Formal Semantics and Analysis of Control Flow in WS-BPEL[⋆]

Chun Ouyang[1], Wil M.P. van der Aalst[2,1], Stephen Breutel[1], Marlon Dumas[1], Arthur H.M. ter Hofstede[1], and Eric Verbeek[2]

[1] Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane QLD 4001, Australia
{c.ouyang,sw.breutel,m.dumas,a.terhofstede}@qut.edu.au

[2] Department of Technology Management, Eindhoven University of Technology, GPO Box 513, NL-5600 MB, The Netherlands
{w.m.p.v.d.aalst,h.m.w.verbeek}@tm.tue.nl

Abstract. Web service composition refers to the creation of new (Web) services by combination of functionality provided by existing ones. This paradigm has gained significant attention in the Web services community and is seen as a pillar for building service-oriented applications. A number of domain-specific languages for service composition have been proposed with consensus being formed around a process-oriented language known as WS-BPEL (or BPEL). The kernel of BPEL consists of simple communication primitives that may be combined using control-flow constructs expressing sequence, branching, parallelism, synchronisation, etc. As a result, BPEL process definitions lend themselves to static flow-based analysis techniques. In this report, we describe a tool that performs two useful types of static checks and extracts meta-data to optimise dynamic resource management. The tool operates by translating BPEL processes into Petri nets and exploiting existing Petri net analysis techniques. It relies on a comprehensive and rigorously defined mapping of BPEL constructs into Petri net structures.

Keywords: Business process modelling, Web services, BPEL, tool-based verification, Petri nets.

1 Introduction

There is an increasing acceptance of Service-Oriented Architectures (SOA) as a paradigm for integrating software applications within and across organisational boundaries. In this paradigm, independently developed and operated applications are exposed as (Web) services that communicate with each other using XML-based standards, most notably SOAP and associated specifications [3]. While the technology for developing basic services and interconnecting them on a point-to-point basis has attained a certain level of maturity, there remain open challenges when it comes to engineering services that engage in complex interactions with multiple other services.

A number of approaches have been proposed to address these challenges. One such approach, known as (process-oriented) service composition [6] has its roots in workflow and business process management. The idea of service composition is to capture the business logic and behavioural interface of services in terms of process models. These models may be expressed at different levels of abstraction, down to the executable level. A number of domain-specific languages for service composition have been proposed, with consensus gathering around the Business Process Execution Language for Web Services, which is known as BPEL4WS [4] and recently WS-BPEL [5] (or BPEL for short).

In BPEL, the logic of the interactions between a given service and its environment is described as a composition of communication actions (send, receive, send/receive, etc). These communication actions are interrelated by control-flow dependencies expressed through constructs corresponding to parallel, sequential, and conditional execution, event and exception handling, and compensation. Data manipulation is captured through lexically scoped variables as in imperative programming languages.

The constructs found in BPEL, especially those related to control flow, are close to those found in workflow definition languages [1]. In the area of workflow, it has been shown that Petri nets provide an appropriate foundation for performing static verification: Tools such as Woflan [21] are able to perform state space-based and transition invariant-based analysis on workflow models in order to verify properties such as soundness [21]. It is thus natural to conjecture that static analysis can be performed on BPEL processes by translating them to Petri nets and applying existing Petri net-analysis techniques. In particular, BPEL incorporates two sophisticated branching and synchronisation constructs, namely

[⋆] Supported by an Australian Research Council (ARC) Discovery Grant (DP0451092).

"control links" and "join conditions", which can be found in a class of workflow models known as *synchronising workflows* formalised in terms of Petri nets in [14].

The work reported in this paper aims at validating the feasibility of using Petri nets for static analysis of BPEL processes. The contributions are:

- A complete formalisation of all control-flow constructs of BPEL in terms of a mapping to Petri nets. This formalisation has served to unveil ambiguities in the current BPEL specification which have been reported to the BPEL standardisation committee.[3]
- A tool (WofBPEL) that employs the output of the above mapping to perform three types of analysis:
  - Checking for unreachable activities.
  - Checking for potentially conflicting "message receipt" actions.
  - Determining, for each action in a BPEL process definition, which messages might eventually be consumed by the process after this action has been performed. This information can be used by a BPEL engine in order to detect which messages in the inbound queue may be discarded and thus perform "garbage collection".

While other formalisations of BPEL have been proposed (see Sect. 2), we have found that none of them is as detailed in terms of capturing control-flow constructs (especially "join conditions") as the one presented in this paper, and none of them has led to a publicly available tool that performs static analysis as described above.

The rest of the paper is organised as follows. Sect. 2 gives a brief introduction to BPEL and a review of related work on formalisation and analysis of BPEL. Sect. 3 provides an informal description of the mapping from BPEL to Petri nets. Sect. 4 presents a formal definition of this mapping, including an abstract syntax of BPEL. Sect. 5 discusses the analysis of BPEL processes using the WofBPEL tool. Finally, Sect. 6 concludes and outlines future work.

2 Background and Related Work

In this section, we provide an overview of BPEL and review related formalisation efforts.

2.1 Overview of WS-BPEL

BPEL is designed to support the description of both behavioural service interfaces and executable service-based processes. A *behavioural interface* (known as *abstract process*) is a specification of the behaviour of a class of services, capturing constraints on the ordering of messages to be sent to and received from a service. An *executable process*, which is the focus of this paper, defines the execution order of a set of activities (mostly communication activities), the partners involved in the process, the messages exchanged between partners, and the events and exception handling specifying the behaviour when specific events or faults occur.

A BPEL process definition relates a number of *activities*. Activities are split into two categories: basic and structured activities. *Basic activities* correspond to atomic actions such as: *invoke*, invoking an operation on some web service; *receive*, waiting for a message from an external partner; *reply*, replying to an external partner; *wait*, waiting for a certain period of time; *assign*, assigning a value to a variable; *throw*, signaling a fault in the execution; *exit*, terminating the entire service instance; and *empty*, doing nothing. *Structured activities* impose behavioural and execution constraints on a set of activities contained within them. These include: *sequence*, for defining an execution order; *flow*, for parallel routing; *switch*, for conditional routing; *pick*, for capturing a race between timing and message receipt events; *while*, for structured looping; and *scope*, for grouping activities into blocks to which event, fault and compensation handlers may be attached (see below). Structured activities can be nested and combined in arbitrary ways, which enables the presentation of complex structures in a BPEL process.

The *sequence*, *flow*, *switch*, *pick* and *while* constructs provide a means of expressing structured flow dependencies. In addition to these constructs, BPEL provides another construct know as *control links* which, together with the associated notions of *join condition* and *transition condition*, support the definition of precedence, synchronization and conditional dependencies on top of those captured by the

[3] See discussions associated to issues 189, 192 and especially 200 in the OASIS WS-BPEL web site `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`.

structured activity constructs. A control link between activities A and B indicates that B cannot start before A has either completed or has been "skipped". Moreover, B can only be executed if its associated *join condition* evaluates to true, otherwise B is skipped. This join condition is expressed in terms of the tokens carried by control links leading to B. These tokens may take either a *positive* (true) or a *negative* (false) value. An activity X propagates a token with a positive value along an outgoing link L if and only if X was executed (as opposed to being skipped) and the transition condition associated to L evaluates to true. Transition conditions are boolean expressions over the process variables (just like the conditions in a *switch* activity). The process by which positive and negative tokens are propagated along control links, causing activities to be executed or skipped, is called *dead path elimination*.

Control links may cross the boundaries of most structured activities. However, they must not create cyclic control dependencies and must not cross the boundary of a *while* activity or a *serializable scope*.[4] Prior to our work, the interaction between structured activities and control links was not fully understood, resulting in ambiguities and contradictions in the wording of the BPEL specification [5]. Following our formalisation effort, some of these issues were reported and discussed in the BPEL standardisation committee, and changes to the specification's wording have been proposed, albeit not yet adopted (see footnote 3).

Also, while the control flow constructs of BPEL have been designed in a way that ensures that no BPEL process execution can deadlock [16], some combinations of structured activities (in particular *switch* and *pick*) with control links can lead to situations where some activities are "unreachable". Consider the process depicted in Fig. 1. Necessarily, either A_1 or A_2 will be skipped because these two activities are placed in different branches of a *switch* and in any execution of a *switch* only one branch is taken. Thus, one of the two control links x_1 or x_2 will carry a negative token. On the other hand, we assume that the join condition attached to activity A_3 (denoted by keyword "AND") evaluates to true if and only if both links x_1 and x_2 carry positive values. Hence, this join condition will always evaluate to false and activity A_3 is always skipped (i.e. it is unreachable).

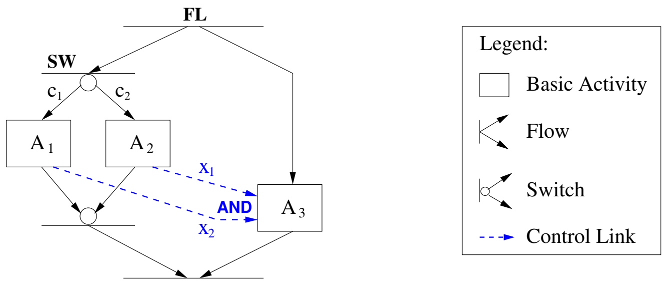


Fig. 1. An example of an unreachable activity (where control links are denoted by dashed lines).

Another family of control flow constructs in BPEL includes *event*, *fault* and *compensation handlers*. An *event handler* is an event-action rule associated with a scope. An event handler is enabled when its associated scope is under execution and may execute concurrently with the main activity of the scope. When an occurrence of the event associated with an enabled event handler is registered (and this may be a timeout or a message receipt), the body of the handler is executed while the scope's main activity continues its execution. *Fault handlers* on the other hand define reactions to internal or external faults that occur during the execution of a scope. Some of these faults may be raised explicitly using the *throw* activity. Unlike event handlers, fault handlers do not execute concurrently with the scope's main activity. Instead, this main activity is interrupted before the body of the fault handler is executed. Finally, *compensation handlers*, in conjunction with the *compensate* basic activity, enable a process to undo the effect of an already completed scope. When the compensate activity is executed for a given scope, the compensation handler of this scope will be executed when it is available. This may involve the execution of the compensation handlers associated to only the sub-scopes of the above given scope.

2.2 Related Work

The provisioning of a formal semantics of BPEL has been the subject of many previous work. Many of these efforts however, focus on small subsets of BPEL like for example only on structured activities [11,12]

[4] serializable scopes are not covered in this paper since they are not a control-flow construct and thus fall outside the scope of this work. Instead, serializable scopes are fundamentally related to data manipulation.

which is a relatively simple subset. A comparative summary of previous efforts in this area is given in Table 1. The columns of the table correspond to the following criteria:

- *Tech* indicates the formalisation technique used: FSM for finite state machines, PA for Process Algebra, ASM for Abstract State Machines and PN for Petri Nets.
- *SA* indicates whether the formalisation covers structured activities fully (+), partially (+/-) or not at all (-). It can be seen that all the cited work cover this subset of BPEL.
- *CL* indicates whether the formalisation covers control links. Here a +/- rating is given for formalisations that cover control link constructs but do not fully cover join conditions. In particular, [13,16,19] map each join condition to a single transition in the Petri net, thus losing information about the details of the join condition expression. As we will show later, join condition expressions can be fully mapped onto Petri nets, and doing so allows us to automatically detect unreachable tasks such as the one illustrated in Fig. 1. More generally, a full mapping of join conditions is needed in order to perform different forms of reachability analysis as outlined in Sect. 5.
- *EH* indicates whether the formalisation covers event and exception handling. Some references cover fault handling, but do not cover compensation and/or event handling, in which case, a +/- rating is assigned.
- *TAV* indicates whether a tool for automatic verification is provided. In the case of some efforts [10, 15], the authors claim to have developed and/or used a tool to verify deadlock-freeness of BPEL processes. However, it has been proved that BPEL processes (abstracting from inter-process communication) are always deadlock-free [16] and hence such analysis is unnecessary. Other work [13,16,19] claim to have developed an automatic verification tool which has been used to check for processes that contain control links forming cyclic dependencies. However, this is a syntactic property which can be verified by applying a simple transitive closure algorithm and does not require the use of Petri net-based analysis techniques. Finally, some of the cited references refer to the possibility of performing formal verification [8, 9], but do not develop automated means of doing so.

Table 1. A comparative summary of related work on BPEL formalisation and analysis.

	Tech	SA	CL	EH	TAV
[12]	FSM	+	-	-	+/-
[11]	FSM	+	-	-	+/-
[10]	FSM	+	-	+/-	+/-
[9]	PA	+	-	+	-
[15]	PA	+	+	-	+/-
[8]	ASM	+	+/-	+	-
[13,16,19]	PN	+	+/-	+	+/-
our work	PN	+	+	+	+

Table 1 shows that our work is the first full formalisation of control flow in BPEL that has led to a verification tool capable of performing useful and non-syntactic analysis. In addition, our BPEL verification tool is publicly available. For the sake of balance, we note that some of the previous efforts have addressed verification issues related to communication aspects, while our work focuses exclusively on control flow. Specifically, Fu et al [12] discuss how automata-based tool can be used to verify the correctness of a collection of inter-communicating BPEL processes. Similarly, Martens [17] shows how a Petri net-based tool could be used to check the compatibility of two services with respect to communication.

This paper follows on our previous work on formalisation of BPEL. A less complete and earlier version of the formalisation presented here (without the tool support) can be found in [20], while an informal analysis of BPEL in terms of a set of workflow patterns is given in [22].

3 Informal Introduction to the Mapping of WS-BPEL

In this section we informally establish a mapping of the WS-BPEL control flow constructs to Petri nets. When using Petri nets for capturing formal semantics of WS-BPEL, we allow the usage of both labeled and unlabelled transitions. The labeled transitions are used to model events and basic activities. The transitions without a label, which we hereafter refer to as λ-transitions, represent internal actions that cannot be observed by external users.

3.1 Activities

We start with the mapping of a basic activity (X) shown in Fig. 2, which also illustrates our mapping approach for structured activities. The net is divided into two parts: one (drawn in solid lines) models the normal processing of X, the other (drawn in dashed lines) models the skipping of X. In the normal processing part, place $\mathsf{r_X}$ ("ready") models an initial state when it is ready to start activity X before checking the status of all control links coming into X, and place $\mathsf{f_X}$ ("finished") indicates a final state when both X completes and the status of all control links leaving from X have been determined. The transition labeled X models the action to be performed. This is an abstract way of modelling basic activities, where the core of each activity is considered as an atomic action. Transition X has an input place $\mathsf{s_X}$ ("started") for the state when activity X has started, and an output place $\mathsf{c_X}$ ("completed") for the state when X is completed. Two λ-transitions (drawn as solid bars) model internal actions for checking pre-conditions or evaluating post-conditions for activities. The skip path is used to facilitate the mapping of control links, which will be described in Sect. 3.2. Note that the to_skip and skipped places are respectively decorated by two patterns (a letter Y and its upside-down image) so that they can be graphically identified. In Fig. 2, hiding the subnet enclosed in the box labeled X yields an abstract graphic representation of the mapping for activities. This will be used in the rest of the paper.

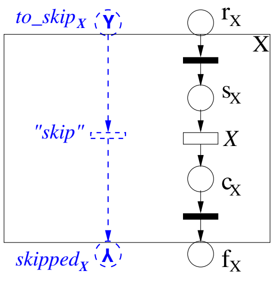


Fig. 2. A basic activity.

Fig. 3 depicts the mapping of structured activities (except *scopes*) with a focus on their normal behaviour. Since *scopes* are mainly concerned with exception handling, their mapping will be given in Sect. 3.3. In Fig. 3, next to the mapping of each activity is a BPEL snippet of the activity. More λ-transitions (drawn as hollow bars) are introduced for the mapping of routing constructs. Note that in Figure 3 and subsequent figures, the skip path of the mapping is not shown when it is not used.

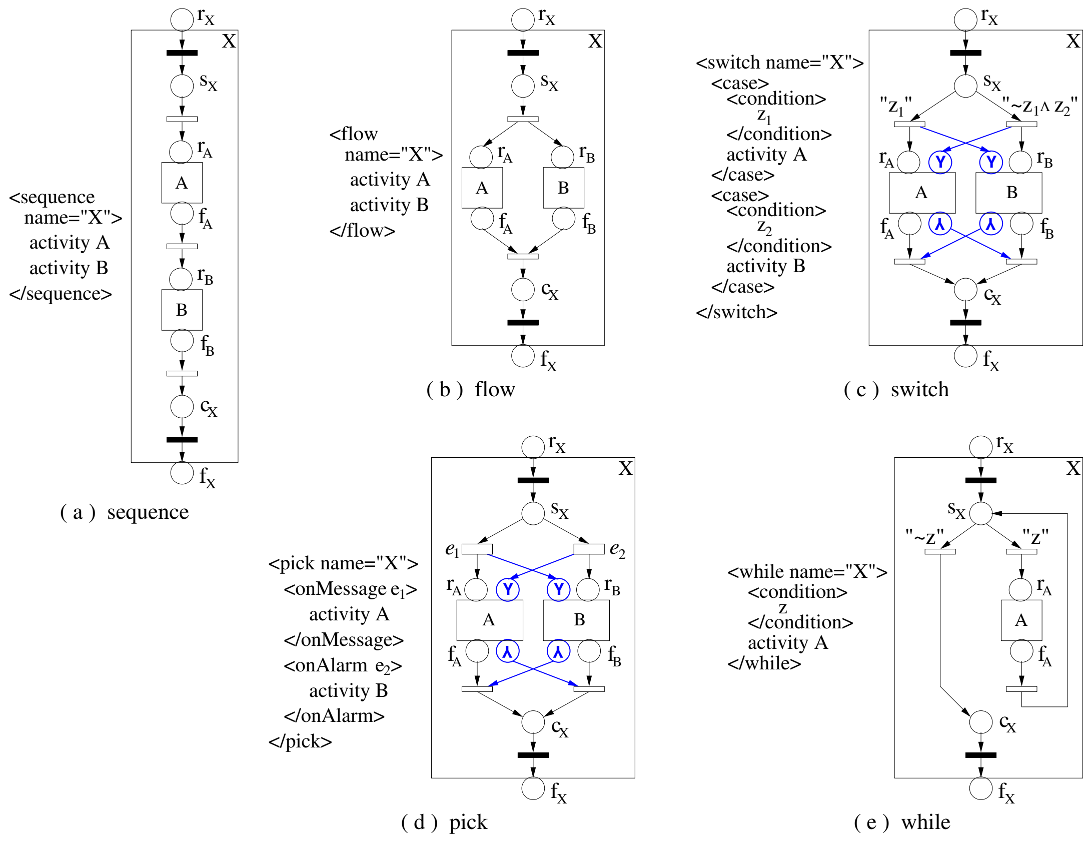
A *sequence* activity consists of one or more activities that are executed sequentially. A *flow* activity provides parallel execution and synchronization of activities. The corresponding mappings in Fig. 3(a) and (b) are straightforward.

A *switch* activity supports conditional routing between activities. In Fig. 3(c), as soon as one of the branches is taken in activity X the other needs to be skipped, and X will not complete until both the activity in the selected branch is finished and the activity in the other branch is skipped. Also, among the set of branches in a switch activity, the first branch whose condition holds will be taken. In Fig. 3(c), this is captured by the two λ-transitions annotated by $\mathsf{z_1}$ or $\sim \mathsf{z_1} \wedge \mathsf{z_2}$, where $\mathsf{z_1}$ and $\mathsf{z_2}$ are conditions for the branches with activity A or B, respectively.

A *pick* activity exhibits the conditional behaviour where decision making is triggered by external events or system timeout. It has a set of branches in the form of an event followed by an activity, and exactly one of the branches is selected upon the occurrence of the event associated with it. There are two types of events: *message events* (`onMessage`) which occur upon the arrival of an external message, and *alarm events* (`onAlarm`) which occur upon timeout. In Fig. 3(d), a pick activity is modelled in a similar way as a switch activity, except for the two transitions, labeled $\mathsf{e_1}$ or $\mathsf{e_2}$, which model the corresponding events. As compared to the two *local* λ-transitions annotated (by conditions associated with branches) in the mapping of a switch activity (in Fig. 3(c)), the event transitions $\mathsf{e_1}$ and $\mathsf{e_2}$ (in Fig. 3(d)) are *global* transitions enabled upon external or system triggers.

A *while* activity supports structured loops. In Fig. 3(e), activity X has a sub-activity A that is performed multiple times as long as the while condition (z) holds and the loop will be exit if the condition does not hold any more ($\sim\mathsf{z}$).

From the above, except for a *while* activity (which always contains one sub-activity), we have considered two sub-activities when mapping the normal behaviour of structured activities. This yields a binary version of the mapping, which can be easily extended to an n-ary version.



```
<sequence
   name="X">
   activity A
   activity B
</sequence>
```

```
<flow
   name="X">
   activity A
   activity B
</flow>
```

```
<switch name="X">
  <case>
    <condition>
      z1
    </condition>
    activity A
  </case>
  <case>
    <condition>
      z2
    </condition>
    activity B
  </case>
</switch>
```

```
<pick name="X">
  <onMessage e1>
    activity A
  </onMessage>
  <onAlarm e2>
    activity B
  </onAlarm>
</pick>
```

```
<while name="X">
  <condition>
    z
  </condition>
  activity A
</while>
```

Fig. 3. Structured activities (normal behaviour).

The mapping of skipping an entire structured activity is shown in Fig. 4. To capture the control dependencies generated by structural constructs like *sequence*, we define separately the mapping of skipping a non-sequence activity in Fig. 4(a), and the mapping of skipping a sequence activity in Fig. 4(b). In both mappings, a skipping place is added to specify an intermediate state when the structured activity (X) waits for all its sub-activities (X_1 to X_n) to be skipped before X itself can be skipped. In Fig. 4(a), when a non-sequence activity is skipped, all its sub-activities will be skipped in parallel. Whereas in Fig. 4(b), when a sequence activity is skipped, all its sub-activities need to be skipped in the same order as their normal occurrences in the sequence.

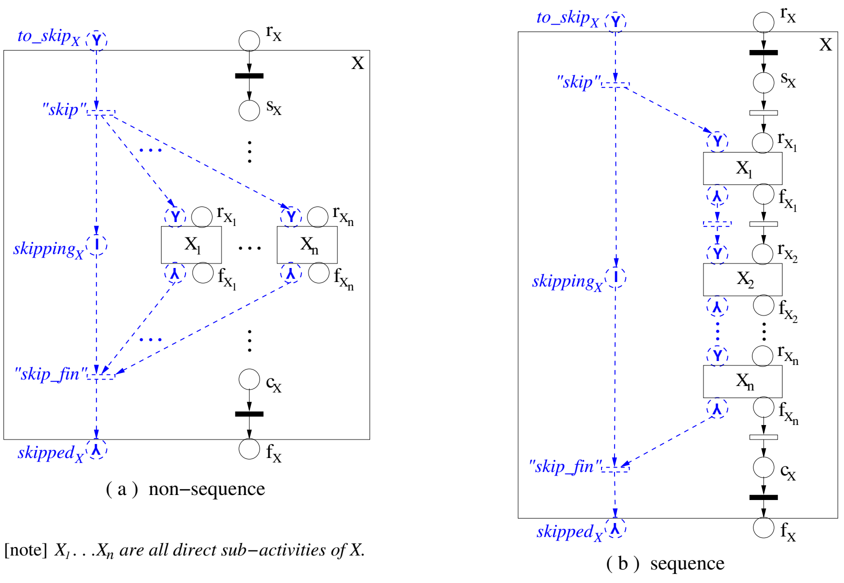
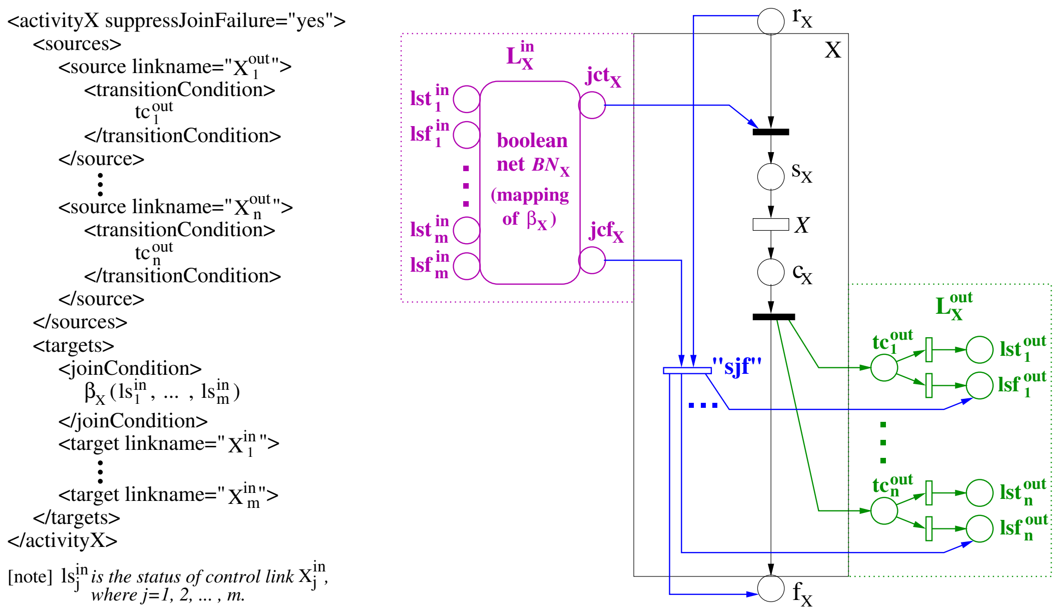


Fig. 4. Skipping structured activities.

3.2 Control Links

Control links are non-structural constructs used to express control dependencies between activities. Fig. 5 depicts the mapping of control links using an example of a basic activity. The given BPEL snippet specifies that activity X is the *source* of links X_1^{out} to X_n^{out} and the *target* of links X_1^{in} to X_m^{in}. Each control link has a *link status* that may be set to true or false, as represented by place lst ("link status true") or lsf ("link status false").



```
<activityX suppressJoinFailure="yes">
  <sources>
    <source linkname="X_1^out">
      <transitionCondition>
        tc_1^out
      </transitionCondition>
    </source>
    ⋮
    <source linkname="X_n^out">
      <transitionCondition>
        tc_n^out
      </transitionCondition>
    </source>
  </sources>
  <targets>
    <joinCondition>
      β_X (ls_1^in, ... , ls_m^in)
    </joinCondition>
    <target linkname="X_1^in">
    ⋮
    <target linkname="X_m^in">
  </targets>
</activityX>
```

[note] ls_j^{in} *is the status of control link* X_j^{in}, *where j=1, 2, ... , m.*

Fig. 5. A basic activity with control links.

The subnet enclosed in the box labeled $\mathsf{L_X^{out}}$ specifies the mapping of outgoing links from activity X. Once X is complete, it is ready to evaluate *transition conditions*, which determine the link status for each of the outgoing links. Since transition condition expressions are part of the data perspective, they are not explicitly specified in the mapping and their boolean evaluation is modelled non-deterministically.

The subnet enclosed in the box labeled $\mathsf{L_X^{in}}$ specifies the mapping of incoming links to activity X. A *join condition* is defined as a boolean expression (e.g. $\beta_X(ls_1^{in},...,ls_m^{in})$) in the set of variables representing the status of each of the incoming links. It is mapped to a boolean net ($\mathsf{BN_X}$), which takes the status of all incoming links as input and produces an evaluation result as output to place $\mathsf{jct_X}$ ("join condition true") or $\mathsf{jcf_X}$ ("join condition false"). The definition of this boolean net is given in Sect. 4.2.

If the join condition evaluates to true, activity X can start as normal. Otherwise, a fault called *join failure* occurs. A join failure can be handled in two different ways, as determined by a so-called `suppressJoinFailure` attribute associated with X. If this attribute is set to `yes`, the join failure will be suppressed, as modelled by transition "sjf" ("suppress join failure"). In this case, the activity will not be performed and the status of *all* outgoing links will be set to false. This is known as *dead path elimination* in the sense that suppressing a join failure has the effect of propagating the false link status transitively along the paths formed by control links, until a join condition is reached that evaluates to true. An activity for which a join failure is suppressed, will end up in the "finished" state (e.g. $\mathsf{f_X}$) as if it is completed as normal, and thus the processing of any following activity will not be affected. Otherwise, if `suppressJoinFailure` is set to `no`, a join failure needs to be thrown, which triggers a standard fault handling procedure as described in Sect. 3.5.

The mapping of skipping a basic activity with control links is shown in Figure 6. Such an activity X, if asked to skip, cannot be skipped until the status of all incoming links has been determined. Place $\mathsf{jcv_X}$ is used to collect either true or false token which represents the corresponding result of join condition evaluation, as such result will not affect the skipping behaviour of X. In this way, we capture the control dependency between activity X and the source activity of each of the incoming links to X. As soon as activity X is skipped, the status of each of the outgoing links from X will be set to false, which captures the dead path elimination as a result of skipping the activity.

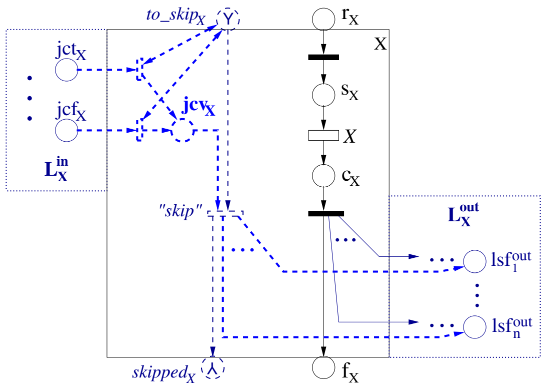


Fig. 6. Mapping of skipping a basic activity with control links.

We now extend the mapping of control links for structured activities. This is shown in Fig. 7 which includes mappings for both normal behaviour (a) and skipping behaviour (b) of a non-sequence activity. For the mapping of suppressing a join failure in Fig. 7(a), place $\mathsf{to\_f_X}$ is added to capture an intermediate state when activity X waits for all its sub-activities X_1 to X_n to be skipped, before it is finished ($\mathsf{f_X}$). For both mappings in Fig. 7(a) and (b), the dead path elimination is captured upon the completion of suppressing the join failure (transition "$\mathsf{sjf_{fin}}$") or skipping all sub-activities of X (transition "$\mathsf{skip_{fin}}$"). The mapping for a sequence activity can be extended in a similar way, as shown in Fig. 8.

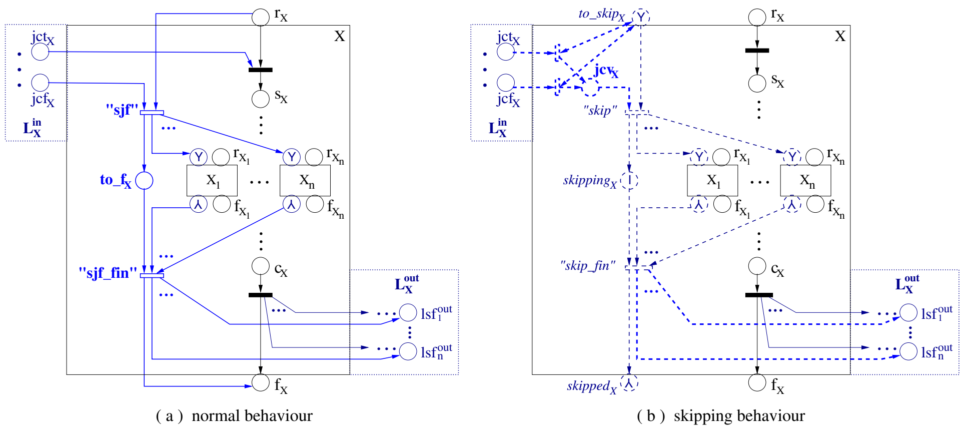


Fig. 7. A non-sequence structured activity with control links.

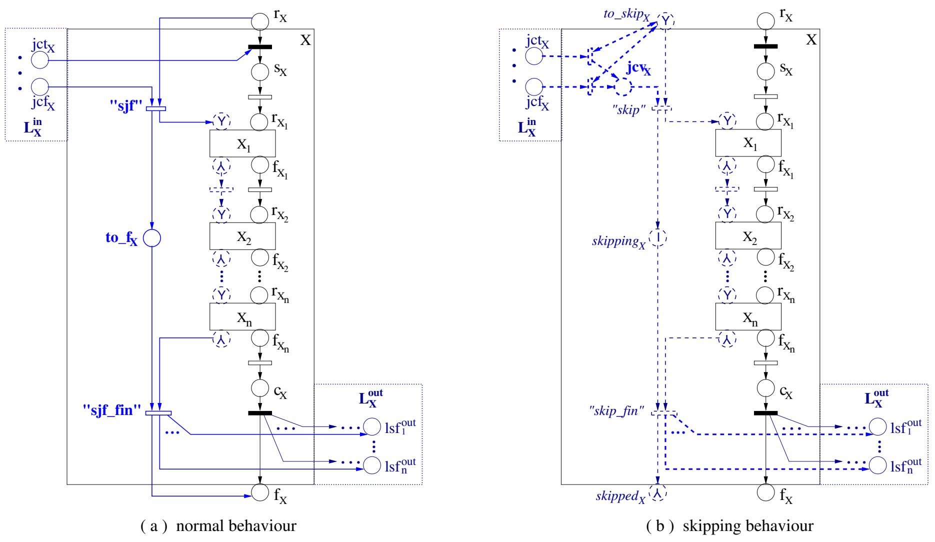


Fig. 8. A sequence activity with control links.

As an example, Figure 9 depicts the mapping of the BPEL process shown in Figure 1.

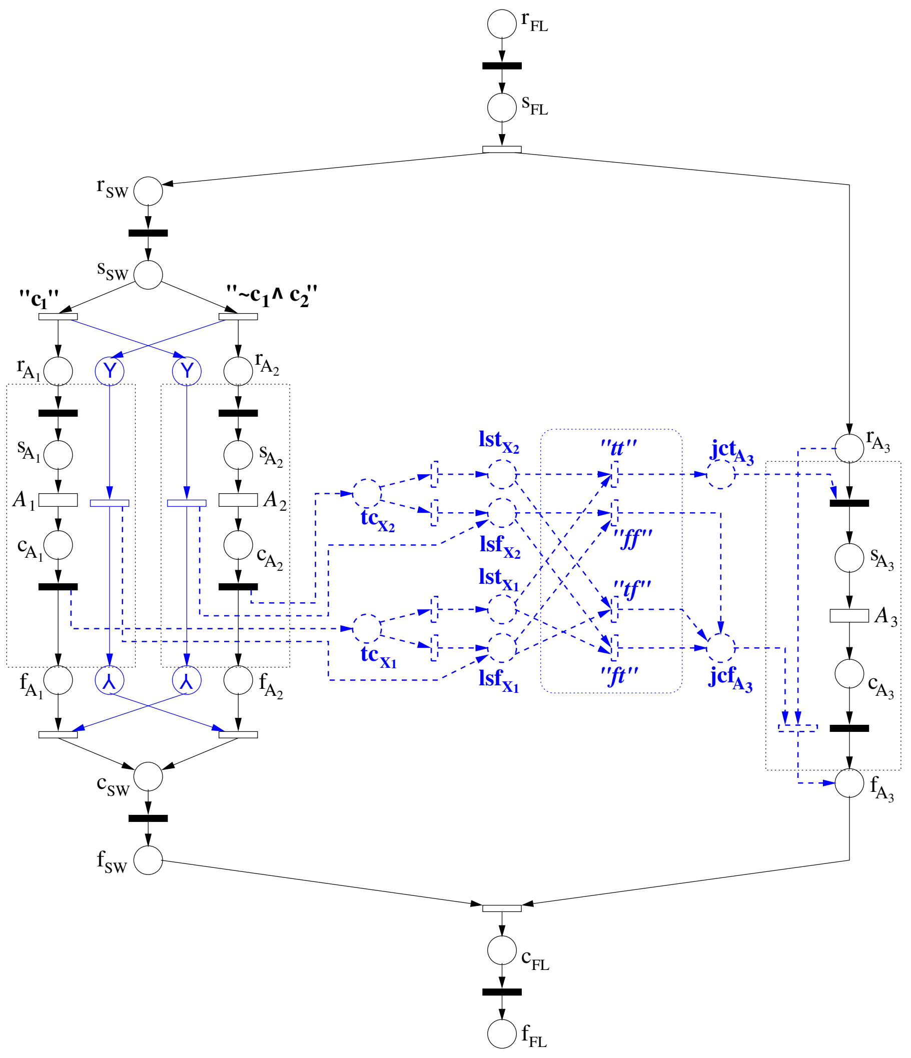


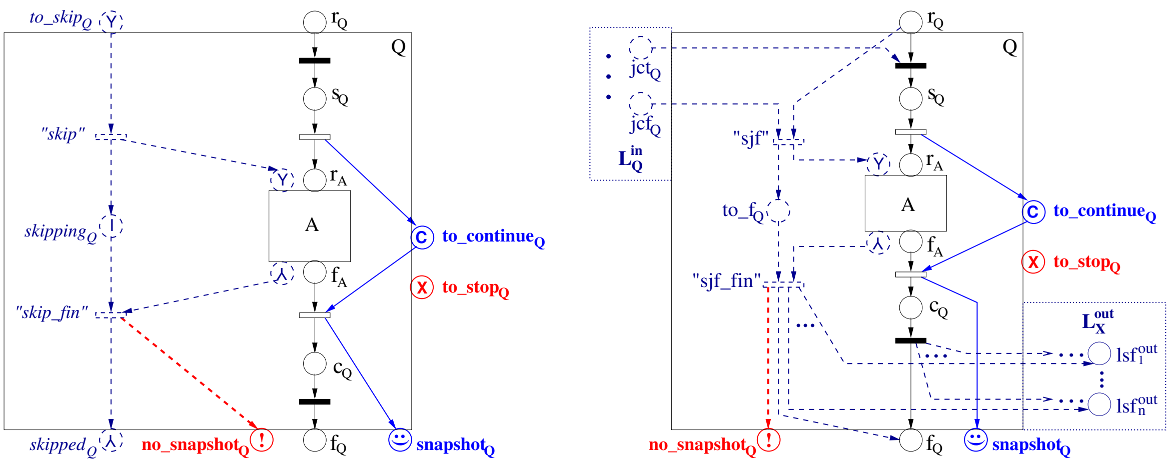
Fig. 9. Mapping of the BPEL process shown in Figure 1.

3.3 Scopes

A *scope* is a special type of structured activity defined for event and exception handling. It has a primary (i.e. main) activity, and can provide *event handlers* (Sect. 3.4), *fault handlers* (Sect. 3.5) and a *compensation handler* (Sect. 3.6). Like other structured activities, scopes can be nested to arbitrary depth, and the whole process is implicitly regarded as the top level scope.

To facilitate the mapping of exception handling, we define four flags for a scope. These are: to_continue, indicating the execution of the scope is in progress and no exception has occurred; to_stop, signaling an error has occurred and all active activities nested in the scope need to stop; snapshot, capturing the *scope snapshot* defined in [5] which refers to the preserved state of a successfully completed uncompensated scope; and no_snapshot, indicating the absence of a scope snapshot.

Fig. 10 depicts the basic mapping for a scope (Q) in which the mapping of any event or exception handler associated with the scope is not included. Assume that no exception occurs. Scope Q remains in the status of to_continue during its normal performance (i.e. the execution of Q's main activity A). Upon the completion of activity A, a snapshot is preserved for scope Q. Next, consider the case of skipping scope Q (see Fig. 10(a)) or the case of suppressing a join failure for Q (see Fig. 10(b)). Once activity A has been skipped (upon the occurrence of transition "skip_fin" or "sjf_fin"), the status indicating the absence of a scope snapshot for Q (place no_snapshot_Q) will be recorded. Finally, faults may occur during the normal performance of scope Q, causing the status of Q to change from to_continue to to_stop. This will be described further in Sect. 3.5.



(a) skipping behaviour (b) suppressing join failure

Fig. 10. A scope with its main activity.

3.4 Event Handlers

A scope can provide *event handlers* that are responsible for handling normal events (i.e. message or alarm events) that occur *concurrently* while the scope is running. A message event handler can be triggered multiple times if the expected message event occurs multiple times, and an alarm event handler, except for a `repeatEvery` alarm, can be invoked at most once (upon timeout). The `repeatEvery` alarm event occurs repeatedly upon each timeout when the scope is active, and the corresponding event handler can be invoked multiple times as long as the alarm event occurs.

We discuss a couple of decisions made for the mapping of event handlers. Firstly, since no control links are allowed to cross the boundary of event handlers, each event handler can be viewed as an independent unit of activities within a scope. Secondly, the occurrence of an event is either triggered by the system (for an alarm event) or by the environment (for a message event), and the event handler is invoked only if the expected event occurs. So it is not necessary to distinguish between the mappings of the different types of event handlers.

Fig. 11 depicts the mapping of a scope (Q) with an event handler (EH). The four flags associated with the scope are omitted. The subnet enclosed in the box labeled EH specifies the mapping of event handler EH. As soon as scope Q starts, it is ready *to invoke* EH. Meanwhile, event $\mathsf{e}_{\mathsf{normal}}$ is *enabled* and may occur upon an environment or a system trigger. When $\mathsf{e}_{\mathsf{normal}}$ occurs, an instance of EH is created, in which activity HE ("handling event") is executed. EH remains active as long as Q is active. Finally, event $\mathsf{e}_{\mathsf{normal}}$ becomes disabled once the normal process of Q (i.e. Q's main activity A) is finished. However, if a new instance of EH has already started when $\mathsf{e}_{\mathsf{normal}}$ is disabled, it is allowed to complete. The completion of the whole scope is delayed until all active instances of event handlers have completed. Note that when a join failure occurs at activity A, event $\mathsf{e}_{\mathsf{normal}}$ will become disabled upon suppressing the join failure (see Sect. 4.2). Hence, by using the enabled place in the mapping, we are able to avoid the situation where the event handler can still be triggered if the corresponding event occurs *after* the normal process of the scope has completed. Such case violates the semantics of "disablement of events" defined in Section 13.5.5 of [5].

It is worth noting that "unlike alarm event handlers, individual message event handlers are permitted to have several *simultaneously* active instances" (Section 13.5.7 of [5]). The mapping in Fig. 11 allows an event handler to have at most one active instance at a time. By adding an arc from transition $\mathsf{e}_{\mathsf{normal}}$ to place $\mathsf{to\_invoked}_{EH}$ in the mapping, multiple active instances of an event handler may be invoked simultaneously. This however will cause the place $\mathsf{to\_invoked}_{EH}$ and thus the net to be unbounded, and the analysis of an unbounded net is always difficult to manage and requires techniques that are computationally expensive. To avoid an unbounded net, we consider another approach in which the subnet enclosed in the box labeled EH in Fig. 11 is duplicated to n subnets to capture at most n active instances of an event handler simultaneously. The resulted mapping is shown in Fig. 12. The analysis of such a net will be a parameterised analysis based on the maximal number (n) of simultaneously active instances of an event handler allowed in the process.

Example BPEL code 1:

```
<scope name="Q">
   <eventHandlers>
      <onMessage  e_normal>
         activity HE
      </onMessage>
   </eventHandlers>
   activity A
</scope>
```

Example BPEL code 2:

```
<scope name="Q">
   <eventHandlers>
      <onAlarm  e_normal>
         activity HE
      </onAlarm>
   </eventHandlers>
   activity A
</scope>
```

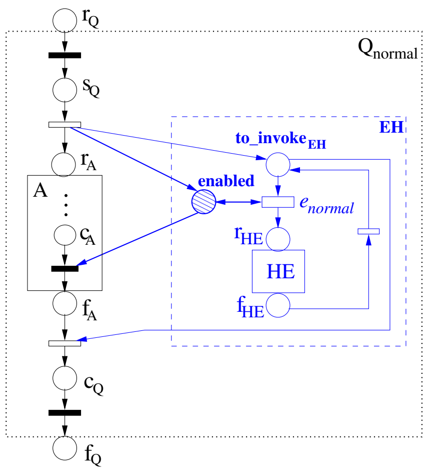


Fig. 11. A scope with an event handler.

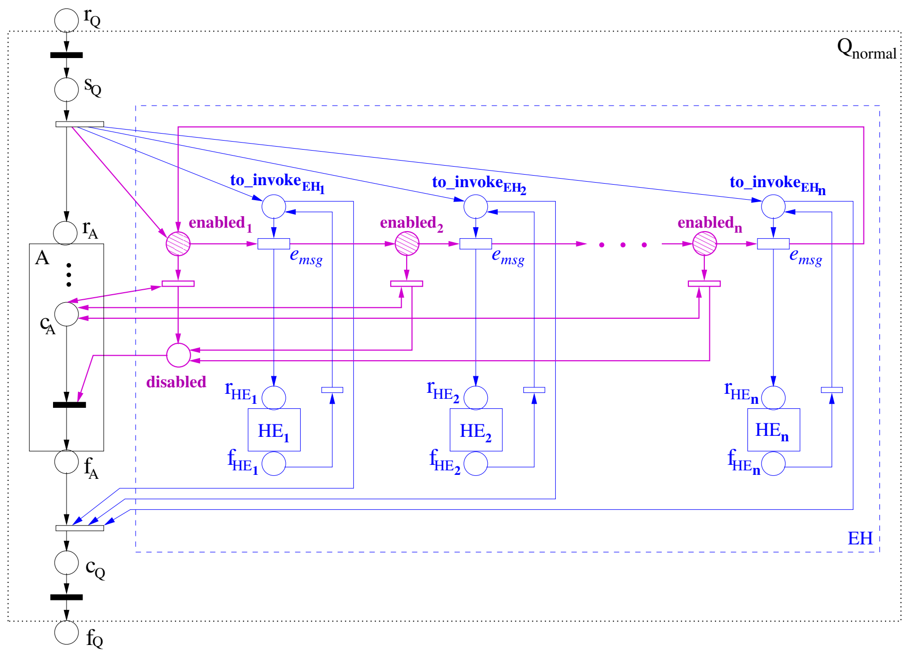


Fig. 12. A message event handler that may have a maximum of n simultaneously active instances.

3.5 Fault Handling

BPEL defines three types of faults that may arise during the process execution. These are: *application faults* (or *service faults*), which are generated by services invoked by the process, such as communication failures; *process-defined faults*, which are explicitly generated by the process using the *throw* activity; and *system faults*, which are generated by the process engine, such as join failures and datatype mismatches.

If a fault occurs during the normal process of a scope, it will be caught by one of the *fault handlers* defined for the scope. The scope switches from the normal (processing) mode to the fault handling (or faulty) mode. Note that "it is never possible to run more than one fault handler for the same scope under any circumstances" (Section 13.4 of [5]). A fault handler is defined either explicitly or implicitly. An implicit fault handler is also known as a default fault handler. It is created to catch any fault that is not caught by all explicit fault handlers within the scope. Therefore, we can assume that each scope has at least one (default) fault handler. If a fault handler cannot handle a fault being caught or another fault occurs during the fault handling, both faults need to be rethrown to the (parent) scope that directly encloses the current scope. A scope in which a fault has occurred is considered to have ended abnormally and thus cannot be compensated, no matter whether or not the fault can be handled successfully (without being rethrown) by the corresponding fault handler.

The mapping of fault handling is defined via the following steps. At first, we present a general mapping of it which can be instantiated given a specific fault. For example, process-defined faults are generated by the throw activity which can be explicitly specified in the mapping, while application or system faults are generated by process services or the process engine which may be treated as external or system triggers. Secondly, as compared to the faults occurred in the normal mode of a scope (which invoke the fault handling within that scope), the faults occurred in the faulty mode of a scope invoke the fault handling within its parent scope. This needs to be realised in the mapping. Thirdly, since event handlers associated with a scope are considered as a part of the normal behaviour of the scope, we also specify the behaviour of handling faults within a scope that has event handlers attached to it. Finally, control links are *only* allowed to *leave* the boundary of a fault handler. So the mapping needs to capture the dead path elimination for the outgoing links of a fault handler that will never be invoked.

General mapping of fault handling. Fig. 13 depicts the general mapping of handling a fault that occurs during the normal process of a scope (Q). The subnet enclosed in the dashed box labeled FH specifies the general mapping of a fault handler. It has a similar structure as the mapping of an event handler (shown in Fig. 11) with differences in the following three aspects.

– Firstly, as compared to the normal events defined within event handlers, faults that may arise during a process execution can be considered as *fault events*. Transition "$\mathsf{e}_{\mathsf{fault}}$" represents such fault event, and upon its occurrence, the status of scope Q changes from to_continue to to_stop. All activities that are currently active in Q need to stop, and any other fault events that may occur are disabled. In this way, we ensure that no more than one fault handler can be invoked for the same scope.
– Secondly, a fault handler, once invoked, cannot start its main activity HF ("handling fault") until the main activity has been terminated in the associated scope. This results in an intermediate state, as captured by place $\mathsf{invoked}_{\mathsf{FH}}$, after the occurrence of event $\mathsf{e}_{\mathsf{fault}}$ but before the execution of activity HF. Note that we will not describe the mapping of activity termination here (see Sect. 3.7), except to mention that an activity being terminated will end up in the "finished" state. For example, in Fig. 15, if activity A is asked to terminate, place f_{A} will get marked upon the termination of A. Accordingly, the arc from place f_{A} to the input transition of place r_{HF}, ensures that activity HF can be started only if the normal process of scope Q has been terminated.
– Finally, if the fault has been handled successfully, any control links leaving from scope Q will be evaluated normally. Accordingly, in Fig. 15, place c_{Q} will get marked. However, the status of Q will change from to_stop to no_snapshot to indicate that a fault has occurred during its performance.

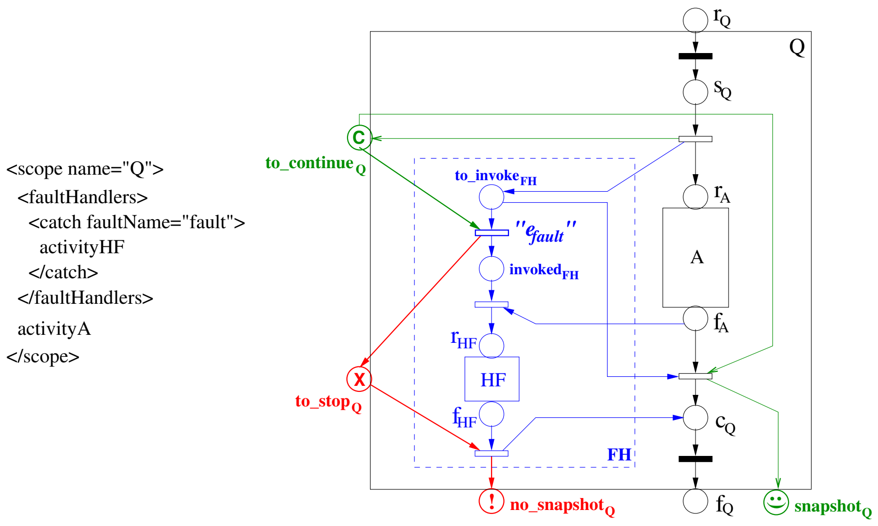
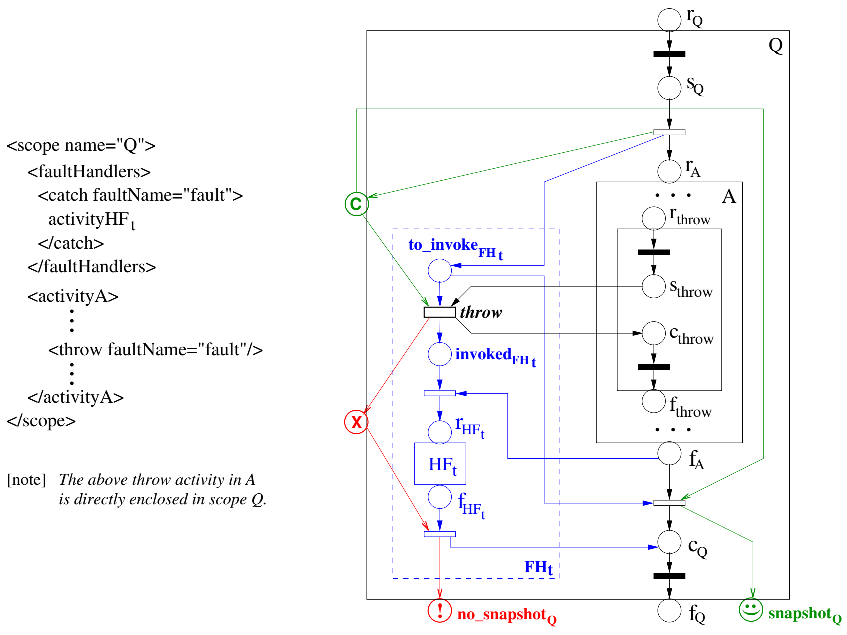

```
<scope name="Q">
  <faultHandlers>
    <catch faultName="fault">
      activityHF
    </catch>
  </faultHandlers>
  activityA
</scope>
```

Fig. 13. A fault handler (general mapping).

Handling a process-defined fault. We instantiate the above general mapping for handling process-defined faults, as shown in Fig. 14. The fault is generated by a throw activity and its occurrence triggers the fault handler FH_t as an instantiation of the fault handler FH in Fig. 13. The throw activity is directly enclosed in Q, i.e., there does not exist any intermediate scope which is both nested in Q and encloses this throw activity. In Fig. 14, the transition labeled *throw* models the action of throwing a fault, and the occurrence of the action itself resembles the occurrence of the corresponding fault event.

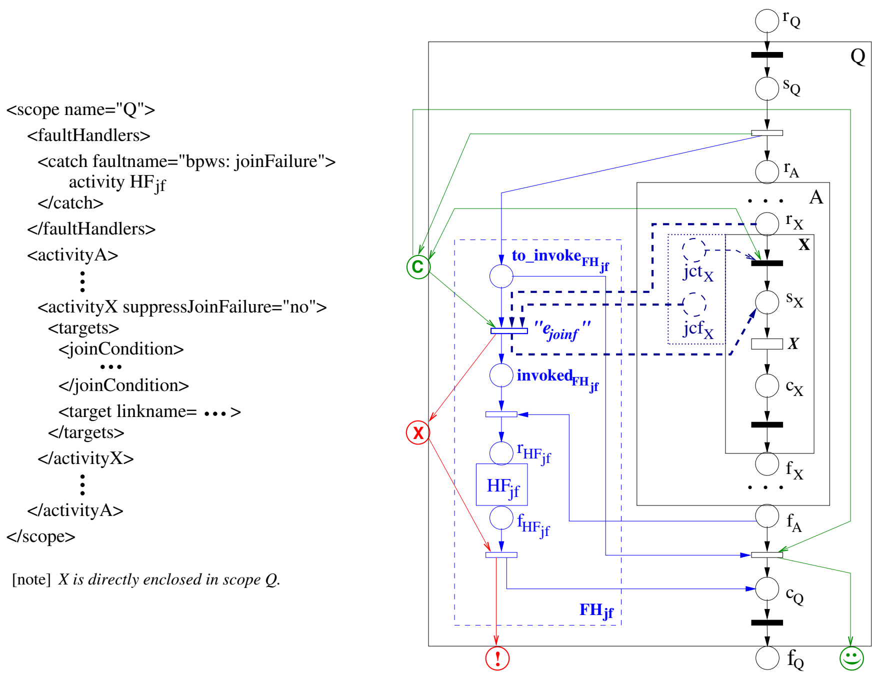


```
<scope name="Q">
   <faultHandlers>
     <catch faultName="fault">
       activityHF_t
     </catch>
   </faultHandlers>
   <activityA>
         :
       <throw faultName="fault"/>
         :
   </activityA>
</scope>
```

[note] *The above throw activity in A is directly enclosed in scope Q.*

Fig. 14. Handling a process-defined fault generated by the throw activity.

Handling a join failure fault. We instantiate the general mapping of a fault handler for a join failure fault, as shown in Fig. 15. Assume that activity X, which is part of the normal process of scope Q and is directly enclosed Q, has the `suppressJoinFailure` attribute set to `no`. The occurrence of fault event e_{joinf} will be triggered if a join failure occurs (place jcf_X being marked) at activity X and X is ready to start (place r_X being marked). The arc from transition "e_{joinf}" to place s_X allows the continuation of the flow in the normal process of scope Q. This is necessary in the mapping of activity termination (see Section 3.7) which requires a dry run of the uncompleted activities in the scope.

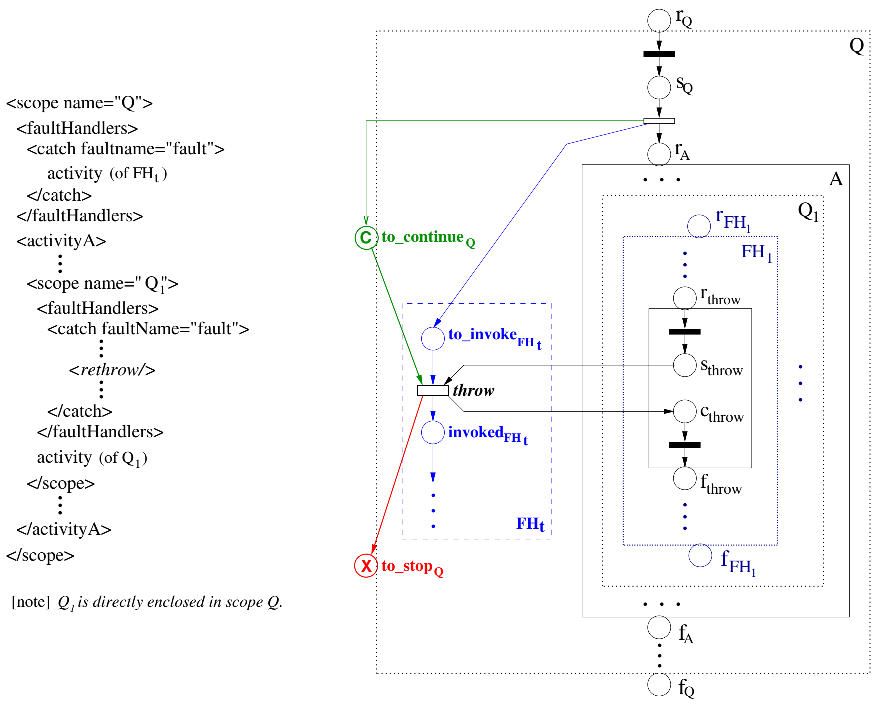


```
<scope name="Q">
   <faultHandlers>
     <catch faultname="bpws: joinFailure">
          activity HF_jf
     </catch>
   </faultHandlers>
   <activityA>
         :
     <activityX suppressJoinFailure="no">
       <targets>
         <joinCondition>
             •••
         </joinCondition>
         <target linkname= •••>
       </targets>
     </activityX>
         :
   </activityA>
</scope>
```

[note] *X is directly enclosed in scope Q.*

Fig. 15. Handling a join failure as an example of a system fault.

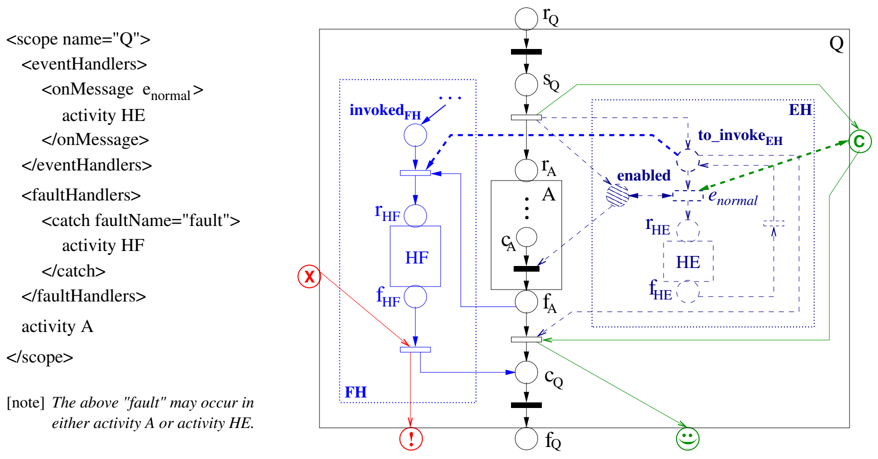
Handling faults occurred within fault handlers. Any type of activity (e.g. a scope) can be used for fault handling, and thus the main activity carried out by a fault handler may contain scope activities that are nested to arbitrary depth. If a fault occurred within a fault handler can be solved inside the fault handler, the mapping of fault handling shown in Fig. 13 can be used. Otherwise, if the fault cannot be solved locally, it needs to be thrown to outside of the fault handler. Fig. 16 depicts the mapping of throwing a fault from within a fault handler (FH_1) of the current scope (Q_1), which invokes the fault handler (FH) for the parent scope (Q) of Q_1. Note that the BPEL spec also defines a *rethrow* activity, which in essence "can be considered a macro for a <`throw`> (i.e. throw activity) that throws the fault caught by the corresponding fault handler" but cannot be solved by that fault handler (Section 13.4 of [5]).



```
<scope name="Q">
 <faultHandlers>
  <catch faultname="fault">
     activity (of FHt)
  </catch>
 </faultHandlers>
 <activityA>
     ⋮
  <scope name="Q1">
   <faultHandlers>
    <catch faultName="fault">
         ⋮
       <rethrow/>
         ⋮
    </catch>
   </faultHandlers>
   activity (of Q1)
  </scope>
     ⋮
 </activityA>
</scope>

[note] Q1 is directly enclosed in scope Q.
```

Fig. 16. Handling a fault thrown from within a fault handler.

Handling faults occurred in a scope with event handlers. Since event handlers are considered as part of the normal process of a scope, faults occurred within a scope include those occurred within the event handlers attached to the scope. Fig. 17 depicts the mapping of handling a fault occurred within a scope that has event handlers attached to it. This mapping is mainly obtained by combining the mappings of event handling (Fig. 11) and fault handling (Fig. 13) for the scope. Based on this, we add the following two arcs, both drawn in thick dashed lines. The arc from place $\mathsf{to\_invoke}_{\mathsf{EH}}$ to the input transition of place r_{HF}, is added to ensure that any active instance of event handler (EH) will terminate before the fault handling (activity HF) starts. The bidirectional arc connecting place $\mathsf{to\_continue}_{\mathsf{Q}}$ and transition "$\mathsf{e}_{\mathsf{normal}}$", is added to disable event e_{normal} once a fault occurs within scope Q.



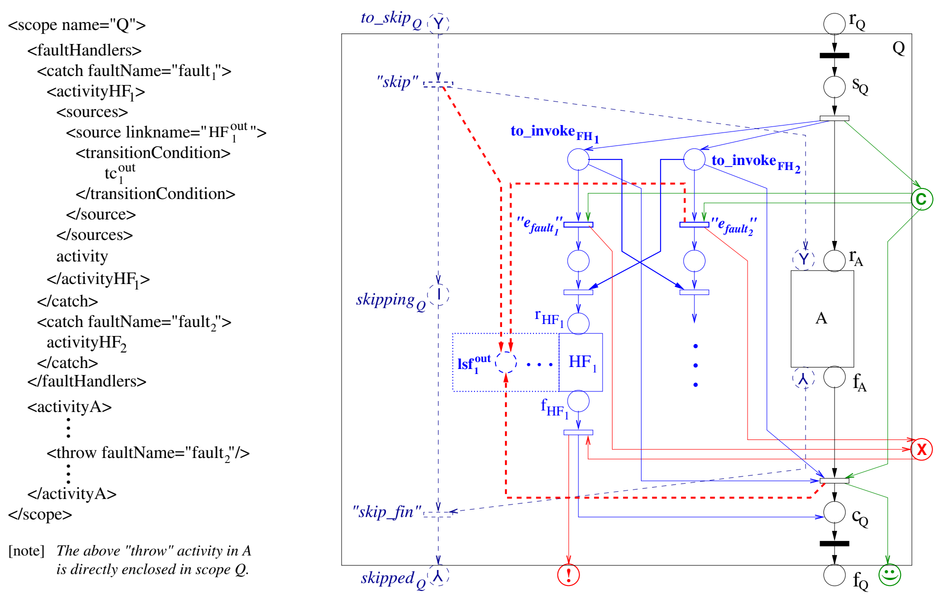
```
<scope name="Q">
  <eventHandlers>
     <onMessage  e_normal>
        activity HE
     </onMessage>
  </eventHandlers>
  <faultHandlers>
     <catch faultName="fault">
        activity HF
     </catch>
  </faultHandlers>
  activity A
</scope>

[note] The above "fault" may occur in
       either activity A or activity HE.
```

Fig. 17. Mapping of fault handling in a scope that involve event handlers.

Dead path elimination for control links leaving from a fault handler. Fig. 18 depicts the mapping of dead path elimination for an outgoing control link (HF_1^{out}) from a fault handler (FH_1). There are three possible cases, which are realised by adding three input arcs to place $\mathsf{lsf}_1^{\mathsf{out}}$ as follows.

- Firstly, assume that scope Q is completed successfully. In this case, none of the fault handlers will be executed for scope Q. The status of link HF_1^{out} will be set to false upon completion of Q, as captured by the arc from the input transition of place $\mathsf{c_Q}$ to place $\mathsf{lsf}_1^{\mathsf{out}}$.
- Secondly, consider the case when another fault event (e_{fault_2}) occurs, which invokes the corresponding fault hander (FH_2) but disables FH_1 in scope Q. The status of link HF_1^{out} will be set to false upon the occurrence of event e_{fault_2}, as captured by the arc from transition "$\mathsf{e_{fault_2}}$" to place $\mathsf{lsf}_1^{\mathsf{out}}$.
- Finally, suppose that scope Q needs to be skipped, and again none of the fault handlers in Q will be performed. The status of link HF_1^{out} will be set to false when Q is being skipped, as captured by the arc from transition "skip" to place $\mathsf{lsf}_1^{\mathsf{out}}$.

Also, note that if one of the fault handlers (e.g. FH_2) gets invoked, the other (e.g. FH_1) cannot be invoked any more. Therefore, the token in the to_invoke place of fault handler FH_1 needs consumed before activity HF_1 for handling fault e_{fault_2} starts.



```
<scope name="Q">
  <faultHandlers>
    <catch faultName="fault1">
      <activityHF1>
        <sources>
          <source linkname="HF1out">
            <transitionCondition>
                tc1out
            </transitionCondition>
          </source>
        </sources>
        activity
      </activityHF1>
    </catch>
    <catch faultName="fault2">
      activityHF2
    </catch>
  </faultHandlers>
  <activityA>
        ⋮
    <throw faultName="fault2"/>
        ⋮
  </activityA>
</scope>
```

[note] *The above "throw" activity in A is directly enclosed in scope Q.*

Fig. 18. Dead path elimination for a control link leaving from a fault handler.

3.6 Compensation

As part of the exception handling, *compensation* refers to application-specific activities that attempt to undo the work of a completed scope. Such activities are defined within a *compensation handler*. Each scope, except the top level scope (i.e. process scope), provides one compensation handler that is defined either explicitly or implicitly. Similarly to a default fault handler, an implicit (or default) compensation handler is created for a scope, if the scope is asked to be compensated but an explicit compensation handler is missing for that scope. A fault handler or the compensation handler of a given scope, may perform a *compensate* activity to invoke the compensation of one of the sub-scopes of this scope. The compensation handler of a scope is available for invocation only if the scope has a scope snapshot as mentioned in Sect. 3.3. Otherwise,"invoking a compensation handler that has not been installed (i.e. unavailable) is equivalent to the empty activity (it is a no-op)" (Section 13.3.3. of [5]).

Fig. 19 depicts the mapping of handling the compensation of a given scope (Q_1), as invoked by the compensate activity within a fault handler (FH) or compensation handler (CH) of the parent scope (Q) of Q_1. The transition labeled compensate models the atomic action of invoking a compensation. Upon its occurrence, it is ready *to invoke* the compensation handler CH_1 of scope Q_1. However, the availability of CH_1 depends on the presence of a scope snapshot of Q_1. Only if Q_1 has a scope snapshot will transition "invoke" occur. Upon its occurrence, activity HC_1 ("handling compensation" of Q_1) will be carried out, and consequently the scope snapshot of Q_1 will be consumed. If Q_1 does not have a scope snapshot, the attempt to invoke CH_1 results in an empty action, as captured by transition "no-op". The compensate activity will end upon the completion of either the compensation of Q_1 or the empty action, and the performance of activity HF or HC can be continued.

```
<scope name="Q">
  <faultHandlers>
    <catch>
      <activityHF>
          ⋮
        <compensate scope="Q1">
          ⋮
      </activityHF>
    </catch>
  </faultHandlers>
  <compensationHandler>
    <activityHC>
        ⋮
      <compensate scope="Q1">
        ⋮
    </activityHC>
  </compensationHandler>
  <activityA>
      ⋮
    <scope name="Q1">
      <compensationHandler>
        activity HC1
      </compensationHandler>
      activity
    </scope>
      ⋮
  </activityA>
</scope>

[note] Q1 is directly enclosed in scope Q.
```

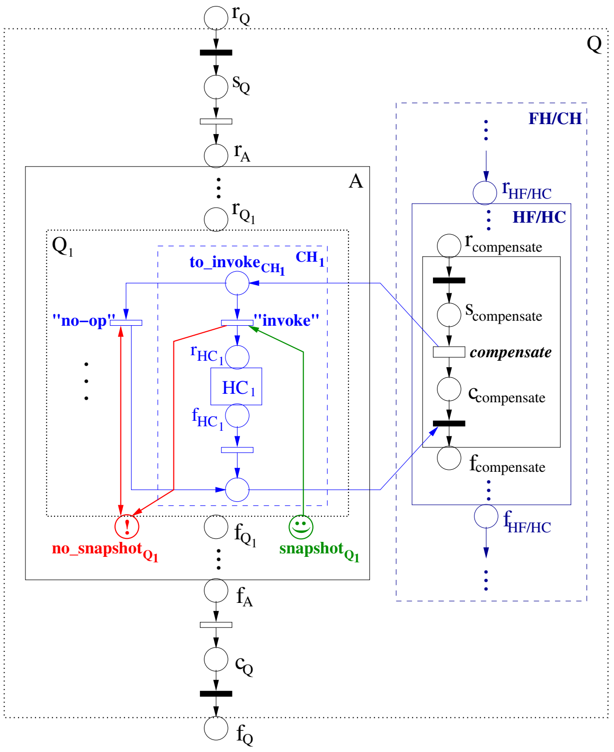


Fig. 19. Handling compensation invoked by the compensate activity.

Similarly to fault handlers, a compensation handler may contain scope activities and faults may occur during compensation. Fig. 20 depicts the mapping of throwing a fault from within a compensation handler (CH_1) of scope (Q_1). Such a fault is treated as being a fault within the scope (Q_i^{CH}) that directly encloses the compensate activity used for the compensation of scope Q_1. Therefore, if the fault is thrown, it will be caught by one of the fault handlers (FH_i) associated with scope Q_i^{CH}.

```
<scope name="Q">
 <compensationHandler>
     ⋮
   <scope name="Q_i^CH">
    <faultHandlers>
      <catch faultName="fault">
        activity (of FH_i )
      </catch>
    </faultHandlers>
    <activityA_i>
        ⋮
       <compensate scope="Q1">
        ⋮
    </activityA_i>
   </scope>
     ⋮
 </compensationHandler>
 <activityA>
    ⋮
  <scope name="Q1">
   <compensationHandler>
    <activityHC1>
        ⋮
      <throw faultname="fault">
        ⋮
    </activityHC1>
   </compensationHandler>
   activity (of Q1)
  </scope>
    ⋮
 </activity>
</scope>
```

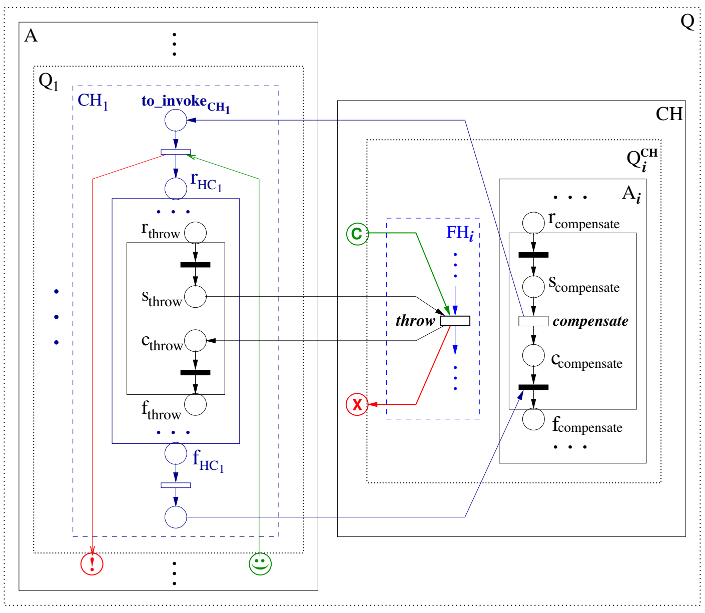


Fig. 20. Handling faults thrown from within a compensate handler.

3.7 Termination

This subsection presents the mapping for termination which includes: termination of activities, termination of a hierarchy of scopes, and termination of the entire process.

Termination of activities. The main activity of a scope will be interrupted and terminated when the scope is faulty. If the fault has been handled successfully, the scope will end as if it is completed normally and thus the processing of its parent scope will not be affected. This requires that the control dependencies should also be preserved during activity termination. Thus, for the mapping of activity termination, we adopt an approach of conducting a dry run of the activity without performing its concrete actions nor allowing it to process any normal events (i.e. message or alarm events). In BPEL, concrete actions are defined by basic activities, while structured activities are mainly a set of pre-defined structural constructs for composition of activities in the process.

Fig. 21 depicts the mapping for the termination of a basic activity with `suppressJoinFailure` attribute set to `yes` (a) or `no` (b). We assume that activity X is directly enclosed in scope Q and has incoming and outgoing links. As shown in both mappings, the core action of X (transition X) can be performed only if Q is allowed *to continue* its normal process. Otherwise, if Q is asked *to stop*, the core action of X will be bypassed, as captured by the λ-transition "bypass". The dead path elimination is realised by imposing an additional condition to link status evaluation of each outgoing link from X, such that a true status may be obtained only if Q is in the normal processing mode.

Suppose that a join failure occurs at activity X when X is asked to terminate. A dry run of suppressing join failure can be performed by transition "sjf" shown in Fig. 21 (a). However, it is not possible to conduct a dry run of throwing join failure to scope Q when Q is asked to stop. Recall the mapping of handling a join failure fault in Fig. 15, where $\mathsf{e}_{\mathsf{joinf}}$ may be triggered only within the normal process of Q. In order to continue the dry run of activity X which attempts to throw a join failure during termination of scope Q, we add transition "ignore" to the mapping in Fig. 21 (b). This transition will fire if a join failure occurs at X and X needs to be terminated. As a result, the join failure is ignored and the dry run of X is continued.

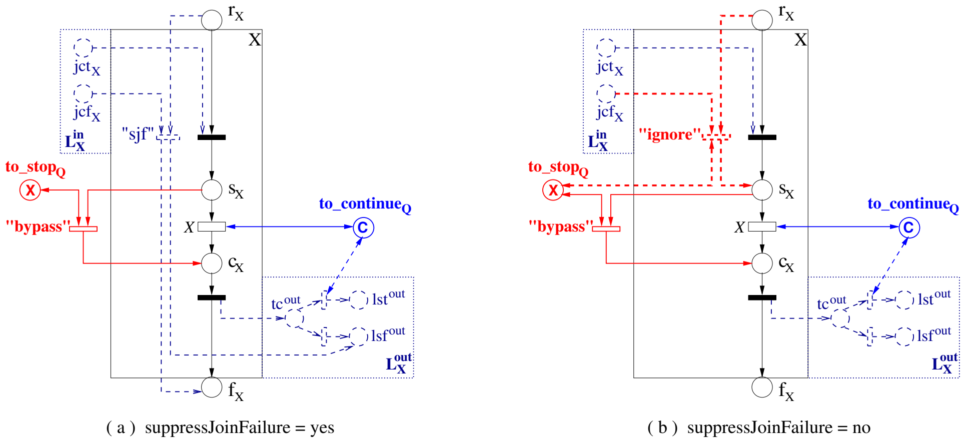
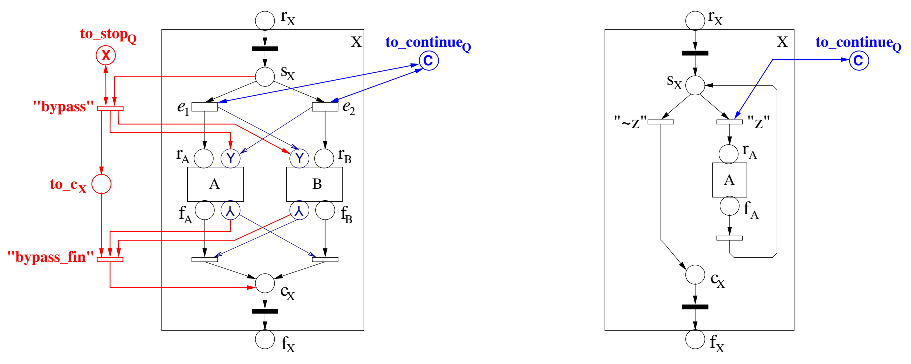


Fig. 21. Termination of a basic activity.

The mapping of the termination of structured activities is defined in a similar way to the mapping for basic activities. In the process of termination, the dry run of a structured activity will be performed until it reaches a basic activity of which the core action will be bypassed. However, there are exceptions which apply to pick activities and while activities, as shown in Fig. 22.

A pick activity involves normal events (i.e. message or alarm events) of which the processing is not allowed upon termination. Fig. 22 (a) depicts the mapping of bypassing all branches of a pick activity (X) during its termination. The bypass path can be viewed as an additional branch added to X. It is taken only if X is asked to terminate. Also, it is the only branch that can be taken upon termination.

A while activity, if asked to terminate, can no longer execute its sub-activity. Fig. 22 (b) depicts the mapping of how a while activity (X) stops further iteration of its sub-activity (A) as soon as its directly enclosing scope (Q) leaves the normal processing mode (i.e. the token is removed from place $\mathsf{to\_continue}_{\mathsf{Q}}$). If activity X needs to terminate, no matter whether or not the while condition z holds, only transition "$\sim$z" can occur, leading to the end of activity X.



(a) pick (b) while

Fig. 22. Exceptions applied to the termination of a pick activity (a) and a while activity (b).

Termination of a hierarchy of scopes. If a scope needs to terminate, all the active (sub-)scopes nested within this scope are forced to terminate. However, "the already active fault handler (of any nested scope) is allowed to complete" (Section 13.4.4 of [5]). If a non-faulty scope is forced to terminate, a *termination handler* will be invoked to perform compensation of all successfully completed scopes nested within this scope. A termination handler can be viewed as a special fault handler that is invoked upon a forced termination event and cannot throw any fault during its performance.

We assume a hierarchy of scopes from Q^1 (innermost) to Q^n (outermost). Let Q^k be any scope from Q^1 to Q^{n-1}, scope Q^k is directly enclosed in scope Q^{k+1}. Therefore, scope Q^1 is nested in scope Q^n to $(n-1)$ depth. If Q^n needs to terminate at some point during its normal performance, all the active scopes (Q^{n-1} to Q^1) nested within Q^n are forced to terminate. If none of the fault handlers have been invoked, the termination handlers (TH^{n-1} to TH^1) for each of these scopes will be executed in outermost-first order, while the termination of the scopes are performed in an innermost-first order.

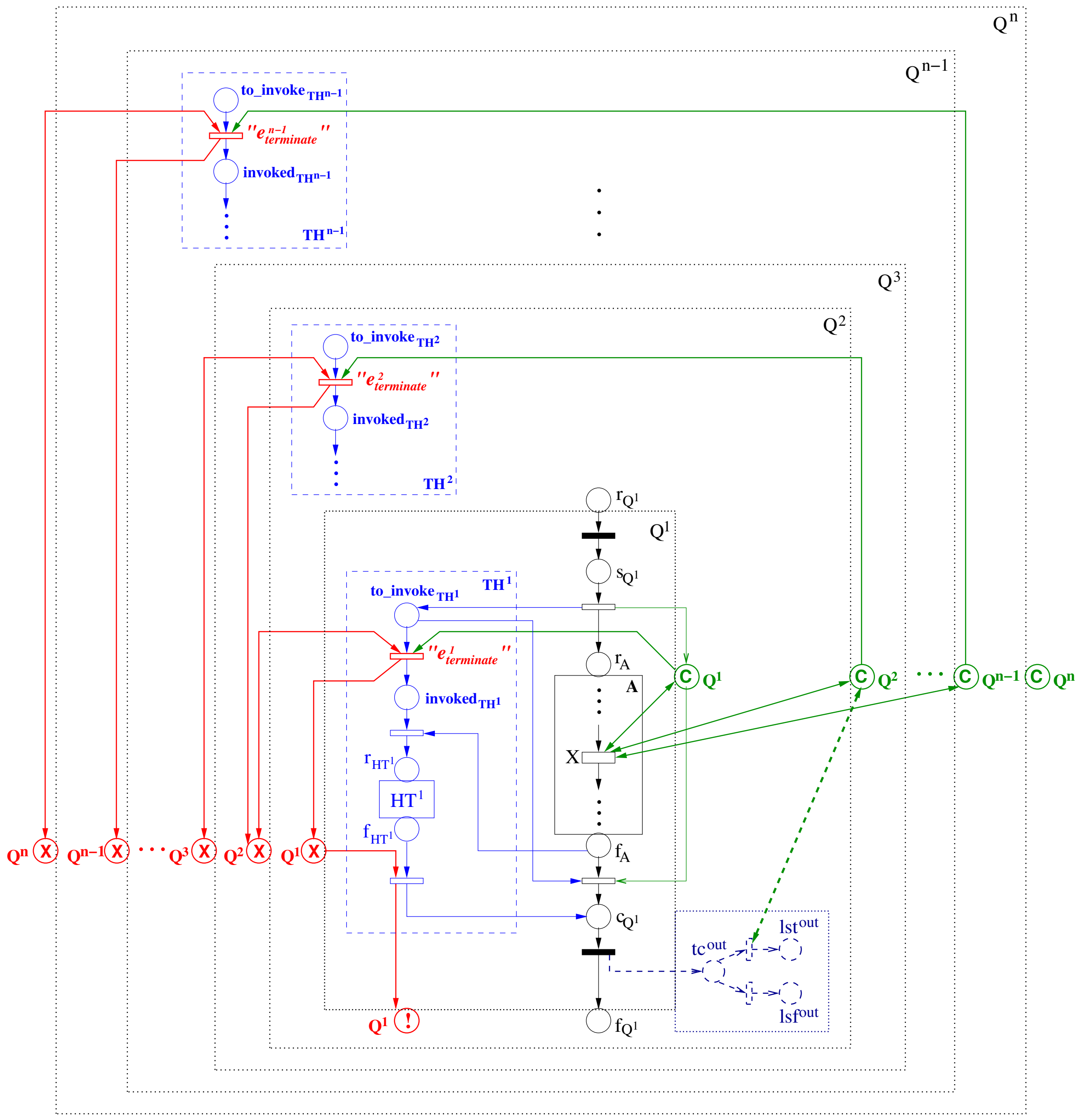
Fig. 23 depicts the mapping for the termination of the above hierarchy of scopes. Once the status of Q^n changes to *to_stop*, a dry run of the rest of the normal process in Q^n is started, and all active scopes among Q^{n-1} to Q^1 are forced to terminate. The subnet enclosed in each dashed box defines the mapping of the termination handler within a scope. It can be viewed as an example of the mapping for a fault handler shown in Fig. 13, where transition "$\mathsf{e}_{\mathsf{fault}}$" is instantiated as "$\mathsf{e}_{\mathsf{terminate}}$". For a given scope Q^k, the termination event $e^k_{terminate}$ is triggered by the termination of its parent scope Q^{k+1}.

In Fig. 23, activity X represents any basic activity that is part of the normal process of scope Q^1 and is directly enclosed in Q^1. The dry run of Q^n will be continued until it reaches X. There are bidirectional arcs connecting transition X with the to_continue place of each of the ancestor scopes of Q^1, so that activity X cannot be executed as long as one of these scopes (e.g., Q^n) is not in the status of *to_continue*. The dry run is then halted waiting to bypass transition X, which requires a token from place $\mathsf{to\_stop}_{\mathsf{Q}^1}$. This place will get marked as a result of the invocation of termination handlers along the hierarchy of scopes from Q^{n-1} (outermost) to Q^1 (innermost). In this way, we ensure that a scope is terminated as soon as one of its ancestor scopes terminates.

Two things need to be mentioned. Firstly, the mapping of bypassing a normal event upon termination of a hierarchy of scopes, is defined in a similar way as for a basic activity shown in Fig. 23. Secondly, any activity or processing of any normal event within a fault handler (including a termination handler) or a compensation handler, will not be affected by the forced termination of the associated scope.

Termination of the process. The termination of a BPEL process is triggered by the execution of an *exit* activity within the process. When the entire process needs to terminate, "all currently running activities MUST be terminated as soon as possible without any fault handling or compensation behavior" (Section 14.6 of [5]). Based on the mapping for scope termination aforementioned, the mapping of the process termination can be obtained as follows:

- Two new flags, *no_exit* and *to_exit*, are defined for the process scope to indicate if it needs to terminate. A process will be in the status of *no_exit* from the beginning to the end, unless an exit activity occurs. The occurrence of the exit activity changes the process status from *no_exit* to *to_exit*.
- The *no_exit* flag must be checked before the core action of any basic activity can be performed or before any normal event can be handled in the process. Note that this applies to all basic activities and normal events in the entire process, including those used by the fault handlers and the compensation handler associated with each scope.



[note] 1. *Q^n is the process scope, and Q^k is directly enclosed in Q^{k+1} $(0<k<n)$* .

2. *X is a basic activity that is directly enclosed in Q^1.*

Fig. 23. Termination of a hierarchy of scopes.

4 Formal Definition of WS-BPEL

In this section, we formally define the syntax and the semantics of WS-BPEL. We firstly introduce boolean function and evaluation function that will be used in the definition. Let f be a boolean function (or propositional statement), $Var(f)$ yields all the propositional variables used in f. Let F be a set of boolean functions and $\mathbb{B}$ be the boolean set $\{\texttt{true}, \texttt{false}\}$, a variable assignment of F is a mapping $assign$: $Var(F) \to \mathbb{B}$, and the set of all possible variable assignments of F is denoted by $Assign(F)$. An evaluation function is a mapping $eval$: $F \times Assign(F) \to \mathbb{B}$.

4.1 Abstract Syntax of WS-BPEL

Definition 1 formally defines an abstract syntax of WS-BPEL.

Definition 1 (WS-BPEL Process Model). *A* WS-BPEL Process Model *is a tuple* $\mathcal{W} = (\mathcal{A}, \mathcal{E}, \mathcal{C}, \mathcal{L}, \mathsf{HR}, \mathsf{type}_{\mathcal{A}}, \mathsf{type}_{\mathcal{E}}, \mathsf{instance}, \mathsf{name}, <_{seq}, <_{swt}, \mathsf{serialscp}, \texttt{process}, \mathsf{trigger}_f, \mathsf{scp}_c, \mathsf{trigger}_c, \mathsf{scp}_t, \mathsf{trigger}_{tf}, \mathsf{LR}, \mathsf{joincon}, \mathsf{supjoinf}, \mathsf{trigger}_{jf})$ *where:*

(∗ *basic elements* ∗)

- $\mathcal{A}$ *is a set of activities,*
- $\mathcal{E}$ *is a set of events,*
- $\mathcal{C}$ *is a set of conditions,*
- $\mathcal{L}$ *is a set of control links,*
- *let* $\mathcal{B} = \mathcal{E} \cup \mathcal{C} \cup \{\perp\}$ *be a set of labels where* $\perp$ *denotes the empty label, then* $\mathsf{HR} \subseteq \mathcal{A} \times \mathcal{B} \times \mathcal{A}$ *is a labeled tree which defines the relation between an activity and its direct sub-activities,*
- $\forall\, a \in \mathcal{A}$, *let* $\mathsf{HR_p}{=}\pi_{1,3}\mathsf{HR}$ *(the projection of* HR *on two activity sets),* $\mathsf{children}(a){=}\{a'{\in}\mathcal{A} \mid \mathsf{HR_p}(a, a')\}$ *is the set of immediate descendants of* a, $\mathsf{descendants}(a){=}\{a'{\in}\mathcal{A} \mid \mathsf{HR}^{+}_{\mathsf{p}}(a, a')\}$ *is the set of all descendants of* a, *and* $\mathsf{clan}(a){=}\{a\}\cup\mathsf{descendants}(a)$ *is the set constituting of* a *and all its descendants,*
- $\mathsf{type}_{\mathcal{A}}$: $\mathcal{A} \to \mathcal{T_A}$ *is a function that assigns types to activities taken from the set of activity types* $\mathcal{T_A} = \{\texttt{sequence}, \texttt{flow}, \texttt{pick}, \texttt{switch}, \texttt{while}, \texttt{scope}, \texttt{throw}, \texttt{compensate}, \texttt{exit}, \texttt{empty}, \texttt{receive}, \texttt{reply}, \texttt{invoke}, \texttt{assign}, \texttt{wait}\}$,
- $\forall\, t \in \mathcal{T_A}$, $\mathcal{A}_t = \{a \in \mathcal{A} \mid \mathsf{type}(a) = t\}$ *is a set of all activities of type* t,
- $\mathsf{type}_{\mathcal{E}}$: $\mathcal{E} \to \mathcal{T_E}$ *is a function that assigns types to events taken from the set of event types* $\mathcal{T_E} = \{\texttt{message}, \texttt{alarm}, \texttt{fault}, \texttt{compensation}, \texttt{termination}\}$.
- $\forall\, t \in \mathcal{T_E}$, $\mathcal{E}_t = \{e \in \mathcal{E} \mid \mathsf{type}(e) = t\}$ *is a set of all events of type* t,
- $\mathsf{instance}$: $\mathcal{A}_{receive} \cup \mathcal{A}_{pick} \to \mathbb{B}$ *is a function which assigns a boolean value to the* `createInstance` *attribute of a receive or a pick activity.*
- name: $\mathcal{A} \to \mathcal{N}$ *is a function assigning names to activities taken from some given set of names* $\mathcal{N}$.

(∗ *activities* ∗)

- *let* $\mathcal{A}^{structured} = \mathcal{A}_{sequence} \cup \mathcal{A}_{flow} \cup \mathcal{A}_{switch} \cup \mathcal{A}_{pick} \cup \mathcal{A}_{while} \cup \mathcal{A}_{scope}$ *be a set of structured activities,* $\forall_{s\in\mathcal{A}^{structured}}(\mathsf{children}(s){\neq}\varnothing)$, *i.e., they are the internal nodes of the* HR *tree,*
- *let* $\mathcal{A}^{basic} = \mathcal{A}_{fault}{\cup}\mathcal{A}_{compensation}{\cup}\mathcal{A}_{receive}{\cup}\mathcal{A}_{other}$ *be a set of basic activities,* $\forall_{s\in\mathcal{A}^{basic}}\, \mathsf{children}(s){=}\varnothing$, *i.e., they are the leaves of the* HR *tree,*
- *given* $\mathcal{A}' = \mathcal{A}_{sequence} \cup \mathcal{A}_{flow}$, $\mathsf{HR} \cap (\mathcal{A}' \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}' \times \{\perp\} \times \mathcal{A})$,
- $\forall\, s \in \mathcal{A}_{sequence}$, $\exists$ *an order* $<^{s}_{seq}$ *which is a strict total order over* $\mathsf{children}(s)$,
- $\mathsf{HR} \cap (\mathcal{A}_{pick} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{pick} \times \mathcal{E}^{normal} \times \mathcal{A})$, *where* $\mathcal{E}^{normal} = \mathcal{E}_{message} \cup \mathcal{E}_{alarm}$ *provides a set of normal events, ,*
- $\forall\, s \in \mathcal{A}_{pick}$, $|\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{message} \times \mathcal{A})| \geqslant 1$, *i.e., a pick activity has at least one message event,*
- *given* $\mathcal{A}' = \mathcal{A}_{switch} \cup \mathcal{A}_{while}$, $\mathsf{HR} \cap (\mathcal{A}' \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}' \times \mathcal{C} \times \mathcal{A})$,
- $\forall\, s \in \mathcal{A}_{switch}$, $\exists$ *an order* $<^{s}_{swt}$ *which is a strict total order over* $\mathsf{children}(s)$,
- $\forall\, s \in \mathcal{A}_{switch}$, *let* $\mathsf{last}(s) \in \mathsf{children}(s)$ *be the sub-activity in the last branch evaluated in* s *such that* $\neg\,\exists_{a\in\mathsf{children}(s)}(\mathsf{last}(s){<}^{s}_{swt}a)$, *let* $c \in \mathcal{C}$, $\mathsf{HR}(s, c, \mathsf{last}(s)){\Rightarrow}\forall_{assign(c)\in Assign(C)}\, eval(c, assign(c)){=}\texttt{true}$. *Note that* $\mathsf{last}(s)$ *represents the* `otherwise` *branch in a switch activity, which ensures that at least one of the branches is taken in the activity,*
- $\forall\, s \in \mathcal{A}_{while}$, $|\mathsf{HR} \cap (\{s\} \times \mathcal{C} \times \mathcal{A})| = 1$, *i.e., each while activity has exactly one sub-activity,*

$(*$ *scopes* $*)$

- $\mathsf{HR} \cap (\mathcal{A}_{scope} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{scope} \times (\mathcal{E} \cup \{\bot\}) \times \mathcal{A})$, *where:* $\forall\, s \in \mathcal{A}_{scope}$,
  - $|\mathsf{HR} \cap (\{s\} \times \{\bot\} \times \mathcal{A})| = 1$, *i.e., each scope has one primary (or main) activity, and therefore* $\mathcal{A}^{mainset}(s) = \{a \in \mathcal{A} \mid \exists_{x\in\mathcal{A}}(\mathsf{HR}(s,\bot,x) \wedge a \in \mathsf{clan}(x))\}$ *is the set constituting of the main activity* x *of scope* s *and all descendants of* x,
  - $|\mathsf{HR} \cap (\{s\}\times\mathcal{E}_{fault}\times\mathcal{A})| \geqslant 1$, *i.e., each scope provides at least one fault handler,*
  - $|\mathsf{HR} \cap (\{s\}\times\mathcal{E}_{compensation}\times\mathcal{A})| \leqslant 1$, *i.e., each scope provides at most one compensation handler,*
  - $|\mathsf{HR} \cap (\{s\}\times\mathcal{E}_{termination}\times\mathcal{A})| \leqslant 1$, *i.e., each scope provides at most one termination handler,*
  - $\forall\, t \in \mathcal{T}_{\mathcal{E}}$, $\mathcal{A}^{t}_{\mathcal{H}}(s) = \{a \in \mathcal{A} \mid \exists_{(e,x)\in\mathcal{E}_t\times\mathcal{A}}(\mathsf{HR}(s,e,x) \wedge a \in \mathsf{clan}(x))\}$ *is the set constituting of the top level activities (represented by* x*) used for handling all events of type* t *for scope* s *and all descendants of these activities,*
  - $\mathcal{A}^{event}_{\mathcal{H}}(s) = \mathcal{A}^{message}_{\mathcal{H}}(s) \cup \mathcal{A}^{alarm}_{\mathcal{H}}(s)$ *is the set of activities used by all event handlers of scope* s*, and therefore* $\mathcal{A}^{normal}(s) = \mathcal{A}^{mainset}(s) \cup \mathcal{A}^{event}_{\mathcal{H}}(s)$ *is the set of all activities that define the normal behaviour of scope* s,
  - $\mathcal{A}^{directenc}(s) = \mathsf{descendants}(s) \setminus (\bigcup_{x\in\mathcal{A}_{scope}\cap\mathsf{children}(s)} \mathsf{descendants}(x))$ *is the set of all activities that are directly enclosed in scope* s,
- $\mathsf{serialscp}$: $\mathcal{A}_{scope} \rightarrow \mathbb{B}$ *is a function assigning a boolean value to the* `variableAccessSerializable` *attribute of a scope.* $\forall_{s\in\mathcal{A}_{scope}}\forall_{a\in\mathsf{descendants}(s)}((\mathsf{serialscp}(s)=\texttt{true} \wedge a\in\mathcal{A}_{scope}) \Rightarrow \mathsf{serialscp}(a)=\texttt{false})$, *i.e., serializable scopes cannot be nested,*
- $\texttt{process} \in \mathcal{A}_{scope}$ *is the root of the* HR *tree, and* $\mathsf{serialscp}(\texttt{process})=\texttt{false}$,

$(*$ *fault handling* $*)$

- $\mathsf{trigger}_{tf}$: $\mathcal{A}_{throw} \rightarrow \mathcal{E}_{fault}$ *is a function which maps each throw activity to a (process-defined) fault event triggered by that activity,*
- $\mathcal{A}^{<re>}_{throw} = \mathcal{A}_{throw} \cap (\bigcup_{s\in\mathcal{A}_{scope}}(\mathcal{A}^{fault}_{\mathcal{H}}(s) \cap \mathcal{A}^{directenc}(s)))$ *is the set of throw activities used to throw faults that cannot be solved during the fault handling. Note that if such a throw activity is used to throw faults that are caught but cannot be solved by the corresponding fault handlers, it may be syntactically named a "rethrow" activity,*
- $\mathcal{A}_{throw} \cap (\mathcal{A}^{fault}_{\mathcal{H}}(\texttt{process}) \cap \mathcal{A}^{directenc}(\texttt{process})) = \varnothing$, *i.e., a fault handler of the process scope cannot throw any fault further,*

$(*$ *compensation* $*)$

- scp_c: $\mathcal{E}_{compensation} \rightarrow \mathcal{A}_{scope}\backslash\{\texttt{process}\}$ *is an injective function mapping a compensation event to a (non-process) scope such that the occurrence of that event invokes the compensation of that scope,*
- $\mathsf{trigger}_c$: $\mathcal{A}_{compensate} \rightarrow \mathcal{E}_{compensation}$ *is an injective function which maps each compensate activity to a compensation event triggered by that activity,*
- $\forall\, s \in \mathcal{A}_{scope}$, $\mathcal{A}_{compensate} \cap (\mathcal{A}^{normal}(s) \cap \mathcal{A}^{directenc}(s)) = \varnothing$, *i.e., a compensate activity cannot be used for the normal behaviour of a scope, that is, it may be used only for exception handling and termination,*
- $\forall\, s \in \mathcal{A}_{scope}$, *let* $\mathcal{A}^{fct}_{\mathcal{H}}(s) = \mathcal{A}^{fault}_{\mathcal{H}}(s) \cup \mathcal{A}^{compensation}_{\mathcal{H}}(s) \cup \mathcal{A}^{termination}_{\mathcal{H}}(s)$, *then* $(a \in \mathcal{A}_{compensate} \cap (\mathcal{A}^{fct}_{\mathcal{H}}(s) \cap \mathcal{A}^{directenc}(s))) \Rightarrow \mathsf{scp}_c(\mathsf{trigger}_c(a)) \in \mathcal{A}^{normal}(s)$, *i.e., a compensate activity is used to invoke compensation of a (descendant) scope nested in the normal process of scope* s *only,*

$(*$ *termination* $*)$

- scp_t: $\mathcal{E}_{termination} \rightarrow \mathcal{A}_{scope}\backslash\{\texttt{process}\}$ *is an injective function mapping a termination event to a scope such that the occurrence of that event invokes the forced termination of that scope,*
- *let* $\mathcal{E}^{ft} = \mathcal{E}_{fault} \cup \mathcal{E}_{termination}$, $\mathsf{trigger}_t$: $\mathcal{E}^{ft} \times \mathcal{A}_{scope} \nrightarrow \mathcal{E}_{termination}$ *is a function which maps a fault or a termination event to another termination event triggered for each inner scope, such that* $dom(\mathsf{trigger}_t) = \{(e,a) \in \mathcal{E}^{ft} \times \mathcal{A}_{scope} \mid \exists_{(s,e,x)\in\mathsf{HR}\cap\mathcal{A}_{scope}\times\mathcal{E}^{ft}\times\mathcal{A}}(a \in \mathsf{clan}(x))\}$,
- $\forall\, s \in \mathcal{A}_{scope}$, $\mathcal{A}_{throw} \cap (\mathcal{A}^{termination}_{\mathcal{H}}(s) \cap \mathcal{A}^{directenc}(s)) = \varnothing$, *i.e., a throw activity cannot be used when handling the termination for a scope. Note that a termination handler is a special type of fault handler that cannot throw any fault unsolved,*

(∗ *control links* ∗)

- $\mathit{LR} \subseteq \mathcal{A} \times \mathcal{L} \times \mathcal{A}$ *is a labeled directed acyclic graph which defines the relation between the source activity of a control link and the target activity of the link,*
- *The boundary crossing restrictions for a control link are defined as follows:*
  - $\forall_{(a,l,a')\in \mathit{LR}}\ (a \notin \mathit{clan}(a'))$, *i.e., a control link cannot connect an activity to any of its ancestors,*
  - $\forall_{s\in\mathcal{A}_{sequence}} \forall_{\{x,x'\}\subseteq \mathit{children}(s)} \forall_{a\in \mathit{clan}(x)} \forall_{a'\in \mathit{clan}(x')} (x <^{s}_{seq} x' \Rightarrow \neg \exists_{l\in\mathcal{L}}\ \mathit{LR}(a', l, a))$, *i.e., in a sequence activity a control link cannot connect a sub-activity or any of its descendants to any preceding sub-activity or any of its descendants,*
  - *let* $\mathcal{A}^{serial}_{scope} = \{s\in\mathcal{A}_{scope} \mid \mathit{serialscp}(s) = \texttt{true}\}$ *be a set of serializable scopes, and* $\mathcal{A}' = \mathcal{A}_{while} \cup \mathcal{A}^{serial}_{scope}$, $\forall_{s\in\mathcal{A}'} \forall_{a\in \mathit{descendants}(s)} \forall_{a'\in\mathcal{A}\setminus \mathit{clan}(s)} (\neg \exists_{l\in\mathcal{L}} (\mathit{LR}(a, l, a') \vee \mathit{LR}(a', l, a)))$, *i.e., a control link cannot cross the boundary of a while activity or a serializable scope,*
  - $\forall\, s\in\mathcal{A}_{scope}$, $\forall\, e\in\mathcal{E}^{normal}$, *let* $\mathcal{A}^{event}_{h}(s, e) = \{a\in\mathcal{A}^{event}_{\mathcal{H}}(s) \mid \exists_{x\in\mathcal{A}^{event}_{\mathcal{H}}(s)} (\mathit{HR}(s, e, x) \wedge a \in \mathit{clan}(x))\}$ *be the set of activities used by an event handler that is triggered upon the occurrence of event e in scope s, then* $\forall_{a\in\mathcal{A}^{event}_{h}(s,e)} \forall_{a'\in\mathcal{A}\setminus\mathcal{A}^{event}_{h}(s,e)} (\neg \exists_{l\in\mathcal{L}}\ (\mathit{LR}(a, l, a') \vee \mathit{LR}(a', l, a)))$, *i.e., a control link cannot cross the boundary of an event handler,*
  - $\forall\, s\in\mathcal{A}_{scope}$, $\forall_{a\in\mathcal{A}^{compensation}_{\mathcal{H}}(s)} \forall_{a'\in\mathcal{A}\setminus\mathcal{A}^{compensation}_{\mathcal{H}}(s)} (\neg \exists_{l\in\mathcal{L}}\ (\mathit{LR}(a, l, a') \vee \mathit{LR}(a', l, a)))$, *i.e., a control link cannot cross the boundary of a compensation handler,*
  - $\forall\, s\in\mathcal{A}_{scope}$, $\forall\, e\in\mathcal{E}^{ft}$, *let* $\mathcal{A}^{ft}_{\mathcal{H}} = \mathcal{A}^{fault}_{\mathcal{H}} \cup \mathcal{A}^{termination}_{\mathcal{H}}$, $\mathcal{A}^{ft}_{h}(s, e) = \{a\in\mathcal{A}^{ft}_{\mathcal{H}}(s) \mid \exists_{x\in\mathcal{A}^{ft}_{\mathcal{H}}(s)} (\mathit{HR}(s, e, x) \wedge a \in \mathit{clan}(x))\}$ *be the set of activities for handling a fault (or termination) event e in scope s, then* $\forall_{a\in\mathcal{A}^{ft}_{h}(s,e)} \forall_{a'\in\mathcal{A}\setminus\mathcal{A}^{ft}_{h}(s,e)}\ (\neg \exists_{l\in\mathcal{L}}\ (\mathit{LR}(a', l, a)))$, *i.e., a control link that crosses the boundary of a fault handler (or termination handler), must be outbound,*
- *let* $\mathcal{A}^{source} = \{a \in \mathcal{A} \mid \exists_{l\in\mathcal{L}}\ ((a, l) \in \pi_{1,2}\mathit{LR})\}$ *be a set of source activities of all control links, and* $\mathcal{A}^{target} = \{a \in \mathcal{A} \mid \exists_{l\in\mathcal{L}}\ ((l, a) \in \pi_{2,3}\mathit{LR})\}$ *be a set of target activities of all control links, then* $\forall\, a \in \mathcal{A}^{source}$, $\mathcal{L}_{out}(a) = \{l \in \mathcal{L} \mid \exists_{a'\in\mathcal{A}}\ \mathit{LR}(a, l, a')\}$ *is a set of all outgoing control links from a, and* $\forall\, a \in \mathcal{A}^{target}$, $\mathcal{L}_{in}(a) = \{l \in \mathcal{L} \mid \exists_{a'\in\mathcal{A}}\ \mathit{LR}(a', l, a)\}$ *is a set of all incoming control links to a,*
- *let* $a \in \mathcal{A}^{target}$, $\mathit{joincon}(a)$, *which expresses the join condition of incoming control links at a, is a boolean function over* $\mathcal{L}_{in}(a)$ *(i.e.* $\mathit{Var}(\mathit{joincon}(a)) = \mathcal{L}_{in}(a)$),
- $\mathit{supjoinf}$: $\mathcal{A} \rightarrow \mathbb{B}$ *is a function assigning a boolean value to the* `suppressJoinFailure` *attribute of each activity,*
- *let* $\mathcal{A}_{SJF} = \{a \in \mathcal{A} \mid \mathit{supjoinf}(a) = \texttt{true}\}$ *and* $\mathcal{A}_{TJF} = \{a \in \mathcal{A} \mid \mathit{supjoinf}(a) = \texttt{false}\}$, *the function* $\mathit{trigger}_{jf}: \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \rightarrow \mathcal{E}_{fault}$ *maps a target activity of which the* `suppressJoinFailure` *attribute is set to* `false`, *to the corresponding join failure fault event.*

4.2 Semantics of WS-BPEL

The following definitions provide auxiliary functions and sets that facilitate the specification of the formal semantics of WS-BPEL:

Definition 2. *Given a WS-BPEL Process Model* $\mathcal{W}$, *we define the following functions to identify the order of activities that occur in a sequence:* $\forall\, s \in \mathcal{A}_{sequence}$,

- $\mathit{head}(s) \in \mathit{children}(s)$ *identifies the first activity to occur in s, such that* $\neg \exists_{a\in \mathit{children}(s)} (a <^{s}_{seq} \mathit{head}(s))$,
- $\mathit{tail}(s) \in \mathit{children}(s)$ *identifies the last activity to occur in s, such that* $\neg \exists_{a\in \mathit{children}(s)} (\mathit{tail}(s) <^{s}_{seq} a)$,
- *the relation* $\lessdot^{s}_{seq} = \{(a, a') \in <^{s}_{seq} \mid \neg \exists_{x\in \mathit{children}(s)} (a <^{s}_{seq} x \wedge x <^{s}_{seq} a')\}$ *is a transitive reduction of* $<^{s}_{seq}$, *i.e., if* $a \lessdot^{s}_{seq} a'$ *then a′ immediately follows a in s.*

Definition 3. *Given a WS-BPEL Process Model* $\mathcal{W}$, $\forall\, a \in \mathcal{A}^{target}$, $\mathit{Var}(\mathit{joincon}(a)) = \mathcal{L}_{in}(a)$. *Let* $\mathsf{b}_a = \mathit{assign}(\mathit{joincon}(a))$, *then* $\mathsf{b}_a \in \mathbb{B}^{\mathcal{L}_{in}(a)}$, *such that* $\forall\, l \in \mathcal{L}_{in}(a)$, $\mathsf{b}_a(l) = \texttt{true}$ *if the status of l is positive, while* $\mathsf{b}_a(l) = \texttt{false}$ *if the status of l is negative.*

Definition 4. *Given a WS-BPEL Process Model* $\mathcal{W}$, *let* $sA \subseteq \mathcal{A}$ *be a subset of activities,* $\mathcal{L}_{OUT}(sA) = \{l \in \mathcal{L} \mid \exists_{a\in sA} \exists_{a'\in\mathcal{A}\setminus sA}\ \mathit{LR}(a, l, a')\}$ *is a set of control links leaving the boundary of the activity set sA.*

Definition 5. *Given a WS-BPEL Process Model* $\mathcal{W}$, *the function* main: $\mathcal{A}_{scope} \rightarrow \mathcal{A}$ *maps each scope activity to its main activity, such that* $\forall_{s\in\mathcal{A}_{scope}}\ \mathit{HR}(s, \bot, \mathit{main}(s))$.

Definition 6. *Given a WS-BPEL Process Model $\mathcal{W}$, $\mathcal{E}^{ft}$ (see Definition 1) is a set of fault and termination events; $\mathcal{E}^{tf} = ran(\mathsf{trigger}_{tf})$ is a set of fault events triggered by throw activities; $\mathcal{E}^{jf} = ran(\mathsf{trigger}_{jf})$ is a set of join failure events; and $\mathcal{E}^{ntfnc} = \mathcal{E} \backslash (\mathcal{E}^{tf} \cup \mathcal{E}_{compensation})$ is a set of non-thrown fault events nor compensation events.*

Definition 7. *Given a WS-BPEL Process Model $\mathcal{W}$, let $s \in \mathcal{A}_{scope}$ and $e \in \mathcal{E}^{ft}$, $\mathcal{A}_h^{ft}(s,e)$ (see Definition 1) is a set of activities for handling a fault event or a termination event e in scope s; $\mathcal{E}_{scp}^{ft}(s) = \{e \in \mathcal{E}^{ft} \mid (s,e) \in \pi_{1,2}\mathsf{HR}\}$ is a set of fault events or termination events associated with scope s; and $\mathcal{L}_{OUT}^{ft}(s) = \bigcup_{e \in \mathcal{E}_{scp}^{ft}(s)} \mathcal{L}_{OUT}(\mathcal{A}_h^{ft}(s,e))$ is a set of control links that leave from the boundary of each of the fault handlers or the termination handler for scope s.*

Definition 8. *Given a WS-BPEL Process Model $\mathcal{W}$, let $s \in \mathcal{A}_{scope}$, $\mathcal{A}_{\mathcal{H}}^{fct}(s)$ (see Definition 1) is a set of activities for handling all exceptions and termination in scope s, and $\mathcal{A}^{nfct}(s) = \mathcal{A}^{normal}(s) \setminus (\bigcup_{x \in \mathcal{A}^{normal}(s) \cap \mathcal{A}_{scope}} \mathcal{A}_{\mathcal{H}}^{fct}(s))$ is a set of activities used for the normal process of both scope s and all scopes nested in s.*

Definition 9 formally defines the semantics of WS-BPEL using Petri nets.

Definition 9 (Petri Net Semantics of WS-BPEL). *Given a WS-BPEL Process Model $\mathcal{W}$, the corresponding labeled Petri net $PN_{\mathcal{W}} = (P_{\mathcal{W}}, T_{\mathcal{W}}, F_{\mathcal{W}}, L_{\mathcal{W}})$ is defined by:*

$P_{\mathcal{W}} = \{r_x \mid x \in \mathcal{A}\} \cup$	*– activity ready*
$\{s_x \mid x \in \mathcal{A}\} \cup$	*– activity started*
$\{c_x \mid x \in \mathcal{A}\} \cup$	*– activity completed*
$\{f_x \mid x \in \mathcal{A}\} \cup$	*– activity finished*
$\{to\_skip_x \mid x \in \mathcal{A}\} \cup$	*– to skip activity*
$\{skipped_x \mid x \in \mathcal{A}\} \cup$	*– skipped activity*
$\{skipping_x \mid x \in \mathcal{A}^{structured}\}$	*– skipping activity*
$\{tc_l \mid l \in \mathcal{L}\} \cup$	*– to evaluate transition condition*
$\{lst_l \mid l \in \mathcal{L}\} \cup$	*– link status true*
$\{lsf_l \mid l \in \mathcal{L}\} \cup$	*– link status false*
$\{jct_x \mid x \in \mathcal{A}^{target}\} \cup$	*– join condition true*
$\{jcf_x \mid x \in \mathcal{A}^{target}\} \cup$	*– join condition false*
$\{jcv_x \mid x \in \mathcal{A}^{target}\} \cup$	*– join condition evaluation value*
$\{to\_f_x \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}\} \cup$	*– to finish activity*
$\{to\_continue_x \mid x \in \mathcal{A}_{scope}\} \cup$	*– to continue scope*
$\{to\_stop_x \mid x \in \mathcal{A}_{scope}\} \cup$	*– to stop scope*
$\{snapshot_x \mid x \in \mathcal{A}_{scope}\} \cup$	*– scope snapshot*
$\{no\_snapshot_x \mid x \in \mathcal{A}_{scope}\} \cup$	*– no scope snapshot*
$\{to\_invoke_{x,e}^{EH} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$	*– ready to invoke event handler*
$\{enabled_{x,e} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$	*– event handler enabled*
$\{to\_invoke_{x,e}^{FH} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$	*– ready to invoke fault handler*
$\{invoked_{x,e}^{FH} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$	*– fault handler invoked*
$\{to\_invoke_x^{CH} \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$	*– intend to invoke compensation*
$\{endc_x \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$	*– end of compensation handling*
$\{to\_invoke_x^{TH} \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$	*– ready to invoke termination*
$\{invoked_x^{TH} \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$	*– termination handler invoked*
$\{to\_c_x \mid x \in \mathcal{A}_{pick}\} \cup$	*– to complete activity*
$\{exiting_x \mid x \in \mathcal{A}_{scope}\} \cup$	*– exiting scope*
$\{to\_exit, no\_exit\}$	*– to or not to exit entire process*
$T_{\mathcal{W}} = \{A_x \mid x \in \mathcal{A}^{basic}\} \cup$	*– basic activity*
$\{PRE_x \mid x \in \mathcal{A}\} \cup$	*– pre-condition evaluation*
$\{PST_x \mid x \in \mathcal{A}\} \cup$	*– post-condition evaluation*
$\{SB_x \mid x \in \mathcal{A}_{sequence} \cup \mathcal{A}_{scope}\} \cup$	*– sequence/scope begin*
$\{SC_{y,y'}^x \mid x \in \mathcal{A}_{sequence} \wedge y <_{seq}^x y'\} \cup$	*– sequence continue*
$\{SE_x \mid x \in \mathcal{A}_{sequence} \cup \mathcal{A}_{scope}\} \cup$	*– sequence/scope end*

$\{AS_x \mid x \in \mathcal{A}_{\mathit{flow}}\} \cup$ – *AND-split*
$\{AJ_x \mid x \in \mathcal{A}_{\mathit{flow}}\} \cup$ – *AND-join*
$\{XS_{x,y} \mid x \in \mathcal{A}_{\mathit{switch}} \wedge y \in \mathsf{children}(x)\} \cup$ – *XOR-split*
$\{XJ_{y,x} \mid x \in \mathcal{A}_{\mathit{switch}} \cup \mathcal{A}_{\mathit{pick}} \wedge y \in \mathsf{children}(x)\} \cup$ – *XOR-join*

$\{LB_x \mid x \in \mathcal{A}_{\mathit{while}}\} \cup$ – *loop begin*
$\{LC_x \mid x \in \mathcal{A}_{\mathit{while}}\} \cup$ – *loop continue*
$\{LE_x \mid x \in \mathcal{A}_{\mathit{while}}\} \cup$ – *loop exit*

$\{SKP_x \mid x \in \mathcal{A}\} \cup$ – *skip activity*
$\{SKPF_x \mid x \in \mathcal{A}^{\mathit{structured}}\} \cup$ – *skipping activity finish*
$\{SKP\_CS^x_{y,y'} \mid x \in \mathcal{A}_{\mathit{sequence}} \wedge y \lessdot^x_{\mathit{seq}} y'\} \cup$ – *skipping continue in sequence*

$\{SET\_LST_l \mid l \in \mathcal{L}\} \cup$ – *set link status to true*
$\{SET\_LSF_l \mid l \in \mathcal{L}\} \cup$ – *set link status to false*
$\{JCE^{\mathsf{b}_x} \mid x \in \mathcal{A}^{\mathit{target}} \wedge \mathsf{b}_x \in \mathbb{B}^{\mathcal{L}_{in}(x)}\} \cup$ – *join condition evaluation*
$\{SJF_x \mid x \in \mathcal{A}^{\mathit{target}} \cap \mathcal{A}_{SJF}\} \cup$ – *suppress join failure*
$\{SJFF_x \mid x \in \mathcal{A}^{\mathit{structured}} \cap \mathcal{A}_{SJF}\} \cup$ – *suppressing join failure finish*
$\{CLT\_JCT_x \mid x \in \mathcal{A}^{\mathit{target}}\} \cup$ – *collect join condition true*
$\{CLT\_JCF_x \mid x \in \mathcal{A}^{\mathit{target}}\} \cup$ – *collect join condition false*

$\{E_{x,e} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap (\mathcal{A}_{\mathit{scope}} \cup \mathcal{A}_{\mathit{pick}}) \times \mathcal{E}^{\mathit{ntfnc}}\} \cup$ – *event of non-thrown fault nor compensation*
$\{HB_{x,e} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A} \times (\mathcal{E} \backslash \mathcal{E}^{\mathit{normal}})\} \cup$ – *handling exception/termination event begin*
$\{HF_{x,e} \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A} \times \mathcal{E}\} \cup$ – *handling any type of event finish*
$\{NOP_x \mid x \in \mathcal{A}_{\mathit{scope}}\} \cup$ – *"no-op" in compensation*

$\{IGN_x \mid x \in \mathcal{A}^{\mathit{target}} \cap \mathcal{A}_{TJF}\} \cup$ – *ignore join failure*
$\{BYP_x \mid x \in \mathcal{A}^{\mathit{basic}} \cup \mathcal{A}_{\mathit{pick}}\} \cup$ – *bypass basic activity or pick*
$\{BYPF_x \mid x \in \mathcal{A}_{\mathit{pick}}\} \cup$ – *bypassing pick finish*
$\{EXT_x \mid x \in \mathcal{A}_{\mathit{scope}}\} \cup$ – *exit scope*
$\{EXTF_x \mid x \in \mathcal{A}_{\mathit{scope}}\}$ – *exited scope*

$L_{\mathcal{W}} = \{(A_x, \mathsf{name}(x)) \mid x \in \mathcal{A}^{\mathit{basic}}\} \cup$ – *labeled transition - basic activity*
$\{(E_{x,e}, e) \mid x \in \mathcal{A}_{\mathit{scope}} \cup \mathcal{A}_{\mathit{pick}} \wedge e \in \mathcal{E}^{\mathit{normal}} \wedge$
$(x,e) \in \pi_{1,2}\mathsf{HR}\} \cup$ – *labeled transition - normal event*
$\{(t, \lambda) \mid t \in T_w \wedge \neg \exists_{x \in \mathcal{A}^{\mathit{basic}}}(t = A_x) \wedge$
$\neg \exists_{(y,e) \in \pi_{1,2}\mathsf{HR} \cap (\mathcal{A}_{\mathit{scope}} \cup \mathcal{A}_{\mathit{pick}}) \times \mathcal{E}^{\mathit{normal}}}(t{=}E_{y,e})\}$ – *unlabeled transition - others*

$F_{\mathcal{W}} = \{(s_x, A_x) \mid x \in \mathcal{A}^{\mathit{basic}}\} \cup \{(A_x, c_x) \mid x \in \mathcal{A}^{\mathit{basic}}\} \cup$ – *basic activity*
$\{(r_x, PRE_x) \mid x \in \mathcal{A}\} \cup \{(PRE_x, s_x) \mid x \in \mathcal{A}\} \cup$ – *activity start*
$\{(c_x, PST_x) \mid x \in \mathcal{A}\} \cup \{(PST_x, f_x) \mid x \in \mathcal{A}\} \cup$ – *activity finish*

$\{(s_x, SB_x) \mid x \in \mathcal{A}_{\mathit{sequence}}\} \cup$ – $(*$ *sequence* $*)$
$\{(SB_x, r_y) \mid x \in \mathcal{A}_{\mathit{sequence}} \wedge y = \mathsf{head}(x)\} \cup$ – *sequence begin*
$\{(f_y, SC^x_{y,y'}) \mid x \in \mathcal{A}_{\mathit{sequence}} \wedge y \lessdot^x_{\mathit{seq}} y'\} \cup$
$\{(SC^x_{y,y'}, r_{y'}) \mid x \in \mathcal{A}_{\mathit{sequence}} \wedge y \lessdot^x_{\mathit{seq}} y'\} \cup$ – *sequence continue*
$\{(f_y, SE_x) \mid x \in \mathcal{A}_{\mathit{sequence}} \wedge y = \mathsf{tail}(x)\} \cup$
$\{(SE_x, c_x) \mid x \in \mathcal{A}_{\mathit{sequence}}\} \cup$ – *sequence end*

$\{(s_x, AS_x) \mid x \in \mathcal{A}_{\mathit{flow}}\} \cup$ – $(*$ *flow* $*)$
$\{(AS_x, r_y) \mid x \in \mathcal{A}_{\mathit{flow}} \wedge y \in \mathsf{children}(x)\} \cup$ – *AND-split*
$\{(f_y, AJ_x) \mid x \in \mathcal{A}_{\mathit{flow}} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(AJ_x, c_x) \mid x \in \mathcal{A}_{\mathit{flow}}\} \cup$ – *AND-join*

$\{(s_x, XS_{x,y}) \mid x \in \mathcal{A}_{\mathit{switch}} \wedge y \in \mathsf{children}(x)\} \cup$ – $(*$ *switch/pick* $*)$
$\{(XS_{x,y}, r_y) \mid x \in \mathcal{A}_{\mathit{switch}} \wedge y \in \mathsf{children}(x)\} \cup$ – *XOR-split*
$\{(s_x, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{\mathit{pick}} \times \mathcal{E}^{\mathit{normal}}\} \cup$
$\{(E_{x,e}, r_y) \mid (x,e,y) \in \mathsf{HR} \cap \mathcal{A}_{\mathit{pick}} \times \mathcal{E}^{\mathit{normal}} \times \mathcal{A}\} \cup$ – *deferred XOR-split*
$\{(XS_{x,y}, \mathit{to\_skip}_{y'}) \mid x \in \mathcal{A}_{\mathit{switch}} \wedge y \in \mathsf{children}(x) \wedge$
$y' \in \mathsf{children}(x) \backslash \{y\}\} \cup$
$\{(E_{x,e}, r_{y'}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{\mathit{pick}} \times \mathcal{E}^{\mathit{normal}} \wedge$
$y' \in \mathsf{children}(x) \wedge (x,e,y') \notin \mathsf{HR}\} \cup$ – *skip unchosen branches*
$\{(\mathit{skipped}_{y'}, XJ_{y,x}) \mid x{\in}\mathcal{A}_{\mathit{switch}}{\cup}\mathcal{A}_{\mathit{pick}} \wedge y{\in}\mathsf{children}(x)$
$\wedge\, y'{\in}\mathsf{children}(x) \backslash \{y\}\} \cup$ – *skipped unchosen branches*
$\{(f_y, XJ_{y,x}) \mid x \in \mathcal{A}_{\mathit{switch}}{\cup}\mathcal{A}_{\mathit{pick}} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(XJ_{y,x}, c_x) \mid x \in \mathcal{A}_{\mathit{switch}}{\cup}\mathcal{A}_{\mathit{pick}} \wedge y \in \mathsf{children}(x)\} \cup$ – *XOR-join*

$\{(s_x, LB_x) \mid x \in \mathcal{A}_{while}\} \cup$	– (∗ *while* ∗)
$\{(LB_x, r_y) \mid x \in \mathcal{A}_{while} \wedge \{y\} = \mathsf{children}(x)\} \cup$	– *loop begin*
$\{(f_y, LC_x) \mid x \in \mathcal{A}_{while} \wedge \{y\} = \mathsf{children}(x)\} \cup$	
$\{(LC_x, s_x) \mid x \in \mathcal{A}_{while}\} \cup$	– *loop continue*
$\{(s_x, LE_x) \mid x \in \mathcal{A}_{while}\} \cup \{(LE_x, c_x) \mid x \in \mathcal{A}_{while}\} \cup$	– *loop end*
$\{(\mathit{to\_skip}_x, SKP_x) \mid x \in \mathcal{A}\} \cup$	– (∗ *skip path* ∗)
$\{(SKP_x, \mathit{skipped}_x) \mid x \in \mathcal{A}^{basic}\} \cup$	– *skipped basic activity*
$\{(SKP_x, \mathit{skipping}_x) \mid x \in \mathcal{A}^{structured}\} \cup$	– *skip structured activity*
$\{(SKP_x, \mathit{to\_skip}_y) \mid x \in \mathcal{A}^{structured}_{nonseq} \wedge y \in \mathsf{children}(x)\} \cup$	
$\{(\mathit{skipped}_y, SKPF_x) \mid x \in \mathcal{A}^{structured}_{nonseq} \wedge y \in \mathsf{children}(x)\} \cup$	– *skipping non-sequence activity*
$\{(SKP_x, \mathit{to\_skip}_y) \mid x \in \mathcal{A}_{sequence} \wedge y = \mathsf{head}(x)\} \cup$	
$\{(\mathit{skipped}_y, SKP\_CS^x_{y,y'}) \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot^x_{seq} y\} \cup$	
$\{(SKP\_CS^x_{y,y'}, \mathit{to\_skip}_{y'}) \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot^x_{seq} y\} \cup$	
$\{(\mathit{skipped}_y, SKPF_x) \mid x \in \mathcal{A}_{sequence} \wedge y = \mathsf{tail}(x)\} \cup$	– *skipping sequence activity*
$\{(\mathit{skipping}_x, SKPF_x) \mid x \in \mathcal{A}^{structured}\} \cup$	
$\{(SKPF_x, \mathit{skipped}_x) \mid x \in \mathcal{A}^{structured}\} \cup$	– *skipped structured activity*
$\{(\mathit{lst}_l, JCE^{\boldsymbol{b}_x}) \mid x \in \mathcal{A}^{target} \wedge l \in \mathcal{L}_{in}(x) \wedge \mathsf{b}_x(l){=}\mathtt{true}\} \cup$	– (∗ *control link* ∗)
$\{(\mathit{lsf}_l, JCE^{\boldsymbol{b}_x}) \mid x \in \mathcal{A}^{target} \wedge l \in \mathcal{L}_{in}(x) \wedge \mathsf{b}_x(l){=}\mathtt{false}\} \cup$	– *join condition evaluation*
$\{(JCE^{\boldsymbol{b}_x}, \mathit{jct}_x) \mid x \in \mathcal{A}^{target} \wedge \mathit{eval}(\mathsf{joincon}(x), \mathsf{b}_x){=}\mathtt{true}\} \cup$	– *join condition to true*
$\{(JCE^{\boldsymbol{b}_x}, \mathit{jct}_x) \mid x{\in}\mathcal{A}^{target} \wedge \mathit{eval}(\mathsf{joincon}(x), \mathsf{b}_x){=}\mathtt{false}\} \cup$	– *join condition to false*
$\{(\mathit{jct}_x, PRE_x) \mid x \in \mathcal{A}^{target}\} \cup$	– *pre-condition positive*
$\{(PST_x, \mathit{tc}_l) \mid x \in \mathcal{A}^{source} \wedge l \in \mathcal{L}_{out}(x)\} \cup$	– *post-condition evaluation*
$\{(\mathit{tc}_l, SET\_LST_l) \mid l \in \mathcal{L}\} \cup \{(SET\_LST_l, \mathit{lst}_l) \mid l \in \mathcal{L}\} \cup$	– *link status to true*
$\{(\mathit{tc}_l, SET\_LSF_l) \mid l \in \mathcal{L}\} \cup \{(SET\_LSF_l, \mathit{lsf}_l) \mid l \in \mathcal{L}\} \cup$	– *link status to false*
$\{(\mathit{jcf}_x, SJF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{SJF}\} \cup$	– *pre-condition negative*
$\{(r_x, SJF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{SJF}\} \cup$	– *suppress join failure* (SJF)
$\{(SJF_x, f_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{basic} \cap \mathcal{A}_{SJF}\} \cup$	– SJF_b (*for basic activity*)
$\{(SJF_x, \mathit{to\_f}_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$	– SJF_s (*for structured activity*)
$\{(SJF_x, \mathit{to\_skip}_y) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}_{nonseq} \cap \mathcal{A}_{SJF} \wedge$ $y \in \mathsf{children}(x)\} \cup$	
$\{(SJF_x, \mathit{to\_skip}_y) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{sequence} \cap \mathcal{A}_{SJF} \wedge$ $y = \mathsf{head}(x)\} \cup$	– SJF_s*: to skip sub-activities*
$\{(\mathit{to\_f}_x, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$	
$\{(\mathit{skipped}_y, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}_{nonseq} \cap \mathcal{A}_{SJF} \wedge$ $y \in \mathsf{children}(x)\} \cup$	
$\{(\mathit{skipped}_y, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{sequence} \cap \mathcal{A}_{SJF} \wedge$ $y = \mathsf{tail}(x)\} \cup$	– SJF_s*: skipped sub-activities*
$\{(SJFF_x, f_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$	– SJF_s *finish*
$\{(SJF_x, \mathit{lsf}_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{basic} \cap \mathcal{A}_{SJF} \wedge l \in \mathcal{L}_{out}(x)\} \cup$	
$\{(SJFF_x, \mathit{lsf}_l) \mid x{\in}\mathcal{A}^{source}{\cap}\mathcal{A}^{structured} \cap \mathcal{A}_{SJF} \wedge l{\in}\mathcal{L}_{out}(x)\} \cup$	– *dead path elimination* (DPE)
$\{(\mathit{to\_skip}_x, CLT\_JCT_x) \mid x \in \mathcal{A}^{target}\} \cup$	– (∗ *skip control link* ∗)
$\{(CLT\_JCT_x, \mathit{to\_skip}_x) \mid x \in \mathcal{A}^{target}\} \cup$	
$\{(\mathit{jct}_x, CLT\_JCT_x) \mid x \in \mathcal{A}^{target}\} \cup$	
$\{(CLT\_JCT_x, \mathit{jcv}_x) \mid x \in \mathcal{A}^{target}\} \cup$	– *collect join condition true*
$\{(\mathit{to\_skip}_x, CLT\_JCF_x) \mid x \in \mathcal{A}^{target}\} \cup$	
$\{(CLT\_JCF_x, \mathit{to\_skip}_x) \mid x \in \mathcal{A}^{target}\} \cup$	
$\{(\mathit{jcf}_x, CLT\_JCF_x) \mid x \in \mathcal{A}^{target}\} \cup$	
$\{(CLT\_JCF_x, \mathit{jcv}_x) \mid x \in \mathcal{A}^{target}\} \cup$	– *collect join condition false*
$\{(\mathit{jcv}_x, SKP_x) \mid x \in \mathcal{A}^{target}\} \cup$	– *skip join condition value*
$\{(SKP_x, \mathit{lsf}_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{basic} \wedge l \in \mathcal{L}_{out}(x)\} \cup$	
$\{(SJFF_x, \mathit{lsf}_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{structured} \wedge l \in \mathcal{L}_{out}(x)\} \cup$	– *DPE in case of skipping*
$\{(s_x, SB_x) \mid x{\in}\mathcal{A}_{scope}\} \cup \{(SB_x, \mathit{to\_continue}_x) \mid x{\in}\mathcal{A}_{scope}\} \cup$	– (∗ *scope* ∗)
$\{(SB_x, r_y) \mid x \in \mathcal{A}_{scope} \wedge y = \mathsf{main}(x)\} \cup$	– *scope begin*
$\{(f_y, SE_x) \mid x \in \mathcal{A}_{scope} \wedge y = \mathsf{main}(x)\} \cup$	
$\{(SE_x, c_x) \mid x \in \mathcal{A}_{scope}\} \cup$	– *scope end*
$\{(\mathit{to\_continue}_x, SE_x) \mid x \in \mathcal{A}_{scope}\} \cup$	
$\{(SE_x, \mathit{snapshot}_x) \mid x \in \mathcal{A}_{scope}\} \cup$	– *to_continue* → *to_stop*
$\{(SKPF_x, \mathit{no\_snapshot}_x) \mid x \in \mathcal{A}_{scope}\} \cup$	– *skipped scope*

$\{(SB_x, \mathit{to\_invoke}^{EH}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$ – *(∗ event handler ∗)*
$\{(\mathit{to\_invoke}^{EH}_{x,e}, SE_x) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$ – *not to invoke EH*
$\{(\mathit{to\_invoke}^{EH}_{x,e}, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$
$\{(E_{x,e}, r_y) \mid (x,e,y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \times \mathcal{A}\} \cup$ – *an instance of EH invoked*
$\{(f_y, \mathit{to\_invoke}^{EH}_{x,e}) \mid (x,e,y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \times \mathcal{A}\} \cup$ – *an instance of EH finish*

$\{(SB_x, \mathit{enabled}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$ – *enable e_{normal}*
$\{(\mathit{enabled}_{x,e}, PST_{y'}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \wedge$
$y' = \mathsf{main}(x)\} \cup$ – *disable e_{normal}*
$\{(\mathit{enabled}_{x,e}, SJF_{y'}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \wedge$
$y' \in \{\mathsf{main}(x)\} \cap \mathcal{A}^{target} \cap \mathcal{A}_{SJF}\} \cup$ – *disable e_{normal} in case of SJF*
$\{(\mathit{enabled}_{x,e}, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$
$\{(E_{x,e}, \mathit{enabled}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$ – *check if e_{normal} is enabled*

$\{(SB_x, \mathit{to\_invoke}^{FH}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$ – *(∗ fault/termination handler ∗)*
$\{(\mathit{to\_invoke}^{FH}_{x,e}, SE_x) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$ – *not to invoke FH*
$\{(\mathit{to\_invoke}^{FH}_{x,e}, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}_{fault} \backslash \mathcal{E}^{tf})\} \cup$
$\{(E_{x,e}, \mathit{invoked}^{FH}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}_{fault} \backslash \mathcal{E}^{tf})\} \cup$ – *general FH invoked*

$\{(\mathit{to\_invoke}^{FH}_{x,e}, A_t) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{tf} \wedge$
$t \in \mathcal{A}_{throw} \wedge \mathsf{trigger}_{tf}(t) = e\} \cup$
$\{(A_t, \mathit{invoked}_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{tf} \wedge$
$t \in \mathcal{A}_{throw} \wedge \mathsf{trigger}_{tf}(t) = e\} \cup$ – *trigger thrown fault event*

$\{(\mathit{jcf}_y, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{jf} \wedge$
$y \in \mathcal{A}^{target} \wedge \mathsf{trigger}_{jf}(y) = e\} \cup$
$\{(r_y, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{jf} \wedge$
$y \in \mathcal{A}^{target} \wedge \mathsf{trigger}_{jf}(y) = e\} \cup$
$\{(E_{x,e}, s_y) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{jf} \wedge$
$y \in \mathcal{A}^{target} \wedge \mathsf{trigger}_{jf}(y) = e\} \cup$ – *trigger join failure event*

$\{(SB_x, \mathit{to\_invoke}^{TH}_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$ – *ready to invoke TH*
$\{(\mathit{to\_invoke}^{TH}_x, SE_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathtt{process}\}\} \cup$ – *not to invoke TH*
$\{(\mathit{to\_stop}_x, E_{y,e}) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{scope} \cap \mathcal{A}^{directenc}(x) \wedge$
$e \in \mathcal{E}_{termination} \wedge (y,e) \in \pi_{1,2}\mathsf{HR}\} \cup$
$\{(E_{y,e}, \mathit{to\_stop}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{scope} \cap \mathcal{A}^{directenc}(x) \wedge$
$e \in \mathcal{E}_{termination} \wedge (y,e) \in \pi_{1,2}\mathsf{HR}\} \cup$ – *trigger termination event*

$\{(f_{y'}, HB_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft} \wedge y' = \mathsf{main}(x)\} \cup$ – *scope terminated*
$\{\mathit{invoked}^{FH}_{x,e}, HB_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$ – *to start FH/TH*
$\{(\mathit{to\_invoke}^{EH}_{x,e}, HB_{x,e'}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \wedge$
$(x,e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$ – *not invoke any more EH*
$\{(\mathit{to\_invoke}^{FH}_{x,e}, HB_{x,e'}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault} \wedge$
$(x,e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}^{ft} \backslash \{e\})\} \cup$ – *not invoke any other FH*
$\{(\mathit{to\_invoke}^{TH}_x, HB_{x,e'}) \mid (x,e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$ – *not invoke TH*
$\{(HB_{x,e}, r_y) \mid (x,e,y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft} \times \mathcal{A}\} \cup$ – *FH/TH start*
$\{(f_y, HF_{x,e}) \mid (x,e,y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft} \times \mathcal{A}\} \cup$ – *FH/TH finish*
$\{(HF_{x,e}, c_x) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$ – *resume scope's normal flow*

$\{(\mathit{to\_continue}_x, E_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}^{ft} \backslash \mathcal{E}^{tf})\} \cup$ – *(∗∗ scope status change ∗∗)*
$\{(E_{x,e}, \mathit{to\_stop}_x) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}^{ft} \backslash \mathcal{E}^{tf})\} \cup$
$\{(\mathit{to\_continue}_x, A_t) \mid x \in \mathcal{A}_{scope} \wedge t \in \mathcal{A}_{throw} \wedge$
$(x, \mathsf{trigger}_{tf}(t)) \in \pi_{1,2}\mathsf{HR}\} \cup$
$\{(A_t, \mathit{to\_stop}_x) \mid x \in \mathcal{A}_{scope} \wedge t \in \mathcal{A}_{throw} \wedge$
$(x, \mathsf{trigger}_{tf}(t)) \in \pi_{1,2}\mathsf{HR}\} \cup$ – *to_continue → to_stop*
$\{(\mathit{to\_continue}_x, SE_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(SE_x, \mathit{snapshot}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – *to_continue → snapshot*
$\{(\mathit{to\_stop}_x, HF_{x,e}) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$
$\{(HF_{x,e}, \mathit{no\_snapshot}_x) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$ – *to_stop → no_snapshot*

$\{(SE_x, \mathit{lsf}_l) \mid x \in \mathcal{A}_{scope} \wedge l \in \mathcal{L}^{ft}_{OUT}(x)\} \cup$ – *DPE: scope completion*
$\{(SKP_x, \mathit{lsf}_l) \mid x \in \mathcal{A}_{scope} \wedge l \in \mathcal{L}^{ft}_{OUT}(x)\} \cup$ – *DPE: skipping scope*
$\{(E_{x,e}, \mathit{lsf}_l) \mid (x,e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}^{ft} \backslash \mathcal{E}^{tf}) \wedge$
$l \in \mathcal{L}^{ft}_{OUT}(x) \backslash \mathcal{L}_{OUT}(\mathcal{A}^{ft}_h(x,e))\} \cup$ – *DPE: non-thrown fault*
$\{(A_t, \mathit{lsf}_l) \mid t \in \mathcal{A}_{throw} \wedge \exists_{x \in \mathcal{A}_{scope}}((x, \mathsf{trigger}_{tf}(t)) \in \pi_{1,2}\mathsf{HR} \wedge$
$l \in \mathcal{L}^{ft}_{OUT}(x) \backslash \mathcal{L}_{OUT}(\mathcal{A}^{ft}_h(x, \mathsf{trigger}_{tf}(t))))\} \cup$ – *DPE: thrown fault*

$\{(A_c, \mathit{to\_invoke}_x^{CH}) \mid c \in \mathcal{A}_{compensate} \wedge x \in \mathcal{A}_{scope} \wedge$ – *(∗ compensation handler ∗)*
$\qquad \mathsf{scp}_c(\mathsf{trigger}_c(c)) = x\} \cup$ – *intend to invoke CH*
$\{(\mathit{snapshot}_x, \mathit{HB}_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$ – *CH is available*
$\{(\mathit{HB}_{x,e}, \mathit{no\_snapshot}_x) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$ – *CH to be unavailable*
$\{(\mathit{to\_invoke}_x^{CH}, \mathit{HB}_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$
$\{(\mathit{HB}_{x,e}, r_y) \mid (x, e, y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation} \times \mathcal{A}\} \cup$ – *CH start*
$\{(f_y, \mathit{HF}_{x,e}) \mid (x, e, y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation} \times \mathcal{A}\} \cup$
$\{(\mathit{HF}_{x,e}, \mathit{endc}_x) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$ – *CH finish*

$\{(\mathit{no\_snapshot}_x, \mathit{NOP}_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathsf{process}\}\} \cup$ – *CH is unavailable*
$\{(\mathit{to\_invoke}_x^{CH}, \mathit{NOP}_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathsf{process}\}\} \cup$
$\{(\mathit{NOP}_x, \mathit{endc}_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathsf{process}\}\} \cup$ – *"no-op" in compensation*
$\{(\mathit{NOP}_x, \mathit{no\_snapshot}_x) \mid x \in \mathcal{A}_{scope} \backslash \{\mathsf{process}\}\} \cup$ – *CH remains unavailable*
$\{(\mathit{endc}_x, \mathit{PST}_c) \mid x \in \mathcal{A}_{scope} \wedge c \in \mathcal{A}_{compensate} \wedge$
$\qquad \mathsf{scp}_c(\mathsf{trigger}_c(c)) = x\} \cup$ – *resume compensate activity*

$\{(\mathit{to\_continue}_x, A_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *(∗ termination ∗)*
$\{(A_y, \mathit{to\_continue}_x) \mid x \in \mathcal{A}_{scope} \wedge$
$\qquad y \in (\mathcal{A}^{basic} \backslash \mathcal{A}_{throw}) \cap \mathcal{A}^{nfct}(x)\} \cup$ – *(∗∗ scope termination ∗∗)*
$\{(A_t, \mathit{to\_continue}_x) \mid x \in \mathcal{A}_{scope} \wedge$
$\qquad t \in \mathcal{A}_{throw} \cap \mathcal{A}^{nfct}(x) \cap (\mathcal{A} \backslash \mathcal{A}^{directenc}(x))\} \cup$ – *can perform basic activity*
$\{(\mathit{to\_continue}_x, \mathit{SET\_LST}_l) \mid x \in \mathcal{A}_{scope} \wedge$
$\qquad \exists_{y \in \mathcal{A}^{nfct}(x) \cap \mathcal{A}^{directenc}(x)}(l \in \mathcal{L}_{out}(y))\} \cup$
$\{(\mathit{SET\_LST}_l, \mathit{to\_continue}_x) \mid x \in \mathcal{A}_{scope} \wedge$
$\qquad \exists_{y \in \mathcal{A}^{nfct}(x) \cap \mathcal{A}^{directenc}(x)}(l \in \mathcal{L}_{out}(y))\} \cup$ – *can set link status to true*

$\{(\mathit{to\_continue}_x, E_{y,e}) \mid x \in \mathcal{A}_{scope} \wedge e \in \mathcal{E}^{normal} \wedge$
$\qquad y \in (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \cap (\{x\} \cup \mathcal{A}^{nfct}(x))$
$\qquad \wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$
$\{(E_{y,e}, \mathit{to\_continue}_x) \mid x \in \mathcal{A}_{scope} \wedge e \in \mathcal{E}^{normal} \wedge$
$\qquad y \in (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \cap (\{x\} \cup \mathcal{A}^{nfct}(x))$
$\qquad \wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$ – *can process e_{normal}*

$\{(\mathit{no\_exit}, A_x) \mid x \in \mathcal{A}_{exit}\} \cup$ – *(∗∗ process termination ∗∗)*
$\{(A_x, \mathit{to\_exit}) \mid x \in \mathcal{A}_{exit}\} \cup$ – *to exit entire process*
$\{(\mathit{no\_exit}, A_y) \mid y \in \mathcal{A}^{basic} \backslash \mathcal{A}_{exit}\} \cup$
$\{(A_y, \mathit{no\_exit}) \mid y \in \mathcal{A}^{basic} \backslash \mathcal{A}_{exit}\} \cup$ – *check exit at basic activity*
$\{(\mathit{no\_exit}, E_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \times \mathcal{E}^{ntfnc}\} \cup$
$\{(E_{x,e}, \mathit{no\_exit}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \times \mathcal{E}^{ntfnc}\} \cup$ – *check exit at event*

$\{(\mathit{to\_stop}_x, \mathit{IGN}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *(∗∗ if throw join failure ∗∗)*
$\{(\mathit{IGN}_y, \mathit{to\_stop}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(\mathit{jcf}_y, \mathit{IGN}_y) \mid \exists_{x \in \mathcal{A}_{scope}}(y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap \mathcal{A}^{nfct}(x))\} \cup$ – *ignore join failure*
$\{(r_y, \mathit{IGN}_y) \mid \exists_{x \in \mathcal{A}_{scope}}(y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap \mathcal{A}^{nfct}(x))\} \cup$
$\{(\mathit{IGN}_y, s_y) \mid \exists_{x \in \mathcal{A}_{scope}}(y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap \mathcal{A}^{nfct}(x))\} \cup$ – *continue dry-run of activity*

$\{(\mathit{to\_stop}_x, \mathit{BYP}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *(∗∗ bypass basic activity ∗∗)*
$\{(\mathit{BYP}_y, \mathit{to\_stop}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *for scope termination*
$\{(\mathit{to\_exit}_x, \mathit{BYP}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$
$\{(\mathit{BYP}_y, \mathit{to\_exit}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$ – *for process termination*
$\{(s_y, \mathit{BYP}_y) \mid y \in \mathcal{A}^{basic}\} \cup \{(\mathit{BYP}_y, c_y) \mid y \in \mathcal{A}^{basic}\} \cup$ – *bypassed basic activity*

$\{(\mathit{to\_stop}_x, \mathit{BYP}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{nfct}(x) \cup$ – *(∗∗ bypass pick ∗∗)*
$\{(\mathit{BYP}_y, \mathit{to\_stop}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{nfct}(x) \cup$ – *for scope termination*
$\{(\mathit{to\_exit}_x, \mathit{BYP}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$
$\{(\mathit{BYP}_y, \mathit{to\_exit}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$ – *for process termination*
$\{(s_y, \mathit{BYP}_y) \mid y \in \mathcal{A}_{pick}\} \cup \{(\mathit{BYP}_y, \mathit{to\_c}_y) \mid y \in \mathcal{A}_{pick}\} \cup$ – *bypassed e_{normal}*
$\{(\mathit{BYP}_y, \mathit{to\_skip}_z) \mid y \in \mathcal{A}_{pick} \wedge z \in \mathsf{children}(y)\} \cup$
$\{(\mathit{skipped}_z, \mathit{BYPF}_y) \mid y \in \mathcal{A}_{pick} \wedge z \in \mathsf{children}(y)\} \cup$ – *skipped all branches*
$\{(\mathit{to\_c}_y, \mathit{BYPF}_y) \mid y \in \mathcal{A}_{pick}\} \cup \{(\mathit{BYPF}_y, c_y) \mid y \in \mathcal{A}_{pick}\} \cup$ – *bypassing finish*

$\{(\mathit{to\_continue}_x, \mathit{LB}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *(∗∗ continue loop in while ∗∗)*
$\{(\mathit{LB}_y, \mathit{to\_continue}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{nfct}(x)\} \cup$ – *for scope termination*
$\{(\mathit{no\_exit}_x, \mathit{LB}_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$
$\{(\mathit{LB}_y, \mathit{no\_exit}_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{fct}_{\mathcal{H}}(x)\} \cup$ – *for process termination*

5 Automated analysis

The output of the mapping from BPEL to Petri nets defined in the previous section can be used to perform formal verification and analysis of BPEL processes on the basis of existing Petri net analysis techniques. The WofBPEL tool, as built using Woflan [21], implements such functionality when coupled with its companion BPEL2PNML tool. The BPEL2PNML tool takes as input an XML file conforming to the BPEL syntax and produces a file conforming to the Petri Net Markup Language (PNML) syntax. This file can then be given as input to the WofBPEL tool which, depending on the selected options, applies a number of analysis methods and produces an XML file describing the analysis results. Both BPEL2PNML and WofBPEL are available under an open-source license at `http://www.bpm.fit.qut.edu.au/projects/babel/tools`.

The current WofBPEL tool can perform three types of analysis:

– Detecting unreachable activities in a BPEL process such as the situation illustrated in Sect. 2.1. This analysis may be performed using two methods as discussed in Sect. 5.2.
– Detecting violations of the BPEL constraint stating that there can never be two simultaneously enabled activities that may consume the same "type of message", where a "type of message" is described by a combination of a partner link, a port type and an operation. Details on how this analysis is performed are given in Sect. 5.3.
– Performing a reachability analysis to determine, for each possible state of a process execution, which types of messages may be consumed in future states of the execution. The result of this analysis can be used for resource management. Rather than keeping a given message in the queue of inbound messages until the message is consumed or the process instance to which the message is associated completes, the message may be discarded as soon as it is detected that no future activity may consume the corresponding type of message. Details of this analysis are given in Sect. 5.4.

Before applying any of the above analysis, WofBPEL performs a simplification of the input net, aiming at removing unnecessary fragments that are introduced because of the way "skipping" is dealt with in the proposed mapping. Details of this net simplification process are presented next.

5.1 Net Simplification

For sake of simplicity and to be consistent in the way structured activity constructs are mapped, we have assumed in our mapping that any activity may be skipped. However, in general, this will not be the case. A straightforward counter example is the root activity in the tree of structured activities, which of course can never be skipped. Another counter-example would be an activity that is not the target of any control link, that is not nested inside any switch or pick activity, nor inside any structured activity that is the target of a control link, that is, an activity that will be executed in every execution of the process if no fault occurs.

Because of our assumption, every activity comes with a number of places and transitions that model the skipping of an activity. However, if the activity cannot be skipped, these places will be unmarkable, and these transitions dead. The unmarkable places and dead transitions might obfuscate results obtained using Petri nets. Therefore, we propose to remove them beforehand. Note that we could have altered our mapping in such a way that it does not generate these places and transitions. The reason for not doing so is that removing them afterwards is more easy.

Fig. 6, Fig. 7(b) and Fig. 8(b) show the skip fragments using dashed lines. The underlying assumption for these figures is that any activity Y that requires another activity X to be skipped, puts a token into $\mathsf{to\_skip}_{\mathsf{X}}$, waits for a token to arrive in place $\mathsf{skipped}_{\mathsf{X}}$, removes that token from that place, and continues. However, if no other activity can put a token into $\mathsf{to\_skip}_{\mathsf{X}}$, then the entire fragment is unmarkable/dead.

Note that Fig. 6, Fig. 7(b) and Fig. 8(b) include two λ-transitions that test whether a $\mathsf{to\_skip}$ place contains a token. These transitions correspond to skipping the activity in case of incoming links, i.e., in case of join condition evaluation. One transition models the situation that the activity needs to be skipped and the join condition has evaluated to true, while the other models the situation where the join condition has evaluated to false. Both transitions will only be generated by our mapping given that incoming links exist.

Thus, the entire skip fragment can and will be removed if the two λ-transitions mentioned above are the only ones that can put tokens into the $\mathsf{to\_skip}$ place. Having removed these redundant skip fragments, we can now use the resulting Petri net to verify certain properties.

5.2 Reachability Analysis: Relaxed Soundness vs. Transition Invariants

The WofBPEL tool can detect unreachable activities in a BPEL process, such as the one illustrated in Fig. 1, for which the corresponding net is shown in Fig. 9. Specifically, WofBPEL is able to detect that there is no run starting from the initial marking and leading to a state where place $\mathsf{jct}_{\mathsf{A}_3}$ in Fig. 9 is marked. As a result, WofBPEL will report that transition A_3, which corresponds to an action with the same name in the original BPEL process, is not reachable from the initial marking. The reason for this is that transition “tt” will never fire. For “tt” to fire, there needs to be a token in both places $\mathsf{lst}_{\mathsf{X}_1}$ and $\mathsf{lst}_{\mathsf{X}_2}$. However, the paths leading from the initial place to these two places are disjoint: an exclusive choice between these paths is made at place s_{SW}. Note that in the mapping defined in [19], there is no equivalent for transition “tt” and places $\mathsf{lst}_{\mathsf{X}_1}$ and $\mathsf{lst}_{\mathsf{X}_2}$, and thus this reachability analysis cannot be performed on the output of the mapping.

To perform this unreachability analysis, WofBPEL relies on two different methods, namely *relaxed soundness* and *transition invariant*. The latter is complete but more computationally expensive than the former. Relaxed soundness [7] takes into account all possible scenarios to get from an initial state (the state with one token in the designated input place) to a final state (the state with one token in the designated output place). Every transition which is covered by any of these scenarios is said to be relaxed sound. On the other hand, transitions that are not covered by these scenarios are called not relaxed sound. If we assume that the goal of the Petri net is to move from the initial state to the final state, then transitions that are not relaxed sound clearly indicate an error, because they cannot contribute in any way to achieving this goal.

However, to check for relaxed soundness we need to compute the full state space of the Petri net, which might take considerable time, especially given the fact that our mapping will generate a lot of parallel behaviour (not that even switch and pick activities are mapped onto parallel behaviour, as the unchosen branches need to be skipped). Therefore, computing relaxed soundness might be a problem.

However, to check for relaxed soundness we need to compute the full state space of the Petri net, which might take considerable time, especially given the fact that our mapping will generate a lot of parallel behaviour (not that even switch and pick activities are mapped onto parallel behaviour, as the unchosen branches need to be skipped). Therefore, computing relaxed soundness might be a problem.

To alleviate this state space problem, we can replace the relaxed soundness by another property, namely transition invariant. Basically, a transition invariant is a multiset of transitions that cancel out, that is, when all transitions from the multiset would be executed simultaneously, then the state would not change. It is straightforward to see that any cycle in the state space has to correspond to some transition invariant. However, not all transitions in the state space will be covered by cycles. For this reason, we add an extra transition that removes a token from the designated output place and puts a token into the designated input place. As a result, every scenario from the initial state to the final state will correspond to a transition invariant, and we can use transition invariants instead of relaxed soundness to get correct results. However, the results using transition invariants are not necessarily complete, because transition invariants might exist that do not correspond to any scenarios in the Petri net. This discrepancy is due to the fact that transition invariants totally abstract from states, they more or less assume that sufficient tokens exist to have every transition executed the appropriate number of times.

5.3 Competing Message-Consuming Activities

The BPEL spec states that “a business process instance MUST NOT simultaneously enable two or more *receive* activities for the same partnerLink, portType, operations and correlation set(s)” (Section 11.4 of [5]), and for the purposes of this constraint, handling message events is equivalent to a receive activity. In other word, activities that can consume the same event may not be simultaneously enabled. Using the full state space mentioned before, we can check this requirement in a straightforward way. Events are considered identical if they have identical partner links, port types and operations. Activities that handle events are receive activities, pick activities, and event handlers.

This property can only be checked if the full state space has been generated. For this property, we could alleviate the possible state space problem by using well-known Petri net reduction rules [18]. Except for the transitions that model the receipt of an event, we could try to reduce every place and every transition before generating the state space.

5.4 Garbage Collection on Queued Events

Again using the full state space, we can compute for each activity a in a BPEL process a set of events E_a such that an event e is in the set of events E_a if and only if it is possible in the state space to consume e after execution of a. In other words, each basic activity a is associated with a set of events E_a such that for each event e in E_a, there exists a run of the process where an activity that consumes event e is executed after activity a. Now, consider the situation where activity a has just been executed, an event e is present in the queue, and e is *not* in E_a. Then event e cannot be consumed anymore (by any activity). Thus, it can be removed from the queue (i.e. it can be garbage collected).

By computing this set for every activity in the BPEL process model off-line, and piggy-backing it with the model (i.e. by adding all event sets to the model) that is handed over to a BPEL engine, the engine can use this information to remove redundant events from its queue and optimise resource consumption. Specifically, the output of the analysis would be an annotated BPEL process where each basic activity is associated with a set of events. After executing an activity a, the BPEL engine could compare the set of events (E_a) associated to a with the current set of events in the queue (E_q) and discard all events in $E_q \backslash E_a$.

6 Conclusions

BPEL is gaining increasing adoption as a process-oriented service composition language, as reflected by the large number of implementations.[5] However, current tools lack the ability to statically detect undesirable situations such as activity unreachability or pairs of activities that may compete for the same message. Also, current BPEL implementations are not optimised with respect to management of inbound messages: a message sent to a given service instance is kept in the queue even when it can be determined that this message will never be consumed. This is because BPEL tools lack the ability to perform reachability analysis.

These limitations can be overcome by translating BPEL process models into Petri nets and applying existing analysis techniques. This paper has presented a mapping from BPEL to Petri nets which is complete in terms of coverage of control flow constructs. In particular, it is the first attempt at providing formal semantics of "join conditions" which can be used to perform reachability analysis on BPEL processes. The mapping has been used as the basis for two open-source tools: BPEL2PNML that translates BPEL code into PNML code and WofBPEL that performs three types of analysis on the generated PNML code and produces output which refers back to the activity names of the original BPEL process.

Our future work aims at extending this mapping to cover communication and data manipulation aspects. By extending the mapping along the communication dimension, it will become possible to check for properties of systems of inter-connected services as opposed to individual services. On the other hand, extending the mapping along the data perspective will allow us to apply simulation techniques to check properties for which static analysis is not suitable. To this end, we plan to use high-level Petri nets or a formally defined process execution language such as YAWL [2].

References

1. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT press, Cambridge, Massachusetts, 2002.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2004.
3. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer-Verlag, 2003.
4. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. *Business Process Execution Language for Web Services Version 1.1.* BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.
5. A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language Version 2.0.* WS-BPEL TC OASIS, February 2005. Available via `http://www.oasis-open.org/committees/download.php/11601/`.
6. F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–162, May 2001.

[5] For a list, see: `http://en.wikipedia.org/wiki/BPEL`.

7. J. Dehnert. *A Methodology for Workflow Modelling: from Business Process Modelling towards Sound Workflow Specification.* PhD thesis, Technische Universität Berlin, Berlin, Germany, August 2003.
8. R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract operational semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2004-03, School of Computer Science, Simon Fraser University, Burnaby B.C. Canada, April 2004.
9. A. Ferrara. Web services: a process algebra approach. In *Proceedings of 2nd International Conference on Service Oriented Computing*, pages 242–251, New York, NY, USA, 2004. ACM Press.
10. J. A Fisteus, L. S. Fernández, and C. D. Kloos. Formal verification of BPEL4WS business collaborations. In *Proceedings of 5th International Conference on Electronic Commerce and Web Technologies (EC-Web'04)*, volume 3180 of *Lecture Notes in Computer Science*, pages 76–85, Zaragoza, Spain, August 2004. Springer-Verlag.
11. H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of Web service composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 152–161, Montreal, Canada, October 2003. IEEE Computer Society.
12. X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of 13th International Conference on World Wide Web*, pages 621–630, New York, NY, USA, 2004. ACM Press.
13. S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri nets. To appear in *Proceedings of 3rd International Conference on Business Process Management*, September 2005.
14. B. Kiepuszewski, A.H.M. ter Hofstede, and W.M.P. van der Aalst. Fundamentals of control flow in workflows. *Acta Informatica*, 39(3):143–209, 2003.
15. M. Koshkina and F. van Breugel. Verification of business processes for Web services. Technical Report CS-2003-11, York University, October 2003. Available via `http://www.cs.yorku.ca/techreports/2003/CS-2003-11.ps`.
16. A. Martens. *Verteilte Geschäftsprozesse - Modellierung und Verifikation mit Hilfe von Web Services (In German).* PhD thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany, 2003.
17. A. Martens. Analyzing Web service based business processes. In M. Cerioli, editor, *Proceedings of 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *Lecture Notes in Computer Science*, pages 19–33. Springer-Verlag, 2005.
18. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
19. C. Stahl. Transformation von BPEL4WS in Petrinetze (In German). Master's thesis, Humboldt University, Berlin, Germany, 2004.
20. H.M.W. Verbeek and W.M.P. van der Aalst. Analyzing BPEL processes using Petri nets. To appear in *Proceedings of 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management*, June 2005.
21. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnozing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
22. P. Wohed, W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede. Analysis of Web services composition languages: The case of BPEL4WS. In I.Y. Song, S.W. Liddle, T.W. Ling, and P. Scheuermann, editors, *Proceedings of 22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215. Springer-Verlag, 2003.