with a *CompensateRegion* recovery transition are as follows:

(a) `ReplaceSequence(Tr_CR,compensate-`T_R^C`)`: associate to the *CompensateRegion* recovery transition the sequence of tasks `compensate-`T_R^C where $T_R^C = \{\mathtt{t} \in T_R \mid state(\mathtt{t}) = \mathtt{Completed}\}$ that removes the effects of the already completed tasks T_R^C of the recovery region `R` and

(b) if $\exists\ \mathtt{t} \in T_R \mid state(\mathtt{t}) = \mathtt{Frozen}$ then `CreateArc(compensate-`T_R^C`,`o_R`)`, i.e., add an outgoing arc from the `compensate-`T_R^C sequence of tasks to the output place o_R of the recovery region (see Figure 16). Otherwise, $\forall\ \mathtt{t} \in T \mid state(\mathtt{t}) = \mathtt{Frozen}\ \forall\ \mathtt{p} \in {}^\bullet\mathtt{t}$ *do* $\mathtt{CreateArc}(\mathtt{compensate} - \mathtt{T_R^C}, \mathtt{p})$, i.e., add an outgoing arc from the `compensate-`T_R^C sequence of tasks to each input place of the suspended running task(s) of the business process (see Figure 17).

4.2.4 CompensateRegionAfter

The *CompensateRegion* recovery policy removes the effects of the already completed tasks of a recovery region any time after the completion of at least one task of the recovery region and before the workflow execution terminates. An interesting case that will have no effects on the subsequent dependant tasks is when compensating a recovery region just after finishing its execution and before initiating any subsequent dependant task. We will call this particular *CompensateRegion* recovery policy *CompensateRegionAfter*.

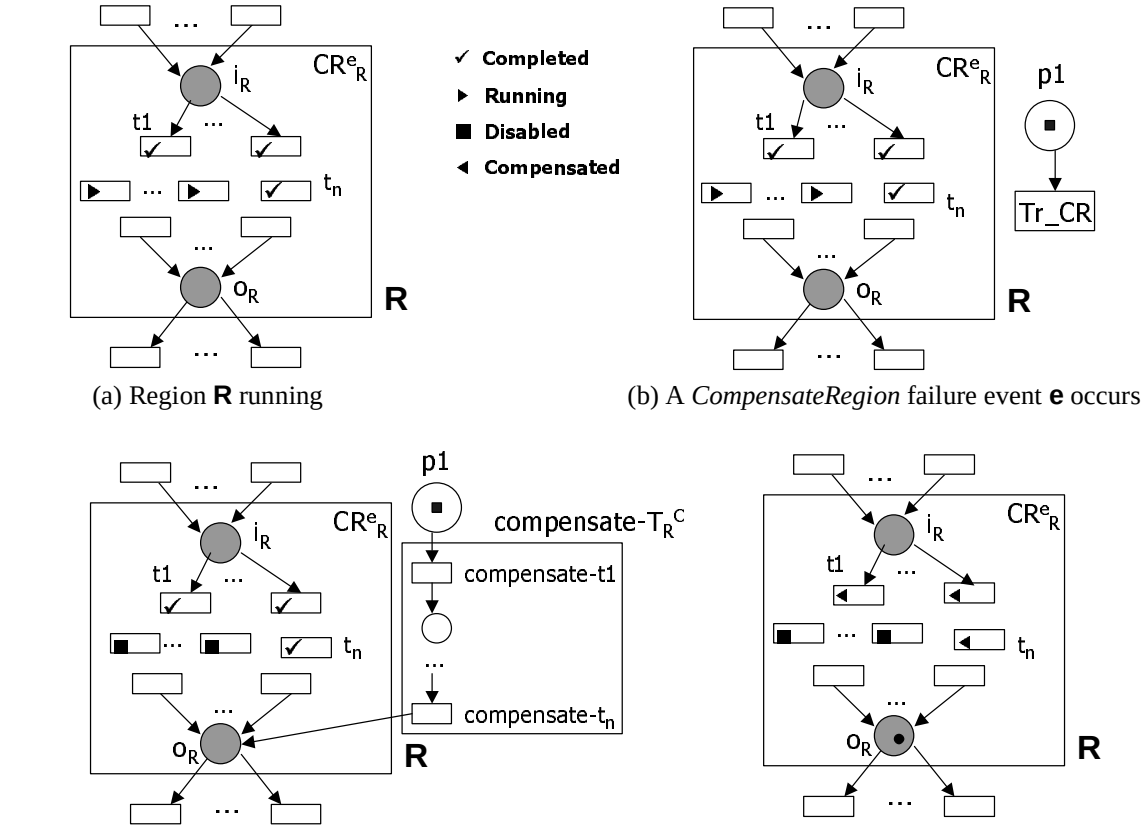
Formally, a `CompensateRegionAfter(Event e, Region R)` recovery policy of a recovery region `R` when its corresponding failure event `e` occurs means:

Precondition:

- $\forall\ \mathtt{T} \in T_R \mid state(\mathtt{T}) = \mathtt{Completed}$ and
- $\forall\ \mathtt{T} \in T_R$ `T` is *compensatable*, i.e., there is a `compensate-T` task of each task `T` of the recovery region `R` to be compensated.

Effect:

1. `CreatePlace(p1)`: create a new place `p1`,



(a) Region **R** running

(b) A *CompensateRegion* failure event **e** occurs

(c) Handling *CompensateRegion* failure event

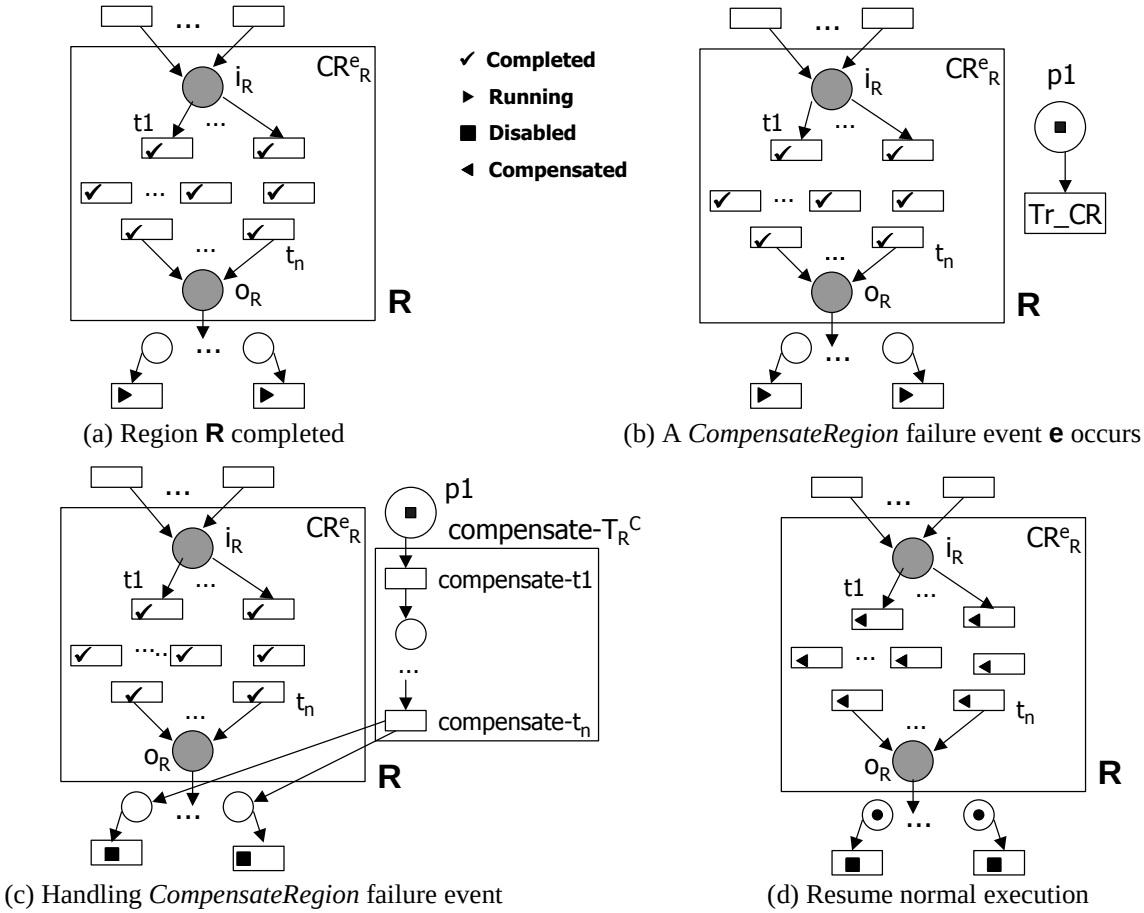
(d) Resume normal execution

Figure 16: *CompensateRegion* Recovery Policy of a Running Recovery Region

2. `CreateTransition(Tr_CRA)`: create a *CompensateRegionAfter* recovery transition,
3. `CreateArc(p1,Tr_CRA)`: `p1` is the input place of the *CompensateRegionAfter* recovery transition,
4. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *CompensateRegionAfter* recovery transition,
5. execute the basic operations associated with the *CompensateRegionAfter* recovery transition,
6. execute the added exceptional part of the SARN net to handle the exception,
7. remove the modifications made for the compensate region after failure event, and
8. resume the execution of the workflow.

The operations associated with a *CompensateRegionAfter* recovery transition are as follows:

(a) `ReplaceSequence(Tr_CRA,compensate-`T_R`)`: associate to the *CompensateRegionAfter* recovery transition the sequence of tasks `compensate-`T_R that removes the effects of the already completed tasks of the recovery region `R` and



(a) Region **R** completed

(b) A *CompensateRegion* failure event **e** occurs

(c) Handling *CompensateRegion* failure event

(d) Resume normal execution

Figure 17: *CompensateRegion* Recovery Policy of a Completed Recovery Region

(b) $\forall$ `t` $\in T \mid$ *state*(`t`) = `Frozen` $\forall$ `p` $\in$ $^\bullet$`t` do `CreateArc(compensate-`T_R`,p)`: add an outgoing arc from the `compensate-`T_R sequence of tasks to each input place of the suspended running task(s) of the business process.

4.2.5 RedoRegion

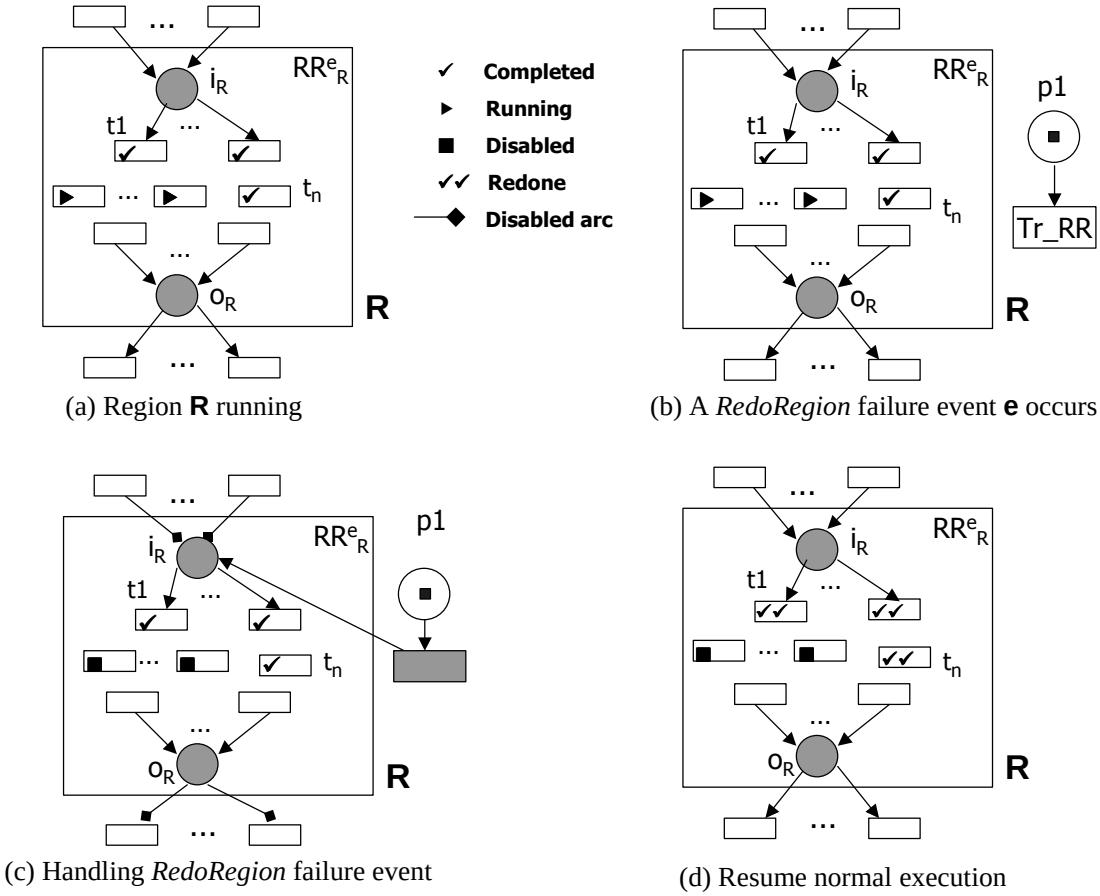
The *RedoRegion* recovery policy repeats the execution of all already completed tasks of the (completed or running) recovery region `R`. Note that, like the *CompensateRegion* recovery policy, the event of redoing a recovery region can occur any time after the completion of at least one task of the recovery region and before the business process execution terminates. Furthermore, we assume that there is no data flow dependencies between the tasks of the recovery region to be repeated and the subsequent completed task(s).

Formally, a `RedoRegion(Event e, Region T)` recovery policy of the recovery region `R` when its corresponding failure event `e` occurs means (see Figure 18):

Precondition: $\exists$ `T` $\in T_R \mid$ *state*(`T`) = `Completed`.

Effect:

1. $\forall$ `T` $\in T_R \mid$ *state*(`T`) = `Running` *do* `DisableTransition`(`T`): possibly (in case of



(a) Region **R** running

(b) A *RedoRegion* failure event **e** occurs

(c) Handling *RedoRegion* failure event

(d) Resume normal execution

Figure 18: *RedoRegion* Recovery Policy

redoing a running recovery region) disable all running transitions of the recovery region `R`,

2. $\forall$ `t` $\in T \mid (o_R, \texttt{t}) \in F^+ \wedge state(\texttt{t}) = \texttt{Running}$ do `DisableTransition(t)`: disable possibly (in case of redoing a completed recovery region) all running subsequent task(s) of the recovery region to be repeated,

3. `CreatePlace(p1)`: create a new place `p1`,

4. `CreateTransition(Tr_RR)`: create a *RedoRegion* recovery transition,

5. `CreateArc(p1,Tr_RR)`: `p1` is the input place of the *RedoRegion* recovery transition,

6. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *RedoRegion* recovery transition (see Figure 18(b)),

7. execute the elementary operations associated with the *RedoRegion* recovery transition,

8. execute the added exceptional part of the SARN net,

9. remove the modifications made for the redo region failure event,

10. remove possibly (in case of redoing a completed recovery region) the recovery token from the output place o_R of the recovery region `R`, and

11. resume the execution of the workflow (see Figure 18(d)).

The operations associated with a *RedoRegion* recovery transition (to complete step (7) above) are as follows (see Figure 9(c)):

(a) disable all incoming arcs of the input place i_R of the recovery region to be repeated,

(b) `SilentTransition(Tr_RR)`: replace the *RedoRegion* recovery transition with an empty task,

(c) add an outgoing arc from the empty *RedoRegion* recovery transition to the input place i_R of the recovery region `R`,

(d) disable all outgoing arcs of the output place o_R of the recovery region, and

(e) add an outgoing arc from the empty *RedoRegion* recovery transition to each input place of the possibly (in case of redoing a completed recovery region) disabled subsequent tasks of the recovery region to be repeated.

4.2.6 RedoRegionAfter

The *RedoRegion* recovery policy repeats the execution of the already completed tasks of a recovery region any time after the completion of at least one task of the recovery region and before the workflow execution terminates. An interesting case that will have no effects on the subsequent dependant tasks is when redoing a recovery region just after finishing its execution and before initiating any subsequent dependant task. We will call this particular *RedoRegion* recovery policy *RedoRegionAfter*.

Formally, a `RedoRegionAfter(Event e, Region R)` recovery policy of a recovery region `R` when its corresponding failure event `e` occurs means (see Figure 19):

Precondition: $\forall$ `T` $\in T_R \mid$ *state*(`T`) = `Completed`.

Effect:

1. `CreatePlace(p1)`: create a new place `p1`,

2. `CreateTransition(Tr_RRA)`: create a *RedoRegionAfter* recovery transition,

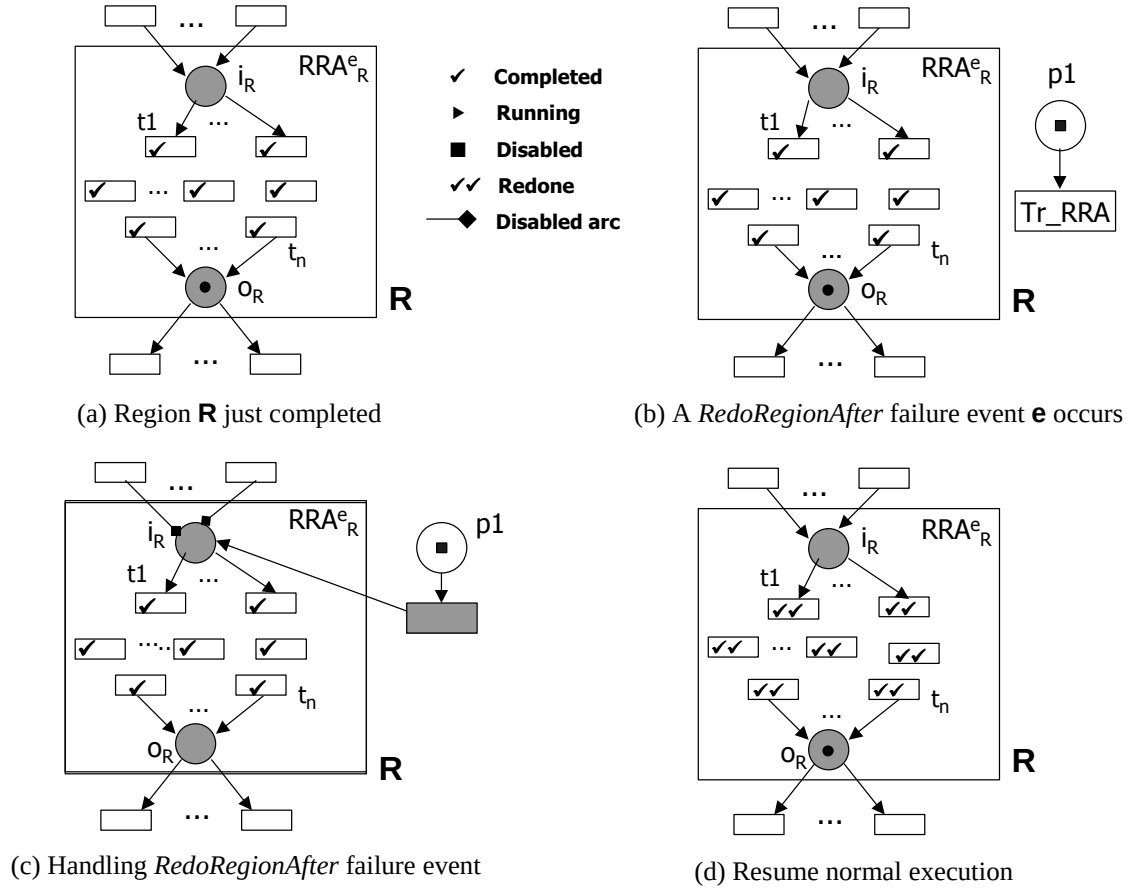
3. `CreateArc(p1,Tr_RRA)`: `p1` is the input place of the *RedoRegionAfter* recovery transition,

4. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *RedoRegionAfter* recovery transition (see Figure 19(b)),

5. execute the primitive operations associated with the *RedoRegionAfter* recovery transition,

6. execute the added exceptional part of the SARN net to handle the exception,

7. remove the modifications made for the redo region after failure event, and



(a) Region **R** just completed

(b) A *RedoRegionAfter* failure event **e** occurs

(c) Handling *RedoRegionAfter* failure event

(d) Resume normal execution

Figure 19: *RedoRegionAfter* Recovery Policy

8. resume the normal execution of the workflow (see Figure 19(d)).

The operations associated with a *RedoRegionAfter* recovery transition (to complete step (5) above) are as follows (see Figure 10(c)):

(a) disable all incoming arcs of the input place i_R of the recovery region to be repeated,

(b) `SilentTransition(Tr_RRA)`: replace the *RedoRegionAfter* recovery transition with an empty task,

(c) add an outgoing arc from the empty *RedoRegionAfter* recovery transition to the input place i_R of the recovery region `R`, and

(d) remove the standard token from the output place o_R.

4.2.7 AlternativeRegion

The *AlternativeRegion* recovery policy allows another recovery region `R'` to be executed in place of a running recovery region `R` in case the later fails.

Formally, in the context of SARN, an `AlternativeRegion(Event e, Region R, Region R')` recovery policy of a recovery region `R` by another recovery region `R'` when its corresponding failure event `e` occurs means:

Precondition: $\exists$ `T` $\in T_R \mid$ *state*(`T`) = `Running`.

Effect:

1. $\forall$ `T` $\in T_R \mid$ *state*(`T`) = `Running` do `DisableTransition(T)`: disable all running tasks of the recovery region `R`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_AR)`: create an *AlternativeRegion* recovery transition,
4. `CreateArc(p1,Tr_AR)`: `p1` is the input place of the *AlternativeRegion* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *AlternativeRegion* recovery transition,
6. modify the SARN structure by executing the basic operations associated with the *AlternativeRegion* recovery transition,
7. run the added exceptional part of the SARN net,
8. remove the modifications made for the SARN net once the exceptional part finishes its execution, and
9. resume the normal execution by transforming the recovery token on the output place o_R of the recovery region `R` into a standard token.

The basic operations associated with an *AlternativeRegion* recovery transition are as follows:

(a) disable all outgoing arcs of the output place o_R of the recovery region to be replaced,

(b) `ReplaceRegion(Tr_AR,R')`: replace the *AlternativeRegion* recovery transition with the alternative recovery region `R'`, and

(c) $\forall$ `t` $\in o_R^{\bullet}$ `CreateArc(`$o'_{\mathtt{R}}$`,t)`: add an incoming arc from the output place o'_R of the alternative recovery region `R'` to each output task of the output place o_R of the recovery region `R`.

4.2.8 TimeoutRegion

The *TimeoutRegion* recovery policy allows a time limit `d` to be associated with a recovery region. The recovery region fails after `d` units of time if it has not completed within that time.

In terms of our SARN model, a `TimeoutRegion(Region R, Time d)` recovery policy of a recovery region `R` when its corresponding timeout region failure event occurs, after `d` units of time, means:

Precondition: $\exists$ `T` $\in T_R \mid$ *state*(`T`) = `Running`.

Effect:

1. $\forall$ `T` $\in T_R \mid$ *state*(`T`) = `Running` do `DisableTransition(T)`: disable all running tasks of the recovery region `R`,
2. `CreatePlace(p1)`: create a new place `p1`,
3. `CreateTransition(Tr_TR)`: create a *TimeoutRegion* recovery transition,
4. `CreateArc(p1,Tr_TR)`: `p1` is the input place of the *TimeoutRegion* recovery transition,
5. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *TimeoutRegion* recovery transition,
6. execute the elementary operations associated with the *TimeoutRegion* recovery transition,
7. execute the added exceptional part of the SARN net,
8. remove the modifications made, and
9. freeze the normal execution of the workflow.

The operations associated with a *TimeoutRegion* recovery transition (to complete step (6) above) are as follows:

(a) $\forall$ `T` $\in T_R \mid$ *state*(`T`) = `Frozen` do `CreateArc(T,p1)`: add an incoming arc from the suspended task `T` of `R` to the input place of the *TimeoutRegion* recovery transition and

(b) `SilentTransition(Tr_TR)`: replace the *TimeoutRegion* recovery transition with an empty task.

Table 3 gives a summary of the identified region-based recovery policies.

4.3 Correctness Preservation

The SARN net generated by using the previously defined task- and region-based recovery policies is a consistent net that satisfies the behavior properties defined in Section 2.4 (i.e., reachability, liveness, and boundedness).

Proposition 4.1 (Correctness Preservation)
The SARN net RN of a workflow obtained after handling an exception using the above defined task- and region-based recovery policies is valid, i.e., the reachability, liveness, and boundedness properties are preserved. □

Proof. Immediate consequence of Definition 2.1 and of the task- and region-based recovery policies defined previously. □

Table 3: 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

5 Related Work and Conclusions

Some studies have considered the problem of exception handling and recovery from activity failures in WfMSs, such as [EKR95, CCPP98, JH98, RD98, Kli00]. Leymann [LR00] 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 very complicated semantics. The model of Hagen and Alonso [HA00] 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. The approach of Hwang et al. [HHT99] supports users in handling exceptions once they have occurred. Exception handling suggestions with information about the way similar situations were handled in previous executions are provided to users at run time. Grigori et al. [GCDS01] 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 [AAA+96, Elm92, GMS87, GHM96, WR92, WS92]. They particularly show how some of the concepts used in transaction management can be applied to workflow environments.

In this report, we proposed the Self-Adaptive Recovery Net (SARN) model for specifying exceptional behavior in WfMSs at design time. SARN allows the handling of prespecified events at run time while keeping the Petri net design simple and easy. For existing models to realize the same functionality, the design effort could be tremendous. We also identified a set of high-level recovery policies that are incorporated with either a single task or a recovery region. SARN can handle not only commonly predefined recovery policies (e.g., *Skip* and *Compensate*) but the user is also free to define new recovery policies. By introducing a set of primitive operations, SARN can be adapted at run time to handle the occurrence of exceptions while keeping the underlying Petri net design simple and easy. This method is particularly interesting in designing flexible and self-adaptive business processes.

To illustrate the viability of our approach, we are currently developing SARN simulator as part of *HiWorD* (HIerarchical WORkflow Designer), a hierarchical Petri net-based workflow modeling tool [BCH+03, CBH+03].

References

[AAA+96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*, New Orleans, USA, February 1996. IEEE Computer Society.

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

[BCTH03] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Service Conversations. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03)*, volume 2681, pages 449–467, Klagenfurt, Austria, June 2003. Springer Verlag.

[CBH+03] P. Chrzastowski-Wachtel, B. Benatallah, R. Hamadi, M. O'Dell, and A. Susanto. A Top-Down Petri Net-Based Approach for Dynamic Workflow Modeling. In *Proceedings of the International Conference on Business Process Management (BPM'03)*, volume 2678, pages 336–353, Eindhoven, The Netherlands, June 2003. Springer Verlag.

[CCPP98] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998.

[EKR95] C.A. Ellis, K. Keddara, and G. Rozenberg. Dynamic Change within Workflow Systems. In *Proceedings of the Conference on Organizational Computing Systems (COOCS'95)*, pages 10–21, Milpitas, USA, August 1995. ACM Press.

[Elm92] A.K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[GCDS01] D. Grigori, F. Casati, U. Dayal, and M.-C. Shan. Improving Business Process Quality through Exception Understanding, Prediction, and Prevention. In *Proceedings of the 27th Very Large Data Base Conference (VLDB'01)*, Rome, Italy, September 2001.

[GHM96] D. Georgakopoulos, M.F. Hornick, and F. Manola. Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):630–649, 1996.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2), April 1995.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD*, San Francisco, USA, 1987.

[GSCB99] D. Georgakopoulos, H. Schuster, A. Cichocki, and D. Baker. Managing Process and Service Fusion in Virtual Enterprises. *Information Systems, Special Issue on Information Systems Support for Electronic Commerce*, 24(6):429–456, 1999.

[HA00] C. Hagen and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering (TSE)*, 26(10):943–958, October 2000.

[HHT99] S. Hwang, S. Ho, and J. Tang. Mining Exception Instances to Facilitate Workflow Exception Handling. In A.L.P. Chen and F.H. Lochovsky, editors, *Proceedings of the Sixth International Conference on Database Systems for Advanced Applications (DASFAA'99)*, Hsinchu, Taiwan, April 1999. IEEE Computer Society.

[JH98] G. Joeris and O. Herzog. Managing Evolving Workflow Specifications. In *Proceedings of the 3rd Conference on Cooperative Information Systems (CoopIS'98)*, New York, USA, August 1998.

[Kli00] J. Klingemann. Controlled Flexibility in Workflow Management. In *Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE'00)*, Stockholm, Sweden, June 2000.

[LR00] F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques*. Prentice Hall, 2000.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE*, volume 77(4), pages 541–580, April 1989.

[Pet62] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, Germany, 1962. (In German).

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, 1981.

[RD98] M. Reichert and P. Dadam. ADEPT flex: Supporting Dynamic Changes of Workflows without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science Vol.4. Springer-Verlag, Berlin, Germany, 1985.

[WfM99] WfMC. *Workflow Management Coalition, Terminology and Glossary.* Document Number WFMC-TC-1011, February 1999. http://www.wfmc.org/standards/docs.htm/.

[WR92] H. Wächter and A. Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–264. Morgan Kaufmann, 1992.

[WS92] G. Weikum and H.J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.