Self-adapting recovery nets for policy-driven exception handling in business processes

**Rachid Hamadi · Boualem Benatallah ·
Brahim Medjahed**

Abstract In this paper, we propose Self-Adapting Recovery Net (SARN), an extended Petri net model, for specifying exceptional behavior in business processes. SARN adapts the structure of the underlying Petri net at run time to handle exceptions while keeping the Petri net design easy. The proposed framework caters for the specification of 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 directives that model exceptions at design time together with a set of primitive operations used at run time to handle the occurrence of exceptions. We identified a set of recovery policies that are useful and commonly needed in many practical situations. A tool has been developed to illustrate the viability of the proposed exception handling technique.

R. Hamadi (✉) · B. Benatallah
School of Computer Science and Engineering, The University of New South Wales, Sydney,
Australia
e-mail: rhamadi@cse.unsw.edu.au

B. Benatallah
e-mail: boualem@cse.unsw.edu.au

B. Medjahed
Department of Computer and Information Science, University of Michigan, Dearborn, USA
e-mail: brahim@umich.edu

1 Introduction

A business process (BP) is a set of tasks which are performed collaboratively to realize a business objective [35, 46]. For example, a *Travel Planning* BP can offer full vacation packages by combining several tasks such as *Flight Booking*, *Hotel Booking*, and *Car Rental*. A business process management system (BPMS) provides a central control point for defining BPs and orchestrating their execution [7, 20]. One important requirement for a BPMS is fault-tolerance. While fault-tolerance and reliability may be defined at different levels (e.g., messaging level), we are interested in this paper in fault-tolerance at the business process level. We define fault-tolerance as the property that allows BPs to respond gracefully to expected but unusual situations and failures (also called *exceptions*). Exceptions constitute events which may occur during business process execution and require deviations from the normal business process behavior. These events can be *generic* such as the unavailability of resources, a time-out, and incorrect input types, or *application specific* such as the unavailability of seats for the *Flight Booking* task in the *Travel Planning* BP. In this latter case, exception events are usually defined by the business process modeler. Exception handling is, therefore, part of the business logic of a BP and may end up dominating its normal behaviour [24]. Existing business process modeling languages such as state machines, statecharts [27, 28], and Petri nets [39, 40] 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 provide specific abstractions for exception modeling.

There are two main approaches that deal with exceptions in existing BPMSs: *Ad Hoc* and *Run Time*. The former specifies the exception handling logic within the normal behavior of the BP. This makes the design of the BP complicated for the designer. The raising of expected exceptions is typically unpredictable. It is, thus, not convenient to represent exceptions directly in the normal behavior of the business process model. Indeed, expected exceptions are not frequent, but once they occur, they may require special treatment. The latter deals with exceptions at run time, meaning that there must be a business process expert who decides which changes have to be made to the business process logic in order to handle exceptions (see, e.g., [41]).

In this paper, we propose Self-Adapting Recovery Net (SARN), an extended Petri net model for specifying exceptional behavior in BPs [25, 26]. SARN adapts the mechanisms of the underlying Petri net at run time to handle exceptions while keeping the Petri net design simple and easy. Although we use a Petri net-based model, the proposed techniques can be based on other formal models such as statecharts [27, 28]. The choice of Petri nets for modeling BPs is motivated by the following reasons: (i) they possess a formal semantics which is essential for analysing service-based BPs, (ii) they have a graphical representation, and (iii) they are suitable for expressing typical control flow constructs such as sequence, parallel, choice, and iteration.

SARN uses 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* [25, 26]. These 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. Our proposed approach concentrates on handling exceptions at the instance

level and not on modifying the business process schema such as in [7]. We identify 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 business process modeling languages is a delicate issue. While, in general, new policies may provide the support and flexibility described above, they also make the business process model more complex. Complexity severely compromises the usability and 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. To achieve this goal, we determine a minimal set of recovery policies that are useful and needed in practice to adequately model and handle most of the exceptions. We focus on recovery policies that are commonly used in practice. The list of recovery policies is, however, not exhaustive. Indeed, new (customized) recovery policies can be added. It should be noted that our focus in this paper is not on exception event detection. Techniques such as the ones defined in the *Rainbow* framework can be used for that purpose [19]. The approach proposed in this paper extends the one introduced in a preliminary version [26]. In particular, we present a comprehensive description of task-based recovery policies (with seven additional policies). Furthermore, we provide a more detailed coverage of region-based recovery policies (with two additional policies) and describe a SARN tool implementation. The motivating scenario is expanded to illustrate the way BPs are modeled as SARNs. Finally, a detailed and in-depth description of related work is included.

The rest of the paper is organized as follows. Section 2 discusses exception handling in business processes as well as a motivating scenario. Section 3 introduces the proposed model SARN. Section 4 presents the task-based recovery policies. Their extension to a recovery region is given in Sect. 5. Section 6 describes the SARN tool. Section 7 reviews some related work. Finally, Sect. 8 concludes the paper.

2 Exception handling in business processes

In this section, we first present a classification of exceptions in BPs, then discuss the main differences between exceptions occurring in BPs and exceptions occurring in databases, and finally describe a running example.

2.1 Exception classification

As stated before, exceptions constitute events which may occur during the execution of a BP and require deviations from the normal BP behavior. Some authors classify exceptions according to system levels [7]. Events that result in such exceptions are database management system, operating system, or network failures and breakdowns. In our work, we only deal with expected exceptions. Unanticipated exceptions can not be handled and they leave the BP in an undefined state.

In addition of dividing events that generate exceptions into *generic* and *application specific* as discussed in the Introduction section, the identified exceptions may span different components of a BP: *user* (of a BP), *task* (of a BP), and (task or BP)

duration. It is therefore natural to adopt the following classification into three categories:

- *User exceptions*: triggered by the user of a BP. The exception events "cancel bike", "modification of flight date and time", "no hotel booking to be made", and "change attraction" of the *Travel Planning* BP are examples of user-generated exceptions.
- *Task exceptions*: related to the unsuccessful outcome of task executions. This can be due to: (1) *data event* such as the "unavailability of seats" for the *Flight Booking* task in the *Travel Planning* BP, (2) *physical unavailability* such as network and server failures, and (3) *logical unavailability* such as, if a task is implemented as a Web service, a change in service's input/output parameters by adding, e.g., additional inputs. In this case, the service is still available for invocation but the BP can not use it anymore.
- *Duration exceptions*: time related exceptions. Each task (and consequently a BP) has an estimated duration. An example of a duration exception is a time-out. A time-out allows a time limit to be associated with a task. The duration exception event is raised if the task has not completed its execution within that time limit.

Business processes are descriptions of an organisation's activities implemented as information processes and/or material processes [20]. Since exceptions have direct impact on the outcome of a BP, they are defined in the context of the BP objective. Exception handling mechanisms consist of three steps: exception detection, diagnosis, and handling. In our work, we mainly focus on the last step used to resolve the exception by applying a recovery policy.

The main difference between a BP and a database exceptions is that database exceptions refer to the sudden unavailability of a particular resource. But BP exceptions are more complicated. A BP exception might occur caused by the modification of a service's syntactic features (such as its input/output parameters). The service is still available for invocation but the task of the BP, implemented as a Web service, can not use it. Another difference is that exceptions in a database often refer to failures (because of a crash or due to concurrency problems) [17]. However exceptions in a BP may in addition refer to task and user events as described above. Furthermore, database exceptions focus on a lower-level view (i.e., data) but BP exceptions focus on a higher-level view (i.e., application).

2.2 Motivating scenario

A *Travel Planning* BP can offer full vacation packages by combining several existing services such as *Flight Booking*, *Hotel Booking*, and *Car Rental*. A simplified *Travel Planning* BP specified as a Petri net model is depicted in Fig. 1.

In this BP, 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.

Note that black or grey rectangles stand for silent or empty transitions (i.e., transitions with no associated tasks) and when two arcs stem from the same place (in
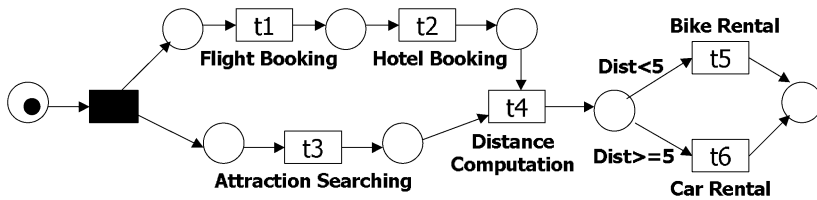
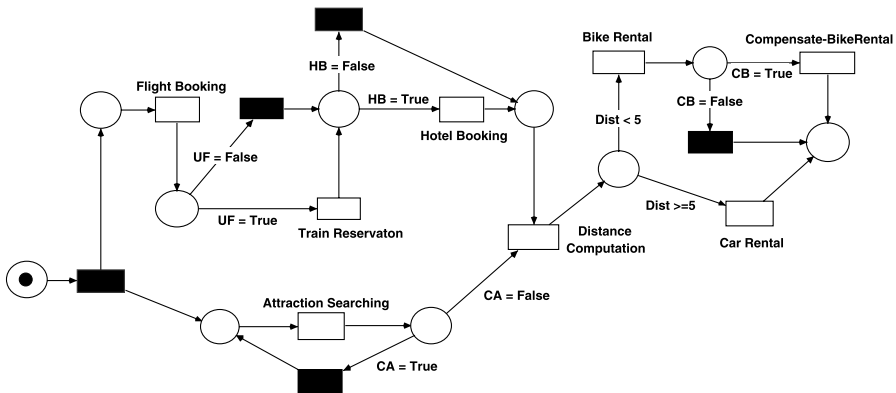**Fig. 1** *Travel Planning* business process

Fig. 2 Handling exceptions as part of the business logic

this case the place following *Distance Computation* task), they denote a conditional branching, and the arcs should therefore be labelled with disjoint conditions ("*Dist* < 5" and "*Dist* >= 5" in our case).

In what follows, we describe the application of the proposed exception handling mechanism on the *Travel Planning* business process scenario. The objective is to illustrate the advantage of the proposed approach compared to other "more traditional" exception handling techniques that integrate the necessary steps to handle exceptions directly into the business logic [1, 24].

Figure 2 shows the *Travel Planning* example that includes all necessary steps for handling examples of exceptions. It includes recovery mechanisms such as compensation: the task *Compensate-BikeRental* if occurrence of the exception event "cancel bike" (CB), alternative task: *Train Reservation* is a replacement for *Flight Booking* when the exception event "unavailability of flights or seats" (UF) occurs, skipping a task: *Hotel Booking* is skipped if occurrence of the exception event "no hotel booking to be made" (HB), as well as redoing a task: *Attraction Searching* is repeated when the exception event "change attraction" (CA) occurs.

The interleaving of the business logic with the exception logic makes the recovery version of the *Travel Planning* BP very complex and the original business logic hardly recognizable. The complexity in this simple example points out the drawbacks of including exception handling as part of the normal business logic behavior. Mixing business logic and exception handling logic makes it difficult to keep track of both, complicating the verification of BPs, as well as their later modifications. It is
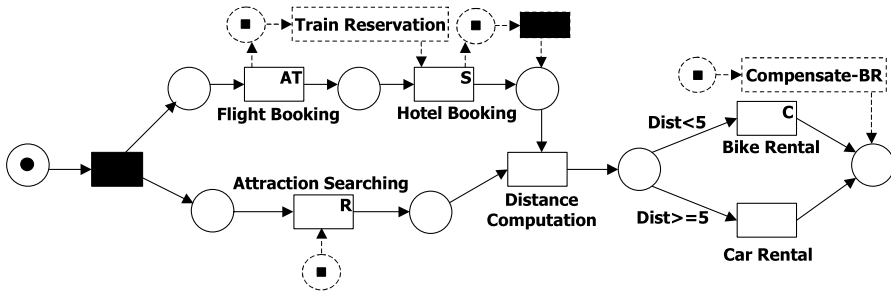
Fig. 3 Business process of Fig. 2 as a SARN

advantageous in such a situation to separate the code corresponding to the exception handling from the normal behavior logic of the BP. Indeed, a clear modularization of this code will ease future business process maintenance allowing: (i) to handle version change of the business logic in a centralized way and (ii) an isolated versioning of the adaptation in response to exception events.

Using SARN's generic directives, the designer is now able to simply and clearly define high-level recovery policies that apply when an exception occurs. Figure 3 shows the BP of Fig. 2 as a SARN model. The parts drawn in dotted lines are executed only if the corresponding exception event occurs.

The identification of recovery patterns and abstracting them into policies can be used to: (i) simplify the design of exception handling and (ii) automate exception handling processes by, e.g., generating exception handling controllers from specifications.

3 Self-adapting recovery net

Self-Adapting Recovery Net (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 business process tasks to be performed and *recovery* transitions that are associated with business process 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 (such a skip or a time out) is designed, a new recovery transition is added. When an exception 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 assume an environment where agents (humans or machines) are responsible for executing business process tasks. Each task has an estimated duration. Different methods are used for obtaining the estimated task durations: they can be given by the business process designer or calculated, e.g., based on statistical information of business

process log files. Note that we do not focus on exception event detection and that silent tasks, called *dummy* activities in [46], have no associated work or resources. We give below a formal definition of SARN.

Definition 1 (SARN) A Self-Adapting Recovery Net (SARN) is defined as a tuple $RN = (P, T, Tr, F, i, o, \ell, M)$ where:

- P is a finite set of places representing the states of the BP,
- T is a finite set of *standard* transitions ($P \cap T = \varnothing$) representing the tasks of the BP,
- Tr is a finite set of *recovery* transitions ($T \cap Tr = \varnothing$ and $P \cap Tr = \varnothing$) associated with business process tasks to adapt the net in-progress when a corresponding 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 BP with ${}^{\bullet}i = \varnothing$,
- o is the output place of the BP with $o^{\bullet} = \varnothing$,
- $\ell : T \to \mathcal{A} \cup \{\tau\}$ 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
- $M : P \to \mathbb{N} \times \mathbb{N}$ 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 primitive operations that can modify the net structure such as adding an arc and disabling a transition (see Table 1). Several of these primitive operations are combined in a specific order to handle different task 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 primitive 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.

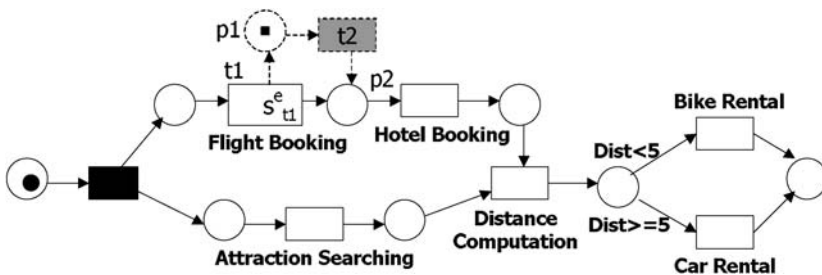In Fig. 4, the *Travel Planning* BP (see Fig. 1) as a SARN model is illustrated.

Fig. 4 *Travel Planning* BP as a SARN

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 put out of action
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, will be able to fire
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

The symbol S_{t1}^e 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 basic operations associated with the *Skip* recovery transition has been executed. When a *Skip* exception 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.

In the example depicted in Fig. 4, the *Skip* recovery policy S_{t1}^e is associated with eight basic operations (see Table 1): (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 primitive operations must be executed as an atomic transaction.

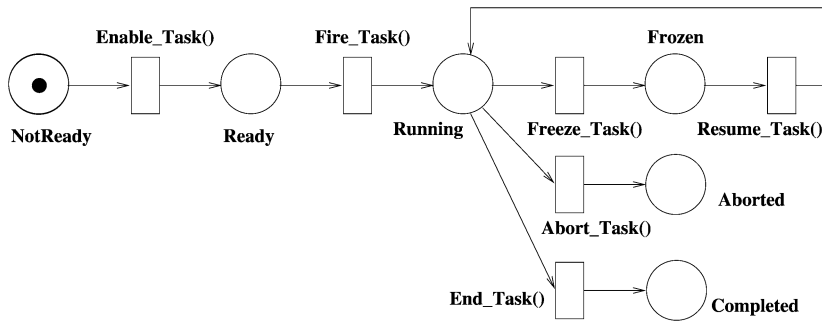[1]Details about recovery policies will be discussed in Sect. 4.

Fig. 5 Generic task states

3.1 Task states

Each task in a BP contains a task state variable. The latter is associated with a task state type that determines the possible task states [21]. A transition from one task state to another constitutes a primitive task event. Figure 5 shows the generic task states. It is consistent with the proposed standard of the Workflow Management Coalition [46]. 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 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.

3.2 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. If no task exception events occur, SARN will obey this enabling rule.

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

1. The recovery transition corresponding to this exception 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 exception event.

3. Once a recovery transition is created, the basic operations associated with it will be executed. The net structure will be modified to handle the corresponding exception. 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 exception will be removed. The net structure will be restored to the configuration before the occurrence of the exception 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.

3.3 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 *absence of deadlock*. For instance, the system must not allow to skip 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 specific parts of the SARN model). SARN must then meet certain consistency constraints in order to ensure the correct execution of the underlying BP 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.

Definition 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 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 $\sigma = M_0 t_1 M_1 t_2 M_2 \cdots t_n M_n$ or simply $\sigma = t_1 t_2 \cdots t_n$. In this case, M_n is reachable from M_0 by σ and we write $M_0[\sigma\rangle M_n$.

We also 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.

Definition 3 (Liveness) A SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ is 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.

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.

Definition 4 (Boundedness) A SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ is 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 safe if it is 1-bounded.

A SARN model is *valid* if it satisfies the three properties (i.e., reachability, liveness, and boundedness).

4 Task-based recovery policies

In this section, we introduce the proposed task-based recovery policies. We first informally describe the recovery policy, then give a formal definition, and finally, give an example to illustrate how a BP modelled as a SARN behaves at run time. We identify nine task-based recovery policies, namely, *Skip*, *SkipTo*, *Compensate*, *CompensateAfter*, *Redo*, *RedoAfter*, *AlternativeTask*, *AlternativeProvider*, and *TimeOut* (see Table 2). Note that this list is not exhaustive. Indeed, customized task-based recovery policies can be added. The addition of a new recovery policy requires providing the following four components: (1) a name for the recovery policy, (2) list of parameters (e.g., the set $\mathcal{T}$ of tasks to be skipped at the occurrence of a certain event e), (3) pre-condition, and (4) effect. A pre-condition is a first-order predicate on the current status of the SARN net (e.g., state of the current task). The predicate should evaluate to true to execute the effect of the recovery policy. The effect component includes the sequence of actions to be executed as part of the policy. The actions may combine SARN primitive operations (defined in Table 1) and other pre-existing recovery policies.

4.1 Skip

The *Skip* recovery policy will, once the corresponding exception 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 exception event e occurs, means (see Fig. 6):

Precondition: *state*(T) = `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 Fig. 6(b)),
6. Execute the basic operations associated with the *Skip* recovery transition to modify the net structure in order to handle the exception (see Fig. 6(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

Table 2 Task-based recovery policies

Recovery policy	Notation	Task status	Brief description
`Skip(Event e, Task T)`	S_T^e	Running	Skips the running task T to the immediate next task(s) if the event e occurs
`SkipTo(Event e, Task T, TaskSet `$\mathcal{T}$`)`	$ST_{T,\mathcal{T}}^e$	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_T^e	Completed	Removes the effect of an already executed task T if the event e occurs
`CompensateAfter(Event e, Task T)`	CA_T^e	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_T^e	Completed	Repeats the execution of a completed task T if the event e occurs
`RedoAfter(Event e, Task T)`	RA_T^e	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')`	$AT_{T,T'}^e$	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)`	$AP_{T,P}^e$	Running	Allows an alternative execution of a task T by another provider P if the event e occurs
`TimeOut(Task T, Time d)`	T_T^d	Running	Fails a task T if not completed within a time limit d. The execution is frozen

    (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,

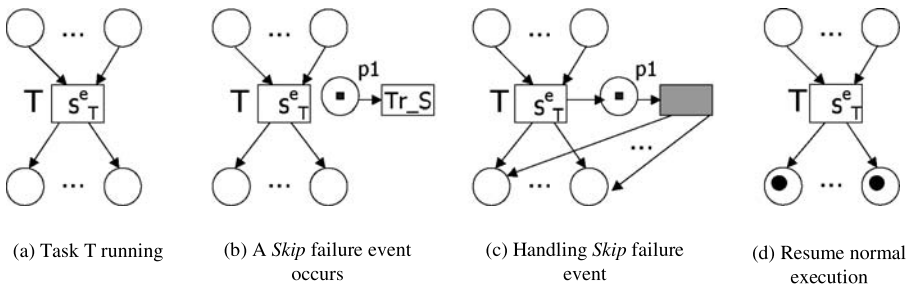7. Execute the added exceptional part of the SARN net,

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

Fig. 6 *Skip* recovery policy

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 exception 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 Fig. 6(d)).

For instance, in the *Travel Planning* BP example (see Fig. 1), we associate with the task *Hotel Booking* a *Skip* recovery policy at design time (see Fig. 7(a)). Upon the occurrence of a *Skip* exception event (for instance, "no response after one day") while executing the *Hotel Booking* task, the resulting SARN net will look like Fig. 7(b). After executing the set of basic operations corresponding to *Skip* recovery policy, the SARN net will become like Fig. 7(c). Once the *Skip* exception is handled, the SARN net will look like Fig. 7(d).

4.2 SkipTo

The *Skip* recovery policy defined previously is generic in the sense that there is no need to ask designers at which step of the execution they want to resume the execution from. Indeed, when skipping the running task, the immediate next task(s) with respect to the 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 exception event `e` occurs when executing a task `T1`, is defined as follows (see Fig. 8):

Precondition:

- $state(\texttt{T1}) = \texttt{Running}$,
- $\forall$ `T2, T2'` $\in \mathcal{T}$ `(T2, T2')` $\notin F^+ \wedge$ `(T2', T2)` $\notin F^+$, that is, the tasks of $\mathcal{T}$ are pairwise independent with respect to the flow relation F (see Definition 1). F^+ denotes the transitive closure of F, and
- $\forall$ `T2` $\in \mathcal{T}$ `(T1, 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.

(a) *Travel Planning* SARN at design time

(b) Occurrence of a *Skip* exception event "no response after one day" while executing *Hotel Booking*

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

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

Fig. 7 *Skip* recovery policy example

(a) Task T1 running Task T2 to skip to

(b) A *SkipTo* failure event occurs

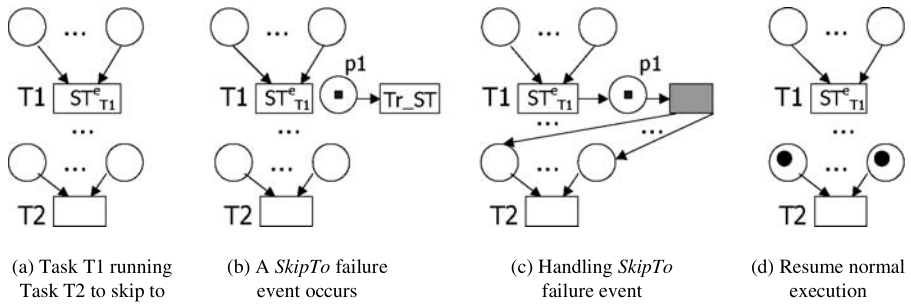(c) Handling *SkipTo* failure event

(d) Resume normal execution
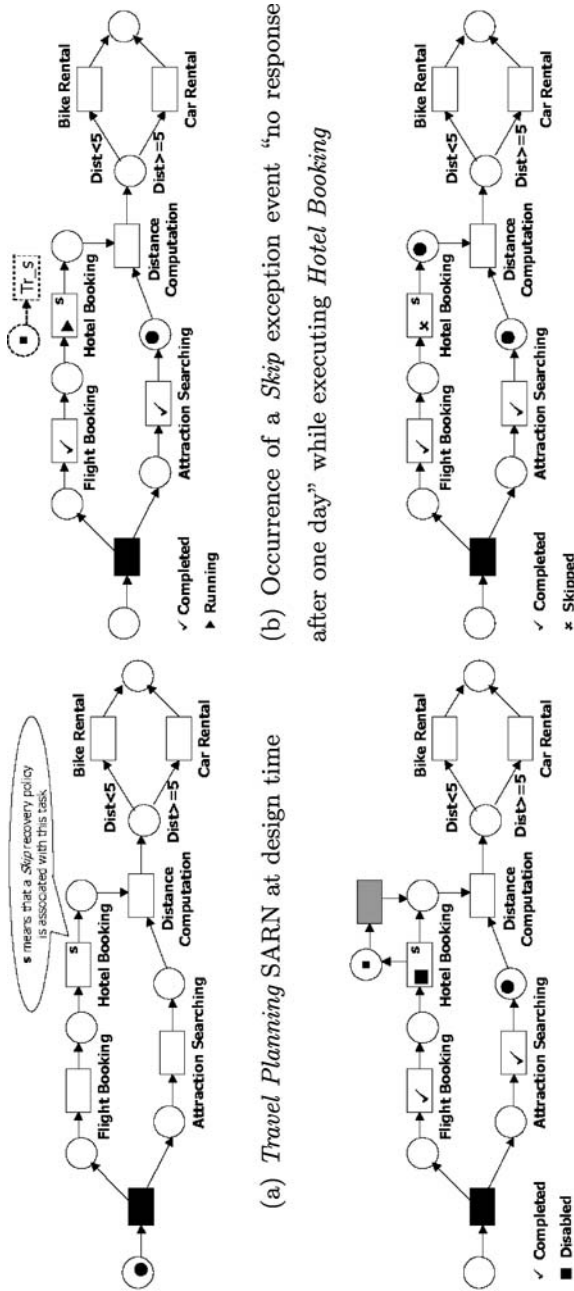
Fig. 8 *SkipTo* recovery policy

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 Fig. 8(b)),
6. Execute the elementary operations associated with the *SkipTo* recovery transition (see Fig. 8(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,
7. Execute the added exceptional part of the SARN net to handle the exception,
8. Remove the modifications made for the *SkipTo* exception 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 Fig. 8(d)).

Figure 9 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 (see Fig. 9(a)). When a *SkipTo* exception event, e.g., "unavailability of seats", occurs while executing the *Flight Booking* task, the resulting SARN net will look like Fig. 9(b). After executing the set of basic operations associated with *SkipTo* recovery policy, the SARN net will become like Fig. 9(c). Finally, once the *SkipTo* exception is handled, the *Travel Planner* BP will look like Fig. 9(d).

4.3 Compensate

The *Compensate* recovery policy removes all the effects of a completed task. The task must be compensable, i.e., there is a `compensate-T` task that removes the effect of

(a) *Travel Planning* SARN at design time

(b) Occurrence of a *SkipTo* exception event "unavailability of seats" while executing *Flight Booking*

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

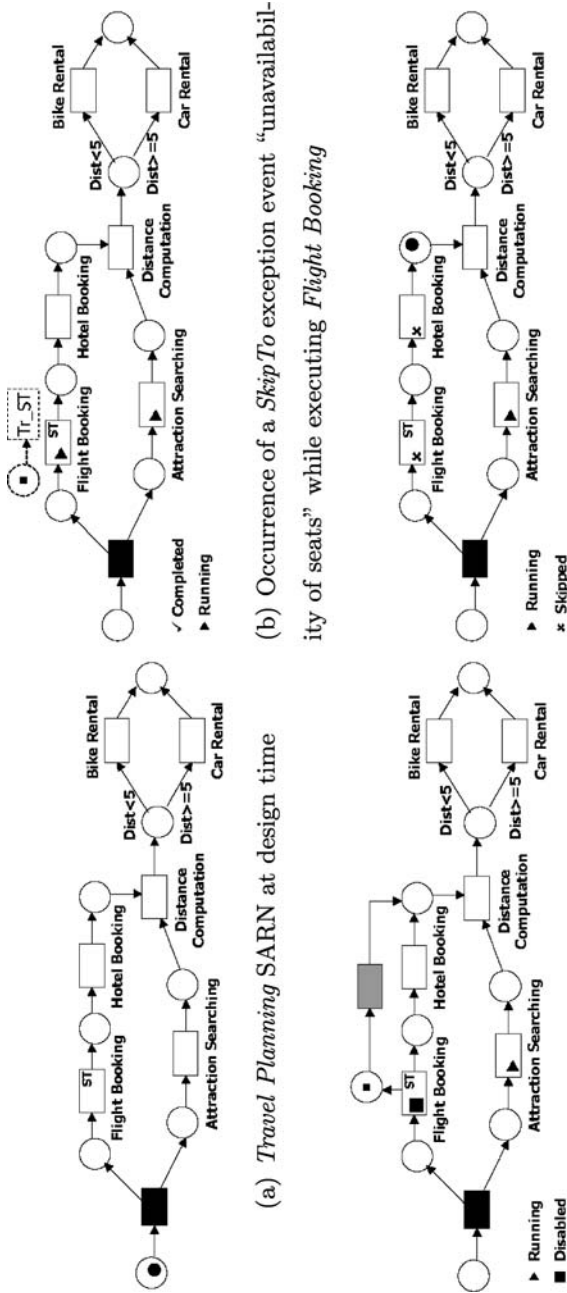(d) The SARN net just before resuming the normal execution

Fig. 9 *SkipTo* recovery policy example

the task T (see [2] for more details about compensable tasks). Note that the event of compensating a task can occur any time after the completion of the task and before the business process 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 exception event e occurs means:

Precondition:

- $state(\text{T}) = \text{Completed}$,
- T is *compensable*, i.e., there is a compensate-T task of T, and
- $\exists\, \text{t} \in T \mid state(\text{t}) = \text{Running}$.

Effect:

1. $\forall\, \text{t} \in T \mid (\text{T}, \text{t}) \in F^+ \wedge state(\text{t}) = \text{Running}$ *do* DisableTransition(t), so that $state(\text{t}) = \text{Frozen}$, 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 BP),
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:
   (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) $\forall\, \text{t} \in T \mid state(\text{t}) = \text{Frozen} \,\forall\, p \in {}^\bullet t$ *do* CreateArc(compensate-T, p): add an outgoing arc from the compensate-T transition to each input place of the suspended running task(s).
7. Execute the exceptional part of the SARN net,
8. Remove the modifications made for the task exception event, and
9. Resume the execution of the BP.

Figure 10 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 a *Compensate* recovery policy with the task *Flight Booking* (see Fig. 10(a)). When a *Compensate* exception event, for instance, "cancel flight", occurs while executing the task *Distance Computation*, the resulting SARN net will look like Fig. 10(b). After executing the set of basic operations associated with *Compensate* recovery policy, the SARN net will appear like Fig. 10(c). Finally, once the *Compensate* exception handled, the *Travel Planner* BP will look like in Fig. 10(d).

(a) *Travel Planning* SARN at design time

(b) Occurrence of a *Compensate* exception event "cancel flight" while executing *Distance Computation*

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

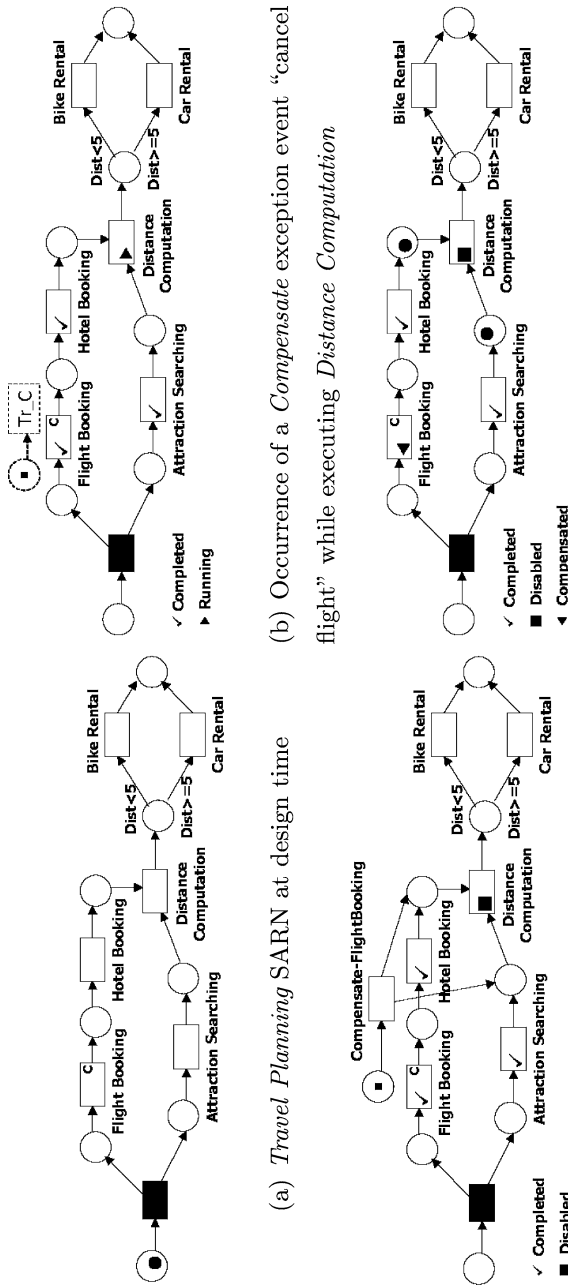(d) The SARN net just before resuming the normal execution

Fig. 10 *Compensate* recovery policy example

4.4 CompensateAfter

The *Compensate* recovery policy removes the effects of a completed task any time after the completion of the task and before the business process 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 exception event e occurs means:

Precondition:

- $state(\mathtt{T}) = \mathtt{Completed}$,
- T is *compensable*, i.e., there is a compensate-T task of T, and
- $\exists\, \mathtt{t} \in T \mid state(\mathtt{t}) = \mathtt{Running}$.

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:
   (a) ReplaceTransition(Tr_CA,compensate-T): associate to the *CompensateAfter* recovery transition the task compensate-T that removes the effects of the task T,
   (b) $\forall\, \mathtt{p} \in \mathtt{T}^{\bullet}$ CreateArc(compensate-T,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,
6. Execute the added exceptional part of the SARN net to handle the exception,
7. Remove the modifications made for the task exception event, and
8. Resume the execution of the BP.

In Fig. 11, 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 (see Fig. 11(a)) at built time. When a *CompensateAfter* exception event, e.g., "cancel attraction", occurs just after completing the execution of the task *Attraction Searching*, the resulting SARN net will look like Fig. 11(b). Once the set of basic operations associated with the *CompensateAfter* recovery policy are executed, the SARN net will look like in Fig. 11(c). Finally, after handling the *CompensateAfter* exception, the *Travel Planner* BP will appear like Fig. 11(d).
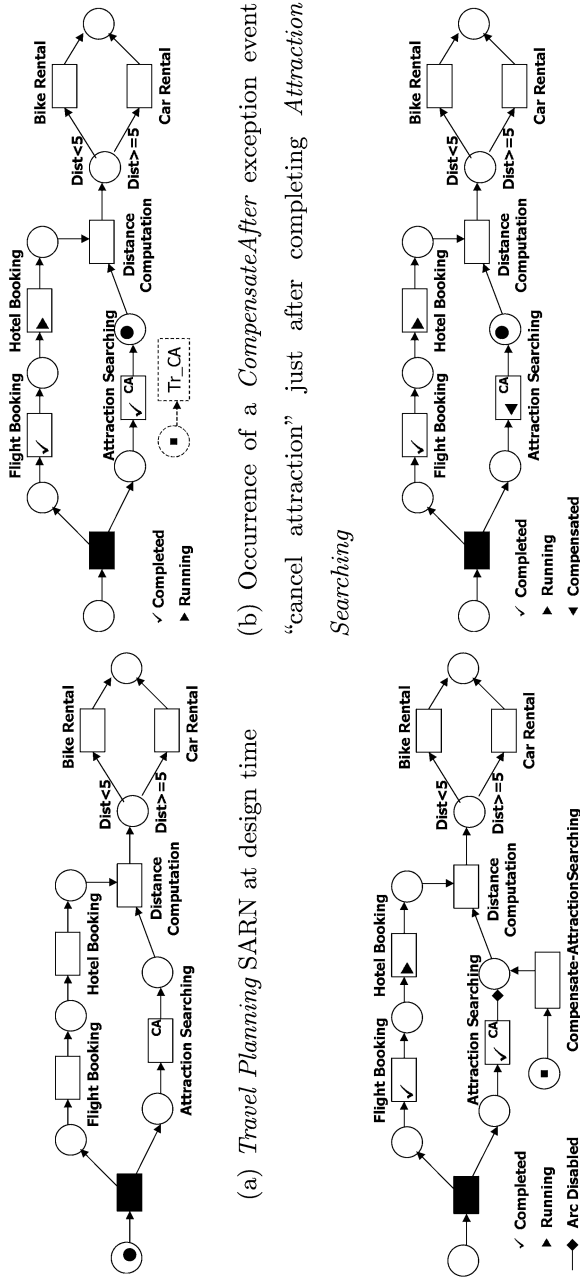
(a) *Travel Planning* SARN at design time

(b) Occurrence of a *CompensateAfter* exception event "cancel attraction" just after completing *Attraction Searching*

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

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

Fig. 11 *CompensateAfter* recovery policy example

4.5 Redo

The *Redo* recovery policy repeats (once) a finished task T if, for instance, one of its parameters needed to deliver (or produce) the results changes. 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 business process 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 exception event e occurs means:

Precondition:

- *state*(T) = Completed and
- $\exists$ t $\in T \mid state$(t) = Running.

Effect:

1. $\forall$ t $\in T \mid$ (T, t) $\in F^+ \wedge state$(t) = Running *do* DisableTransition(t), so that *state*(t) = 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:
   (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.
6. Execute the exceptional part of the SARN net,
7. Remove the modifications made for the task exception event, and
8. Resume the execution of the BP.

Figure 12 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 (see Fig. 12(a)). When a *Redo* exception event, e.g., "change flight date", occurs while executing the task *Distance Computation*, the resulting SARN net will look like Fig. 12(b). After executing the set of primitive operations associated with the *Redo* recovery policy, the SARN net will become like Fig. 12(c). Finally, once the *Redo* exception is handled, the *Travel Planner* BP will look like Fig. 12(d).
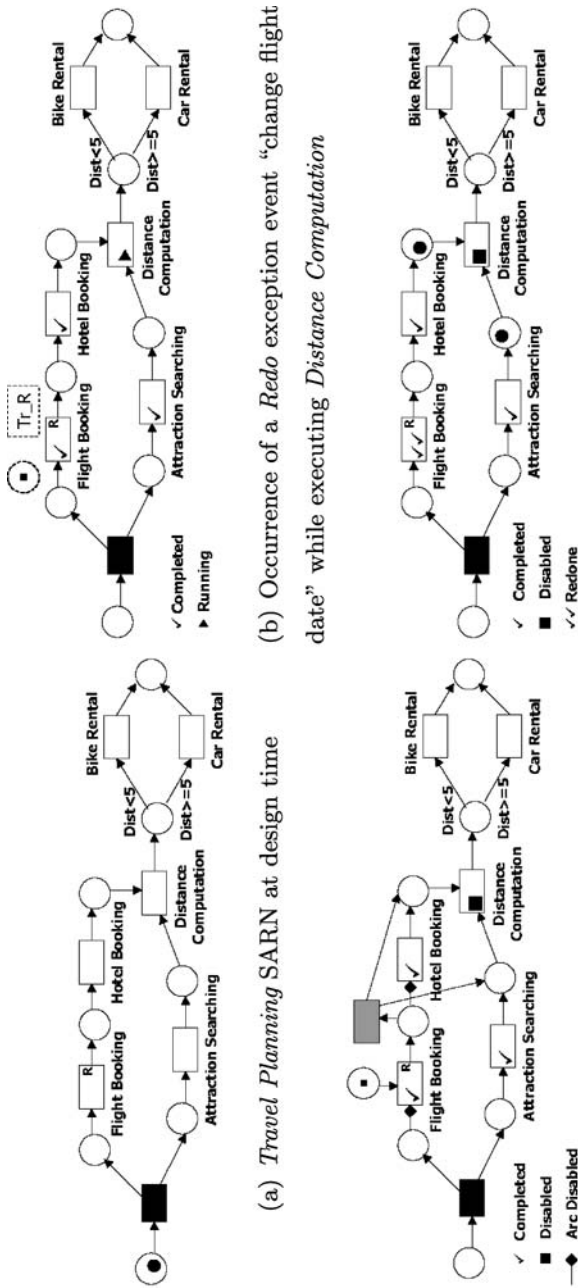
(a) *Travel Planning* SARN at design time

(b) Occurrence of a *Redo* exception event "change flight date" while executing *Distance Computation*

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

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

Fig. 12 *Redo* recovery policy example

4.6 RedoAfter

The *Redo* recovery policy repeats the execution of a completed task any time after the completion of the task and before the business process 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 exception event `e` occurs means:

Precondition:

- $state(T) = $ `Completed` and
- $\exists\, \text{t} \in T\,|\,state(\text{t}) = $ `Running`.

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:
   (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,
5. Execute the added exceptional part of the SARN net to handle the exception,
6. Remove the modifications made for the task exception event, and
7. Resume the execution of the BP.

In Fig. 13, 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 (see Fig. 13(a)) at built time. When a *RedoAfter* exception event "modify attraction time", for instance, occurs just after completing the execution of the task *Attraction Searching*, the resulting SARN net will look like Fig. 13(b). Once the set of basic operations associated with the *RedoAfter* recovery policy are executed, the SARN net will look like Fig. 13(c). Finally, after handling the *RedoAfter* exception, the *Travel Planner* BP will look like Fig. 13(d).

4.7 AlternativeTask

The *AlternativeTask* recovery policy allows another task `T'` to be executed in place of a running task `T` in case the latter fails. Formally, in the context of SARN, an `AlternativeTask(Event e, Task T, Task T')` recovery policy of a task

(a) *Travel Planning* SARN at design time
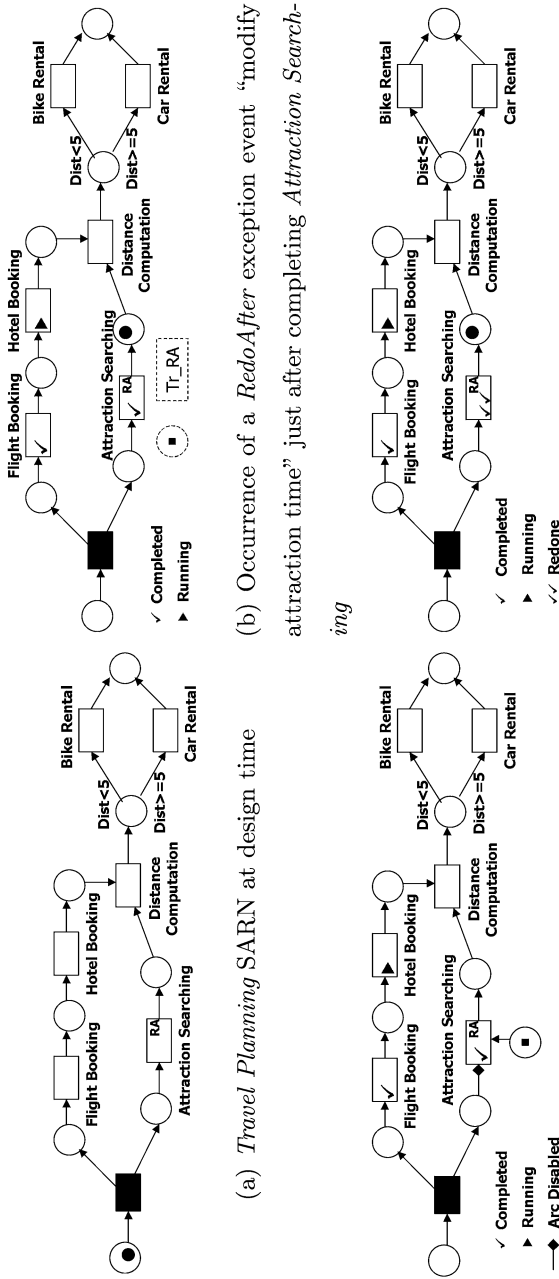
(b) Occurrence of a *RedoAfter* exception event "modify attraction time" just after completing *Attraction Search-ing*

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

(d) The SARN net just before resuming the normal exe-cution

Fig. 13 *RedoAfter* recovery policy example

(a) Task T running (b) An *AlternativeTask* failure event occurs (c) Handling *AlternativeTask* failure event (d) Resume normal execution
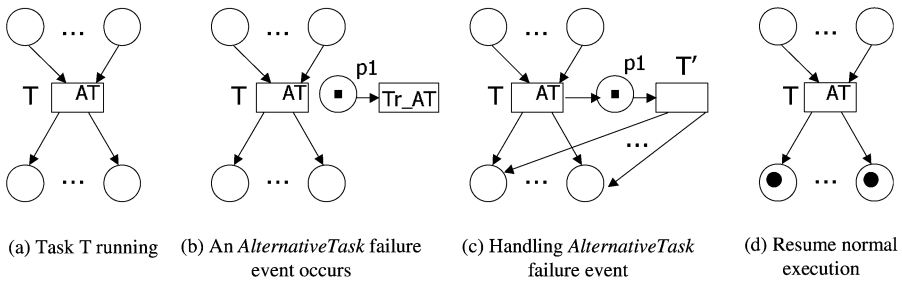
Fig. 14 *AlternativeTask* recovery policy

`T` by another task `T'` when its corresponding exception event `e` occurs means (see Fig. 14):

Precondition: *state*(`T`) = `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 Fig. 14(b)),
6. Execute the basic operations associated with the *AlternativeTask* recovery transition to modify the SARN structure (see Fig. 14(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,
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 Fig. 14(d)).

4.8 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 `Alternative-Provider(Event e, Task T, Provider P)` recovery policy of a task `T`

(a) Task T running   (b) An *AlternativeProvider* failure event occurs   (c) Handling *AlternativeProvider* failure event   (d) Resume normal execution
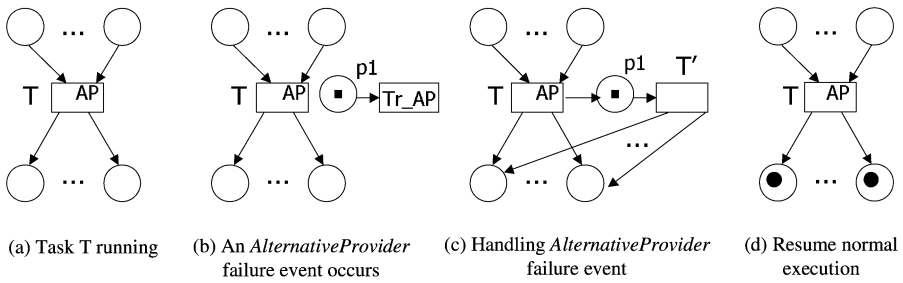
Fig. 15 *AlternativeProvider* recovery policy

by another provider P when its corresponding exception event e occurs means (see Fig. 15):

Precondition: *state*(T) = 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 Fig. 15(b)),
6. Modify the SARN structure by executing the basic operations associated with the *AlternativeProvider* recovery transition (see Fig. 15(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 *Alternative-Provider* recovery transition with the task T' of the alternative provider P, and
   (c) $\forall p \in T^{\bullet}$ `CreateArc(T',p)`: add an outgoing arc from T' to each output place of the replaced task,
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 Fig. 15(d)).

4.9 TimeOut

The *TimeOut* recovery policy allows a time limit d to be associated with a task. The task is failed after d units of time if it has not completed within that time. In terms of SARN model, a `TimeOut(Task T1, Time d)` recovery policy of a task T1

(a) Task T1 running   (b) A *Timeout* failure event occurs   (c) Handling *Timeout* failure event   (d) Freeze normal execution
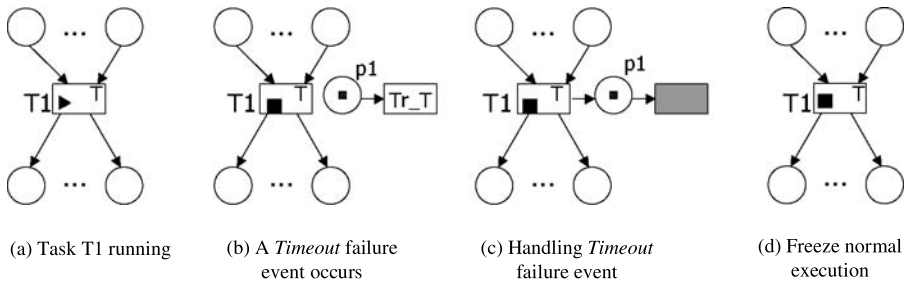
Fig. 16 *TimeOut* recovery policy

when its corresponding *TimeOut* exception event occurs after d units of time means (see Fig. 16):

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:
   (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,
7. Execute the added exceptional part of the SARN net,
8. Remove the modifications made, and
9. Freeze the normal execution of the BP.

5 Region-based recovery policies

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

5.1 Recovery region

A *recovery region* is a sub-business process, i.e., a set of related activities. These activities are grouped because they represent, for instance, the same unit of work

from the business perspective. A set of region-based recovery policies is assigned to a recovery region. As such, a recovery region is more than a traditional sub-business process. It is important to note that we do not require a recovery region to satisfy any of the transactional properties (e.g., atomicity). Furthermore, a recovery region must satisfy some structural constraints with respect to the underlying Petri net model described below. Formally, a *recovery region* is defined as follows.

Definition 5 (Recovery Region) Let $RN = (P, T, Tr, F, i, o, \ell, M)$ be a SARN net. A recovery region is a subnet $R = (P_R, T_R, Tr_R, F_R, i_R, o_R, \ell_R, S_R)$ 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 where $\mathcal{A}$ is a set of task names,
- 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 $T_R = \{t \in T \mid t^\bullet \cap P_R \neq \varnothing \ \wedge \ {}^\bullet t \cap P_R \neq \varnothing\}$. R must be connected (i.e., there are no 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. A recovery region is thus 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 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).

In this paper, we assume that recovery regions are correctly designed. One way of having a correct design of recovery regions is to use a top-down approach and region refinement operators we have defined in [11, 25].

5.2 Region-based 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 identify eight region-based recovery policies, namely, *SkipRegion*, *SkipRegionTo*, *CompensateRegion*, *CompensateRegionAfter*, *RedoRegion*, *RedoRegionAfter*, *AlternativeRegion*, and *TimeOutRegion*. Due to space limitation, we will only describe the *SkipRegion*, *CompensateRegion*, and *RedoRegion* recovery policies (refer to Table 3 and [25] for details about other region-based recovery policies).

Table 3 Region-based recovery policies

Recovery Policy	Notation	Region Status	Brief Description
SkipRegion(Event e, Region R)	SR_R^e	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 $\mathcal{T}$)	$SRT_{R,\mathcal{T}}^e$	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_R^e	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_R^e	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_R^e	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_R^e	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')	$AR_{R,R'}^e$	Running	Allows an alternative execution of a region R by another region R' if the event e occurs
TimeOutRegion(Region R, Time d)	TR_R^d	Running	Fails a region R if not completed within a time limit d. The execution is frozen

5.2.1 SkipRegion

The *SkipRegion* recovery policy will, when the corresponding exception 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 to the immediate next task(s) of R. This recovery policy applies to running recovery regions only, i.e., there
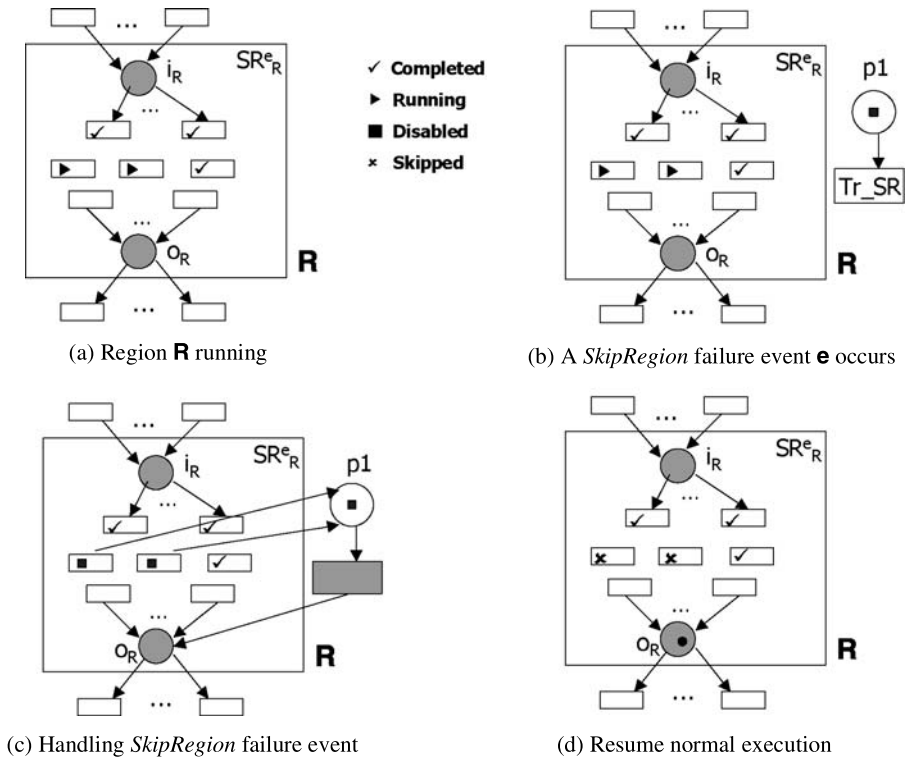
(a) Region **R** running

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

(c) Handling *SkipRegion* failure event

(d) Resume normal execution

Fig. 17 *SkipRegion* recovery policy

are tasks within the recovery region R that are still running. This means that, eventually, some tasks within the recovery region are completed, some are running, and others are not executed yet. Formally, a `SkipRegion(Event e, Region R)` recovery policy when executing tasks of the recovery region R and the corresponding exception event e occurs means (see Fig. 17):

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 Fig. 17(b)),
6. Execute the basic operations associated with the *SkipRegion* recovery transition to modify the net structure in order to handle the exception (see Fig. 17(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,

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 *SkipRegion* task exception 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 Fig. 17(d)).

5.2.2 *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 compensable, 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, a `CompensateRegion(Event e, Region R)` recovery policy of a recovery region R when its corresponding exception event `e` occurs means (see Figs. 18 and 19):

Precondition:

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

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, 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:
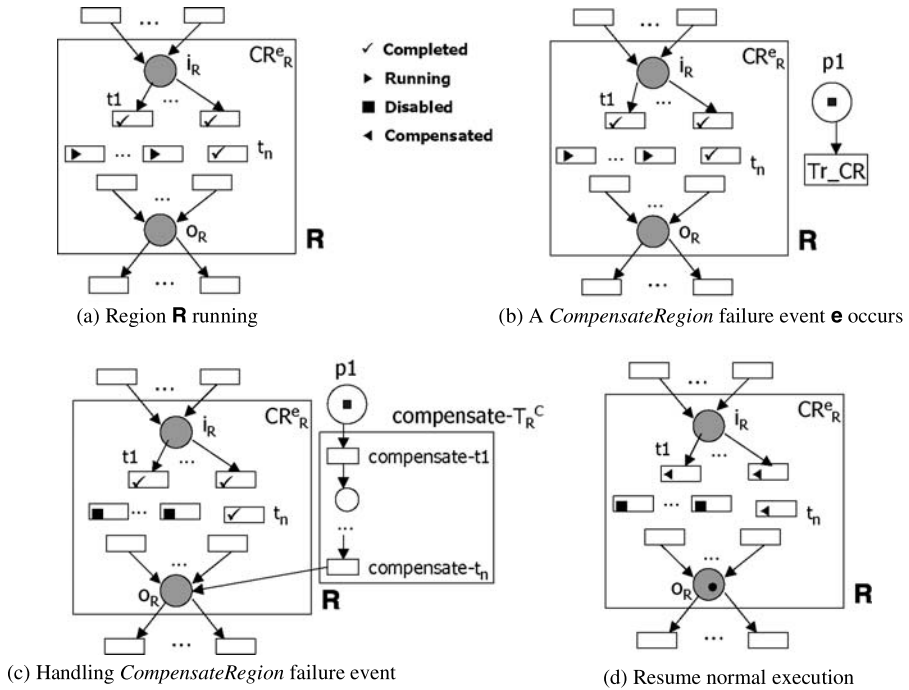
(a) Region **R** running

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

(c) Handling *CompensateRegion* failure event

(d) Resume normal execution

Fig. 18 *CompensateRegion* recovery policy of a running recovery region

(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 = \{t \in T_R \mid state(t) = $ `Completed`$\}$ that removes the effects of the already completed tasks T_R^C of the recovery region R and

(b) If $\exists\ t \in T_R \mid state(t) = $ `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 Fig. 18(c)). Otherwise, $\forall\ t \in T \mid state(t) = $ `Frozen` $\forall\ p\ \in\ {}^{\bullet}t\ do$ `Create-Arc(compensate` $-\ T_R^C, p)$, 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 BP (see Fig. 19(c)),

8. Execute the exceptional part of the SARN net,
9. Remove the modifications made for the *CompensateRegion* exception event, and
10. Resume the execution of the BP.

5.2.3 RedoRegion

The *RedoRegion* recovery policy repeats the execution of all already completed tasks of a (completed or running) recovery region. 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
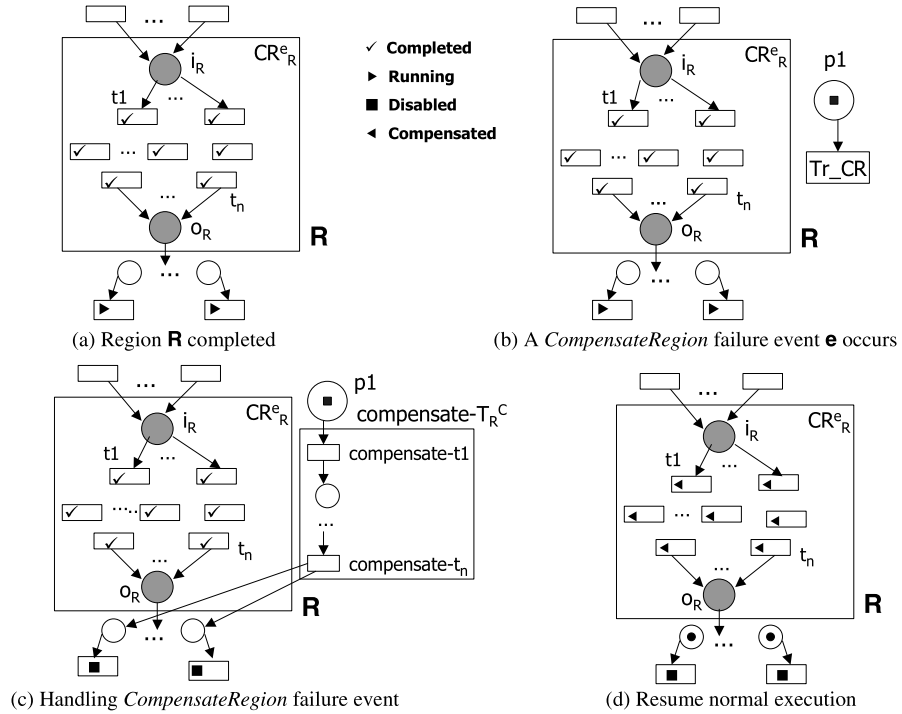
(a) Region **R** completed

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

(c) Handling *CompensateRegion* failure event

(d) Resume normal execution

Fig. 19 *CompensateRegion* recovery policy of a completed recovery region

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 R)` recovery policy of the recovery region R when its corresponding exception event e occurs means (see Fig. 20):

Precondition:

- $\exists \, T \in T_R \mid state(T) = $ `Completed` and
- $\exists \, t \in T \mid state(t) = $ `Running`.

Effect:

1. $\forall \, T \in T_R \mid state(T) = $ `Running` *do* `DisableTransition(T)`: possibly (in case of redoing a running recovery region) disable all running transitions of the recovery region R,
2. $\forall \, t \in T \mid (o_R, t) \in F^+ \ \wedge \ state(t) = $ `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,

(a) Region **R** running

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

(c) Handling *RedoRegion* failure event
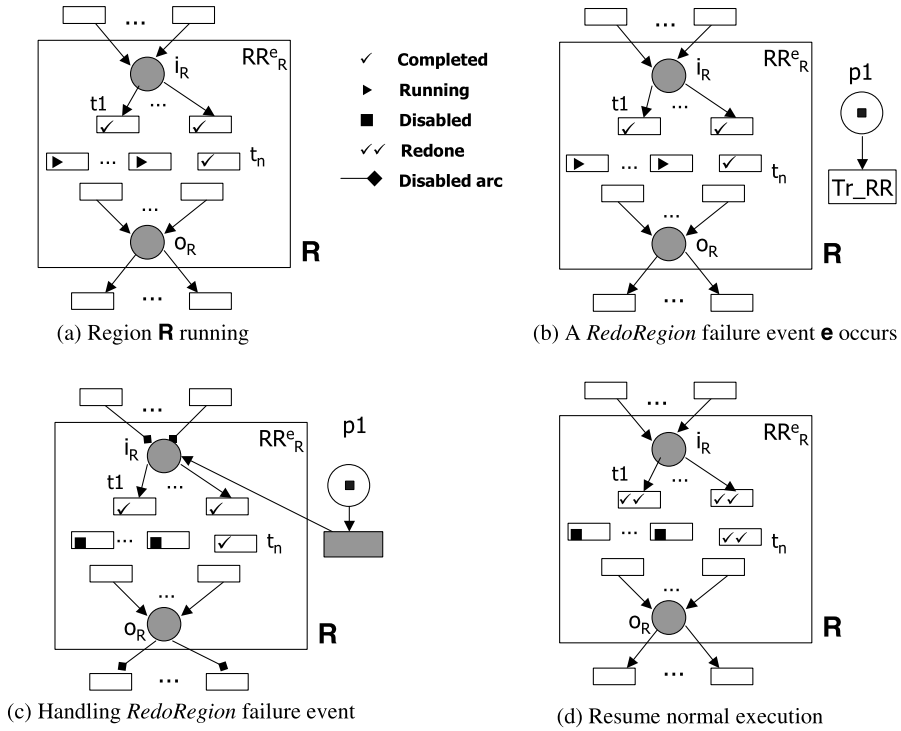
(d) Resume normal execution

Fig. 20 *RedoRegion* recovery policy

6. `AddRecoveryToken(p1)`: inject a recovery token into the input place of the *RedoRegion* recovery transition (see Fig. 20(b)),
7. Execute the elementary operations associated with the *RedoRegion* recovery transition (see Fig. 20(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,
8. Execute the added exceptional part of the SARN net,
9. Remove the modifications made for the *RedoRegion* exception 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 BP (see Fig. 20(d)).

5.3 Correctness preservation

Handling exceptions in BPs raises an important issue related to correctness preservation. Indeed, 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 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 Sect. 3.3 (i.e., reachability, liveness, and boundedness).

Proposition 1 (Correctness Preservation) *The SARN net $RN = (P, T, Tr, F, i, o, \ell, M)$ of a BP 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 Let us concentrate on the *Skip* and *SkipRegion* recovery policies since the proofs for other task- and region-based recovery policies follow the same line of reasoning. Let $RN = (P, T, Tr, F, i, o, \ell, M)$ (respectively $RN' = (P', T', Tr', F', i', o', \ell', M')$) be the SARN net of a BP before (respectively after) handling the exception using either *Skip* or *SkipRegion* recovery policy. Let us assume also that RN is a valid SARN net.

We start first with the *Skip* task-based recovery policy. Recall that a `Skip(Event e1, Task T1)` recovery policy skips the running task `T1` to the immediate next task(s) if the event `e1` occurs. To be able to handle this exception, the basic operations associated with the *Skip* recovery policy modify RN by: (a) adding a recovery place with a recovery token and an empty recovery transition, (b) adding an incoming arc from `T1` to the input place of the recovery transition, and (c) adding an outgoing arc from the recovery transition to each output place of `T1`. Since RN is valid and by definition of the *Skip* recovery policy, the system does not allow to skip to non-subsequent tasks (in the control flow), the obtained net RN' is valid, i.e., the reachability (each transition must be reachable from the initial marking), liveness (a final marking can be reached from every reachable marking), and boundedness (the number of tokens in each place is finite) properties are preserved.

Let us focus now on the *SkipRegion* recovery policy. Recall that `SkipRegion(Event e1, Region R1)` recovery policy skips the running task(s) of the region `R1` to the its immediate next task(s) if the event `e1` occurs. To handle this exception, the primitive operations associated with the *SkipRegion* recovery policy modify RN by: (a) adding a recovery place with a recovery token and an empty recovery transition, (b) adding an incoming arc from the running tasks of `R1` to the input place of the recovery transition, and (c) adding an outgoing arc from the recovery transition to the output place o_{R1} of `R1`. Since a recovery region contains one input place and one output place, when executing the tasks of the recovery region a token will be ultimately created in the output place of the region. It is natural then to do step (c) above which allows to skip to subsequent task(s) of the recovery region. Thus, the modified SARN net RN' is valid, i.e., preserves the reachability, liveness, and boundedness properties. □

6 SARN tool

This section provides an overview of the SARN tool we developed to illustrate the viability of the proposed exception handling technique. The tool has primarily been developed as a proof of concept and consequently the goal was not to implement a full-fledged business process management system. The tool implements regions and recovery policies for handling exceptions. It simulates the execution of BPs, the occurrence of exceptions, and their handling.

The architecture of the tool consists of two main components (see Fig. 21): (i) SARN Editor and (ii) SARN Simulator. In the following, we give a description of each component.

6.1 SARN editor

SARN Editor is composed of *Process Definition Editor* and *Exception Handling Editor*. *Process Definition Editor* is used by the business process modeler to design correct BPs.

The tool supports the creation of recovery policies for handling exceptions through the *Exception Handling Editor* component. These recovery policies are associated with either a task or a region. The business process modeler specifies how to recover from a particular exception by using predefined common recovery policies. A more important feature is that the tool lets the modeler define new and customized recovery

Fig. 21 SARN architecture
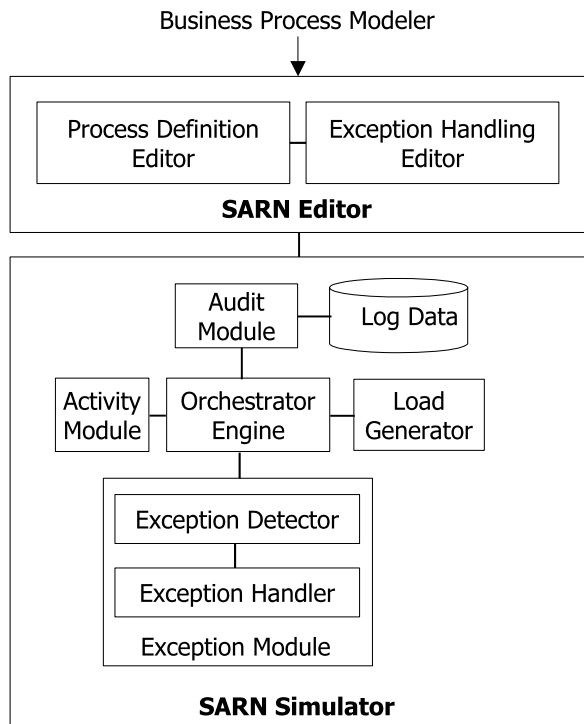
policies by combining several primitive operations, such as adding a transition and disabling a transition (see Table 1).

Exception Handling Editor supports the addition of recovery policies to tasks. The tool offers common task recovery policies such as *Skip* and *Compensate* as defined previously. A task recovery policy has a condition attribute which specifies the event condition of the corresponding exception.

The tool also supports the addition of region recovery policies to regions through the *Exception Handling Editor* component. A region recovery policy has a condition attribute the same as for a task recovery policy, which specifies the event condition of the corresponding exception. When a new region recovery policy is designed, a new recovery transition is added.

6.2 SARN simulator

SARN Simulator simulates the execution of a BP, the occurrence of exceptions, and their handling through recovery policies. It consists of five distinct components: *Orchestrator Engine*, *Load Generator*, *Activity Module*, *Exception Module*, and *Audit Module*. The *Orchestrator Engine* component is responsible for the interpretation of the business process specification and the execution of the business process instances. It interacts with *Load Generator* and *Activity Module*. The *Load Generator* component starts new business process instances. The starting rate is a parameter of the simulation. The *Activity Module* component simulates the business process activities. It receives the input data from the *Orchestrator Engine*, awaits for some time that corresponds to the duration of the activity, and delivers the results back to the *Orchestrator Engine*. *Exception Module* simulates the occurrence of exceptions. It includes the *Exception Detector* component that detects the occurrence of exception events and the *Exception Handler* component that recovers from exceptions. Finally, the *Audit Module* part logs information about the execution of the activities of the BP such as start and end times of activities, exception events, and recovery procedures.

SARN Editor and SARN Simulator understand a common XML schema which provides a framework of communication. The XML format follows the Petri Net Markup Language (PNML) standards [3]. The simulator takes the XML generated by the editor as input to perform the simulation.

It is important to note that SARN can accept any BP specified in Business Process Execution Language for Web Services (BPEL4WS) [12] since there are now tools, such as BPEL2PN [43] and BPEL2PNML [38], that transform a process specified in BPEL4WS into PNML, the standard interchange format for Petri nets. If a language such as BPEL4WS is used, then tools such as BPWS4J, ActiveBPEL, and Oracle BPEL Process Manager can be adopted for designing the BPEL4WS process. In this case, each primitive activity (e.g., invoke, receive, reply) in the BPEL4WS specification may be perceived as a task in the business process.

The validation (beyond a proof of concept) of the proposed approach is an important pre-requisite to the widespread adoption of SARN. We identify three major validation factors: productivity, ease of use, and scalability. Several definitions of software productivity have been given in the literature such as the ratio between the amount of software produced to the labor (e.g., measured in the total man-hours

expended during development and integration) and expenses of producing it. Ease-of-use refers to the property that BPs can be designed in SARN without having to overcome a steep learning curve. SARN should be proven to be intuitive for non-expert designers. One way to assess the ease-of-use is to measure the amount of time required for a non-expert designer to acquire an average knowledge of SARN tool. Scalability measures the ability of SARN to perform well for large BPs, both at design time and run time.

7 Related work

Some studies have considered the problem of exception handling and recovery from task failures in workflow management systems. In addition, a significant amount of work show how some of the concepts used in transaction management can be applied to business process environments.

7.1 Workflow change management

Ellis et al. proposed a workflow evolution model which uses a meta-model to design workflows as special kinds of Petri nets [16]. The meta-model covers the structural and behavioral aspects of workflows. A workflow schema can be modified by replacing a subnet of an existing net by another one. But no concrete modification operations are provided. A correction criterion, defining valid workflow execution sequences, specifies which modifications are correct. In particular, a modification of a workflow schema is only considered to be correct, if the instances of the workflow schema are correct executions of the modified schema.

Leymann 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 [34]. The fact that spheres do not have to be a connected graph leads to very complicated semantics.

The workflow evolution approach proposed by Casati et al. provides a set of modifications operations that allow to change a workflow schema and to migrate existing workflow instances to the modified schema [7]. The meta-model supports the functional, structural, and behavioral aspects of workflows. A complete set of model modification operations is supported. The notion of compliance of a workflow instance to a workflow schema is introduced, where a workflow instance i is compliant to a workflow schema s if i has been executed according to s. In case i complies to s, i can be migrated to s.

The approach for dynamic workflow model evolution proposed by Joeris and Herzog is based on the explicit versioning of workflow schemas [30]. The meta-model supports the functional, structural, and behavioral aspects of workflows. Model modification operations are provided. However, these operations are not described in detail. In addition, no correctness criteria for the model or for the workflow instances are defined.

Based on a conceptual graph-based workflow model, called ADEPT, which has a formal foundation and an operational semantics, Reichart and Dadam developed a

set of change operations, namely ADEPT$_{flex}$ [41]. This framework supports structural changes of workflow instances. Schema evolution is not considered. Some modification operations are described that can be applied to workflows. The modification of a workflow is allowed, if certain correctness criteria hold which consider the structure as well as the state of the workflow at modification time. In this approach, workflow instances do not have to be in strict accordance with their schema, since workflow instances can be modified without their schema being modified as well. ADEPT$_{flex}$ focuses on providing a minimal and correct set of change primitives rather than on managing complex workflow changes and migrations.

Klingemann developed an approach for incorporating additional constraints into workflow specification to adapt the workflow execution for an optimized goal fulfillment [32]. To make the workflow flexible, the author integrated execution alternatives in the form of flexible elements into the workflow structure. This flexibility can then be used for a goal-driven workflow execution and to react actively to changes in the runtime conditions.

WIDE introduces a trigger-based approach for exception handling [6]. Exception handlers can be predefined to handle events such as the cancellation of a task or the break of the normal flow. For each type of exception, WIDE provides a default exception handler (e.g., for user notification), which may be overwritten by the workflow administrator. However, it is the responsibility of administrators to avoid inconsistencies and errors, which greatly complicates application development and may introduce new errors and exceptions into the workflow model.

7.2 Transactions in business processes

Since BPs contain activities that access shared and persistent data resources, they have to be subject to transactional semantics [14, 17, 29]. However, it is not adequate to treat an entire BP as a single ACID transaction mainly since BPs: (i) are of long duration and treating an entire process as a transaction would require locking resources for long periods of time, (ii) involve many independent database and application systems and enforcing ACID properties across the entire process would require expensive coordination among these systems, and (iii) have external effects and guaranteeing atomicity using conventional transactional rollback mechanisms is not feasible. Several models for long-running transactions have then been developed to allow the definition of ACID-like properties at the business process level and to handle activity failures [17, 29].

The earliest of the long-running transaction models was the Saga model [18]. A Saga is a chain of transactions that is itself atomic. Each transaction in the chain is assumed to have a semantic inverse, or compensation, transaction associated with it. If one of the transactions in the Saga fails, the transactions are rolled back in the reverse order of their execution. Committed transactions are rolled back by executing their corresponding compensation transactions.

In the Activity Transaction Model (ATM), long-running transactions are allowed to be both nested and chained [13]. Nesting allows concurrency within a transaction and also provides fine-grained and hierarchical control for failure and exception handling. The original nested transaction model of [36] which supports only closed

sub-transactions, was extended to include also open sub-transactions [45]. A closed sub-transaction commits its results to its parent. These partial results are externalized only after the top (root) transaction commits, thus ensuring atomicity and isolation of the whole transaction. In contrast, open sub-transactions sacrifice isolation by directly externalizing their results. In ATM, failure handling is hierarchical. When a sub-transaction fails, its parent is notified, and the parent can decide to execute an exception handler and retry the failed child, execute an alternate (contingency) task, or propagate the failure up the hierarchy. Propagating failures requires compensating already committed sub-transactions. Some tasks can be defined as vital, meaning that their failure causes the failure of the transaction hierarchy.

Subsequent work extended the ATM model to allow sub-transactions whose commit scopes were in between the two extremes of closed and open nested transactions [8, 9]. Failure handling was hierarchical, i.e., the highest ancestor that needed to be aborted was identified, and then the sub-tree rooted at this ancestor was compensated or aborted. The model allowed compensation and contingencies to be associated with different levels of the hierarchy. Thus, sometimes it might be preferable to compensate an entire sub-tree instead of compensating every sub-transaction in it. A further extension of this model, in [9, 10], applied to the case of propagating failures from one transaction hierarchy to another.

Similar ideas to those in ATM also exist in other transactional workflow models. For example, ConTracts [42, 44] provide an execution and failure model for long-lived transactions and workflow applications. A ConTract is a long-lived transaction composed of steps, whose order of execution is specified by a script. Isolation between steps is relaxed, so that the results of completed steps are visible to other steps. To guarantee semantic atomicity, each step is associated with a compensating step that semantically undoes the effect of the step. ConTracts provide both forward and backward recovery to manage failures. Backward recovery is achieved by compensating completed steps, typically in the reverse order of their execution. Compensation may be partial, meaning that it is performed up to a point in the contract from where forward execution can be resumed, possibly along a path that is different from the faulty one.

The basic ideas of: (i) transactional grouping of parts of a BP, (ii) attaching compensation and contingency activities to the activities of the BP, (iii) declaring some of the activities to be critical, and (iv) defining points in the process up to which rollback occurs in case of failure followed by forward execution, spread many of the transactional models that were subsequently proposed, e.g., WAMO [15], WIDE [23], and CREW [31]. These models typically differ in how much flexibility the business process designer has in specifying the backward compensation and forward execution process.

The Exotica project describes methods and tools to implement advanced transaction models on top of Flowmark (predecessor of IBM MQ Series Workflow) [1]. The basic idea is to provide the user with an extended workflow model that integrates advanced transaction concepts. The user could define a compensating task for each task of the workflow. A preprocessor will then translate these specifications into plain FDL (Flowmark Definition Language) by properly inserting additional compensating paths after each task or group of tasks, which are conditionally executed upon a task

failure. In particular, it is shown how Sagas and flexible transactions could be implemented in Flowmark.

Godart et al. developed the COO cooperative transaction protocol to allow exchange of intermediate results during cooperative transaction execution [22]. This relaxes the isolation property of traditional transaction protocol but this is considered as fundamental for long duration transactions such as BPs. Relaxing isolation induces the risk of inconsistency due to dirty read or lost update and the COO protocol provides means to avoid this risk. It obliges users to compensate dirty read before terminating their task.

Krishnamoorthy and Shan presented a transactional model for HP Changengine (later called Process Manager) [33]. The model allows the definition of Virtual Transaction (VT) regions on top of a workflow graph. If a failure occurs during the execution of a task enclosed in a VT region, then all tasks in the region are compensated in the reverse order of their forward execution, until a compensation end point is reached. Then, the system can retry the execution (up to a maximum number of times), follow an alternate path, or terminate the entire process execution. The virtual transaction model also allows for different isolation levels for VT regions: (i) serializable (needs shared locks for reads and exclusive locks for writes), (ii) read committed (like serializable, but releases shared locks after reading), (iii) read uncommitted (no locks needed for reads), and (iv) virtual isolation (get read locks and release after reading, and get write locks only at the end of the transaction to perform all the updates in one shot).

Standards bodies and industry consortia are also engaged in efforts to define transaction models at the business process level, both for processes within an organization and for inter-organization processes. In particular, OASIS has formed a Business Transaction Technical Committee (BTTC), with the goal of defining a transaction protocol for BPs that span across organizational boundaries [37]. While the proposals introduced above aim at defining transactional semantics for BPs, BTTC aims at defining transactional semantics for B2B protocols, such as the RosettaNet standards. BTTC assumes that each party involved in a multi-party B2B interaction is responsible for supporting transactional properties for the internal BPs, and instead defines a coordination protocol to ensure that all or none of the involved parties "commit" the effects of the B2B interaction. This problem is similar to that of coordinating distributed transactions in database systems, although carried over to the context of BPs and long-running transactions.

WS-Transaction and WS-Coordination specifications support transactional coordination of Web services. WS-Coordination [5] defines a generic coordination framework that can support various coordination protocols. Each protocol is intended to coordinate a different role that a Web service plays in the activity. A *Coordination Service* propagates and coordinates activities between services. The messages exchanged between participants carry a *Coordination Context* that contains critical information for linking the various activities within the protocol. A *Coordination Service* consists of several components: an *Activation Service* that allows a *Coordination Context* to be created, a *Registration Service* that allows a Web service to register its participation in a *Coordination Protocol*, and a set of *Coordination Protocol Services* for each supported *Coordination Type* (e.g., Completion, 2PC). WS-Transaction [4] specifies

transactional properties of Web services independently of coordination aspects. It uses two completion patterns: (i) *Atomic Transaction* (AT) and (ii) *Business Activity* (BA). An *Atomic Transaction* is used to coordinate activities having a short duration and executed within limited trust. It has the classical atomicity property ("all or nothing" behavior). A *Business Activity* provides flexible transaction properties (relaxing Isolation and Atomicity) and is used to coordinate activities that have long running duration. Actions are applied immediately and are permanent. This is because the long duration nature of the activities prohibits locking of data resources.

In general, transactional approaches offer extreme and expensive solutions in terms of lost work, and therefore some task failures may deserve an ad hoc handling. In this case, task failures are in fact handled as expected exceptions, and can be modeled by applying the directives and techniques we described in this paper.

8 Conclusions

In this paper, we proposed the Self-Adapting Recovery Net (SARN) model for specifying exceptional behavior in business processes at design time. We also identified a set of high-level recovery policies that are incorporated with a single task and a recovery region. It is important to mention that, operationally, each region-based recovery policy is a succession of its corresponding task-based recovery policy. 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 pre-specified exception events while keeping the underlying Petri net design simple and easy. For existing models to realize the same functionality the design effort could be significant. Most importantly, the correctness of the modified SARN w.r.t. consistency constraints (e.g., reachability, liveness, and boundedness) is preserved. A tool has been developed to illustrate the feasibility of SARN.

References

1. Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Günthör, R., Mohan, C.: 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, Los Alamitos (1996)
2. Benatallah, B., Casati, F., Toumani, F., Hamadi, R.: Conceptual modeling of web service conversations. In: Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), Klagenfurt, Austria, June 2003. LNCS, vol. 2681, pp. 449–467. Springer, Berlin (2003)
3. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri net markup language: concepts, technology, and tools. In: van der Aalst, W., Best, E. (eds.) Proceedings of the 24th International Conference on Application and Theory of Petri Nets (ICATPN'03), Eindhoven, The Netherlands, June 2003. LNCS, vol. 2679, pp. 483–505. Springer, Berlin (2003)
4. Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., Thatte, S.: Web services transaction (WS-transaction). http://dev2dev.bea.com/techtrack/ws-transaction.jsp, August 2002
5. Cabrera, F., Copeland, G., Freund, T., Klein, J., Langworthy, D., Orchard, D., Shewchuk, J., Storey, T.: Web services coordination (WS-coordination). http://www-106.ibm.com/developerworks/library/ws-coor, August 2002

6. Casati, F., Grefen, P., Pernici, B., Pozzi, G., Sanchez, G.: WIDE workflow model and architecture. Technical Report 96-16, Centre for Telematics and Information Technology (CTIT), University of Twente, The Netherlands (1996)
7. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. Data Knowl. Eng. **24**(3), 211–238 (1998)
8. Chen, Q., Dayal, U.: A transactional nested process management system. In: Proceedings of the 12th International Conference on Data Engineering (ICDE03), New Orleans, USA, February 1996. IEEE Computer Society, Los Alamitos (1996)
9. Chen, Q., Dayal, U.: Failure handling for transaction hierarchies. In: Proceedings of the 12th International Conference on Data Engineering (ICDE'96), Birmingham, UK, April 1997. IEEE Computer Society, Los Alamitos (1997)
10. Chen, Q., Dayal, U.: Multi-agent cooperative transactions for E-commerce. In: Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS'00), Eilat, Israel, September 2000. LNCS, vol. 1901. Springer, Berlin (2000)
11. Chrzastowski-Wachtel, P., Benatallah, B., Hamadi, R., O'Dell, M., Susanto, A.: A top-down Petri net-based approach for dynamic workflow modeling. In: Proceedings of the International Conference on Business Process Management (BPM'03), Eindhoven, The Netherlands, June 2003. LNCS, vol. 2678, pp. 336–353. Springer, Berlin (2003)
12. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business process execution language for web services (BPEL4WS). http://dev2dev.bea.com/techtrack/BPEL4WS.jsp, August 2002
13. Dayal, U., Hsu, M., Ladin, R.: A transactional model for long-running activities. In: Proceedings of the 17th Very Large Data Base Conference (VLDB'91), Barcelona, Spain, September 1991
14. Dayal, U., Hsu, M., Ladin, R.: Business process coordination: state of the art, trends, and open issues. In: Proceedings of the 27th Very Large Data Base Conference (VLDB'01), Rome, Italy, September 2001
15. Eder, J., Liebhart, W.: The workflow activity model WAMO. In: Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS'95), Vienna, Austria, May 1995, pp. 87–98
16. Ellis, C.A., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of the Conference on Organizational Computing Systems (COOCS'95), Milpitas, USA, August 1995, pp. 10–21. ACM Press, New York (1995)
17. Elmagarmid, A.K.: Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Mateo (1992)
18. Garcia-Molina, H., Salem, K.: Sagas. In: Proceedings of the ACM SIGMOD, San Francisco, USA, 1987
19. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput. **37**(10), 46–54 (2004)
20. Georgakopoulos, D., Hornick, M., Sheth, A.: An overview of workflow management: from process modeling to workflow automation infrastructure. Distrib. Parallel Databases 3(2), 1995
21. Georgakopoulos, D., Schuster, H., Cichocki, A., Baker, D.: Managing process and service fusion in virtual enterprises. Inf. Syst. Spec. Issue Inf. Syst. Support Electron. Commer. **24**(6), 429–456 (1999)
22. Godart, C., Canals, G., Charoy, F., Molli, P., Skaf, H.: Designing and implementing COO: design process, architectural style, lessons learned. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 342–352. IEEE Computer Society, Los Alamitos (1996)
23. Grefen, P., Vonk, J., Boertjes, E., Apers, P.: Two-layer transaction management for workflow management applications. In: Proceedings of the 8th International Conference on Database and Expert Systems Applications (DEXA'97), Toulouse, France, September 1997
24. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Trans. Softw. Eng. (TSE) **26**(10), 943–958 (2000)
25. Hamadi, R.: Formal composition and recovery policies in service-based business processes. PhD thesis, The University of New South Wales, Sydney, Australia (2005)
26. Hamadi, R., Benatallah, B.: Recovery nets: towards self-adaptive workflow systems. In: Proceedings of the 5th International Conference on Web Information Systems Engineering (WISE'04), Brisbane, Australia, November 2004. LNCS, vol. 3306, pp. 439–453. Springer, Berlin (2004)
27. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
28. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4), 293–333 (1996)

29. Jajodia, S., Kerschberg, L.: Advanced Transaction Models and Architectures. Kluwer Academic, Dordrecht (1997)
30. Joeris, G., Herzog, O.: Managing evolving workflow specifications. In: Proceedings of the 3rd Conference on Cooperative Information Systems (CoopIS'98), New York, USA, August 1998
31. Kamath, M., Ramamritham, K.: Failure handling and coordinated execution of concurrent workflows. In: Proceedings of the 14th International Conference on Data Engineering (ICDE'98), Florida, USA, February 1998. IEEE Computer Society, Los Alamitos (1998)
32. Klingemann, J.: Controlled flexibility in workflow management. In: Proceedings of the 12th Conference on Advanced Information Systems Engineering (CAiSE'00), Stockholm, Sweden, June 2000
33. Krishnamoorthy, V., Shan, M.-C.: Virtual transaction model to support workflow applications. In: Proceedings of the 2000 ACM Symposium on Applied Computing (SAC'00), Como, Italy, March 2000. IEEE Computer Society, Los Alamitos (2000)
34. Leymann, F.: Supporting business transactions via partial backward recovery in workflow management systems. In: Datenbanksysteme in Buro, Technik und Wissenschaft, pp. 51–70 (1995)
35. Medjahed, B., Benatallah, B., Bouguettaya, A., Ngu, A.H.H., Elmagarmid, A.K.: Business-to-business interactions: issues and enabling technologies. VLDB J. 12(1), 59–85 (2003)
36. Moss, J.E.B.: Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, Cambridge (1985)
37. OASIS Committee Specification. Business Transaction Protocol, version 1.0 (June 2002)
38. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Wof-BPEL: a tool for automated analysis of BPEL processes. In: Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC'05), Berlin, Germany, December 2005. LNCS, vol. 3826, pp. 484–489. Springer, Berlin (2005)
39. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall, Englewood Cliffs (1981)
40. Petri, C.A.: Kommunikation mit automaten. PhD thesis, University of Bonn, Germany (1962) (in German)
41. Reichert, M., Dadam, P.: ADEPT flex: supporting dynamic changes of workflows without losing control. J. Intell. Inf. Syst. 10(2), 93–129 (1998)
42. Reuter, A., Schneider, K., Schwenkreis, F.: ConTracts revisited. In: Jajodia, S., Kerschberg, L. (eds.) Advanced Transaction Models and Architectures. Kluwer Academic, Dordrecht (1997)
43. Stahl, C., Hinz, S., Schmidt, K.: Transforming BPEL to Petri nets. In: Proceedings of the Third International Conference on Business Process Management (BPM'05), Nancy, France, September 2005. LNCS, vol. 3649, pp. 220–235. Springer, Berlin (2005)
44. Wächter, H., Reuter, A.: The ConTract model. In: Elmagarmid, A.K. (ed.) Database Transaction Models for Advanced Applications, pp. 219–264. Morgan Kaufmann, San Mateo (1992)
45. Weikum, G., Schek, H.J.: Concepts and applications of multilevel transactions and open nested transactions. In: Elmagarmid, A.K. (ed.) Database Transaction Models for Advanced Applications. Morgan Kaufmann, San Mateo (1992)
46. WfMC. Workflow management coalition, terminology and glossary. Document Number WFMC-TC-1011, February 1999. http://www.wfmc.org/standards/docs.htm/