Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 67 (2007) 162–198
Science of Computer Programming
www.elsevier.com/locate/scico

Formal semantics and analysis of control flow in WS-BPEL☆

Chun Ouyang[a,*], Eric Verbeek[b], Wil M.P. van der Aalst[a,b], Stephan Breutel[a], Marlon Dumas[a], Arthur H.M. ter Hofstede[a]

[a] *Faculty of Information Technology, Queensland University of Technology, GPO Box 2434, Brisbane QLD 4001, Australia*
[b] *Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands*

Received 24 February 2006; received in revised form 1 September 2006; accepted 26 March 2007
Available online 5 April 2007

Abstract

Web service composition refers to the creation of new (Web) services by combining functionalities provided by existing ones. 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, synchronization, etc. We present a comprehensive and rigorously defined mapping of BPEL constructs onto Petri net structures, and use this for the analysis of various dynamic properties related to unreachable activities, conflicting messages, garbage collection, conformance checking, and deadlocks and lifelocks in interaction processes. We use a mapping onto Petri nets because this allows us to use existing theoretical results and analysis tools. Unlike approaches based on finite state machines, we do not need to construct the state space, and can use structural analysis (e.g., transition invariants) instead. We have implemented a tool that translates BPEL processes into Petri nets and then applies Petri-net-based analysis techniques. This tool has been tested on different examples, and has been used to answer a variety of questions.

Keywords: Business process modeling; 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 organizational 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 [5]. 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.

☆ This work is supported by the Australian Research Council under the Discovery Grant "*Expressiveness Comparison and Interchange Facilitation between Business Process Execution Languages*".

* Corresponding author. Fax: +61 7 31389390.
E-mail addresses: c.ouyang@qut.edu.au (C. Ouyang), h.m.w.verbeek@tm.tue.nl (H.M.W. Verbeek), w.m.p.v.d.aalst@tm.tue.nl (W.M.P. van der Aalst), sw.breutel@qut.edu.au (S. Breutel), m.dumas@qut.edu.au (M. Dumas), a.terhofstede@qut.edu.au (A.H.M. ter Hofstede).

0167-6423/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.03.002

A number of approaches have been proposed to address these challenges. One such approach, known as (process-oriented) service composition [9], has its roots in workflow and business process management. The idea of service composition is to capture the business logic and behavioral interfaces 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 [6] and recently WS-BPEL [7] (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 [3]. In the area of workflow, it has been shown that Petri nets [26,30] provide an appropriate foundation for static verification. Tools such as Woflan [34] are able to perform state-space-based and transition-invariant-based analysis on workflow models in order to verify properties such as soundness [1]. It is thus natural to presuppose that static analysis can be performed on BPEL processes by translating them to Petri nets and applying existing Petri net-based analysis techniques. In particular, BPEL incorporates two sophisticated branching and synchronization constructs, namely "control links" and "join conditions", which can be found in a class of workflow models known as *synchronizing workflows* formalized in terms of Petri nets in [20].

By using Petri nets as an intermediate representation, we can build upon a large body of theoretical results as well as techniques and tools for verifying properties such as liveness and soundness [1]. In particular, we can use structural analysis techniques such as transition invariants [25] to check certain properties without generating the state space of the process, which can be very large and hinder the scalability of analysis techniques. As discussed later in the paper, other tools for BPEL analysis construct the state space of the process for analysis purposes. Our proposal also leverages on existing liveness and soundness-preserving transformation rules [26] to reduce the size of the generated Petri net prior to analysis.

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

- A complete formalization of all control-flow constructs and communication actions of BPEL in terms of a mapping to Petri nets. This formalization has served to unveil ambiguities in the current BPEL specification, which have been reported to the BPEL standardization committee.[1]
- A tool (WofBPEL) that employs the output of the above mapping to perform three types of analysis. These are: detection of unreachable tasks, detection of potentially conflicting "message receipt" actions, and meta-data generation for garbage collection of unconsumable messages to optimize resource consumption.

In addition, the mapping from BPEL to Petri nets, as proposed in this paper, has also been used for conformance checking between process traces and a BPEL specification defining the desired behavior of the process [2].

The remainder of the paper is organized as follows. Section 2 gives a brief introduction to BPEL. Section 3 presents the mapping from BPEL to Petri nets. Section 4 discusses the analysis of BPEL processes using the WofBPEL tool, and Section 5 presents an empirical evaluation of the tool. The application of the proposed mapping to conformance checking is briefly mentioned in Section 6. Section 7 provides a review of related work on formalization and analysis of BPEL. Finally, Section 8 concludes and outlines future work. A complete formal definition of the mapping from BPEL to Petri nets, including an abstract syntax of BPEL, is given in an appendix attached at the end of the paper.

2. Overview of WS-BPEL

BPEL is designed to support the description of both behavioral service interfaces and executable service-based processes. A *behavioral interface* (known as *abstract process*) is a specification of the behavior of a class of services,

[1] See discussions associated to issues 189, 193 and especially 200 at the OASIS WS-BPEL website http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

capturing constraints on the ordering of messages to be sent to and received from a service. An *executable process* defines the execution order of a set of activities (mostly communication actions), the partners involved in the process, the messages exchanged between partners, and reactions to specific events or faults.

Activities. 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*, signalling a fault in the execution; *compensate*, undoing the effects of already completed activities; *exit*, terminating the entire service instance; and *empty*, doing nothing. *Structured activities* impose behavioral 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 (as described shortly). Structured activities can be nested and combined in arbitrary ways, which enables the presentation of complex structures in a BPEL process.

Control links. The *sequence*, *flow*, *switch*, *pick* and *while* constructs provide means of expressing structured flow dependencies. In addition, BPEL provides another construct known 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 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 iff 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*.[2] 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 [7]. Following our formalization effort, some of these issues were reported and discussed in the BPEL standardization committee, and changes to the specification's wording have been proposed, albeit not yet adopted (see footnote 1).

Also, whilst the control flow constructs of BPEL have been designed in a way to ensure that no "individual" BPEL process execution can deadlock,[3] some combinations of structured activities (in particular *switch* and *pick*) with control links can lead to situations where some activities are "unreachable". Consider the BPEL process definition in Fig. 1 where both the XML code and a graphical representation are provided. During the execution of this process, either A1 or A2 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 x1 or x2 will carry a negative token. On the other hand, we assume that the join condition attached to activity A3 (denoted by keyword "AND") evaluates to true if and only if both links x1 and x2 carry positive values. Hence, this join condition will always evaluate to false, and activity A3 is always skipped (i.e. it is unreachable).

Event, fault and compensation handlers. 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

[2] 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.

[3] Although it has not been formally proved that BPEL processes are deadlock-free, to the best of our knowledge no example of a deadlocking BPEL process has been put forward. Also, Kiepuszewski et al. [20] proves that *synchronizing workflows* (a subset of BPEL processes made up of elementary actions, control links, and restricted forms of join conditions) are non-deadlocking. Here, we refer to "individual BPEL processes" as opposed to "sets of interacting BPEL processes".

```
<process name="unreachableTask">
 <flow name="FL" suppressJoinFailure="yes">
  <links>
    <link name="x1"/>
    <link name="x2"/>
  </links>
  <switch name="SW">
    <case>
     <invoke name="A1">
       <sources> <source linkName="x1"/>  </sources>
     </invoke>
    </case>
    <otherwise>
     <invoke name="A2">
       <sources> <source linkName="x2"/>  </sources>
     </invoke>
    </otherwise>
  </switch>
  <invoke name = "A3">
   <targets>
    <joinCondition>
     bpws:getLinkStatus('x1') and bpws:getLinkStatus('x2')
    </joinCondition>
    <target linkName="x1"/>
    <target linkName="x2"/>
   </targets>
  </invoke>
 </flow>
</process>
```

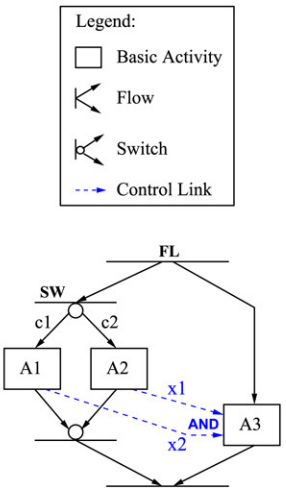


Fig. 1. Example of a BPEL process with an unreachable activity.

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* activity, enable a process to undo the effect of a successfully completed scope. When the compensating 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.

3. Mapping WS-BPEL to Petri nets

In this section, we informally establish a mapping of BPEL onto Petri nets. We use plain Petri nets (i.e. place/transition nets [26,30]), and allow the usage of both labeled and unlabeled transitions in capturing formal semantics of BPEL. The labeled transitions are used to model basic activities (such as receive and invoke actions) and events. The labels encode meta-data which is associated with the basic activities and events (e.g. names of message types sent or received). These labels are used in the analysis techniques outlined in Section 4. The unlabeled transitions, which we hereafter refer to as τ-transitions, represent internal actions that cannot be observed by external users (i.e. silent steps). A complete formal definition of the mapping from BPEL to Petri nets, including an abstract syntax of BPEL, is given in Appendix A.

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, and the other (drawn in dashed lines) models the skipping of X.

In the normal processing part, place 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 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 abstractly labeled X models the action to be performed. This is an abstract way of modeling basic activities, where the core of each activity is considered as an atomic action. Transition X has an input place s_X ("started") for the state when activity X has started, and an output place c_X ("completed") for the state when X is completed. Two τ-transitions (drawn as solid bars) model internal actions for checking pre-conditions or evaluating post-conditions for activities.

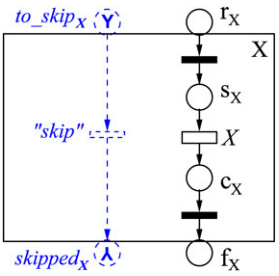


Fig. 2. A basic activity.

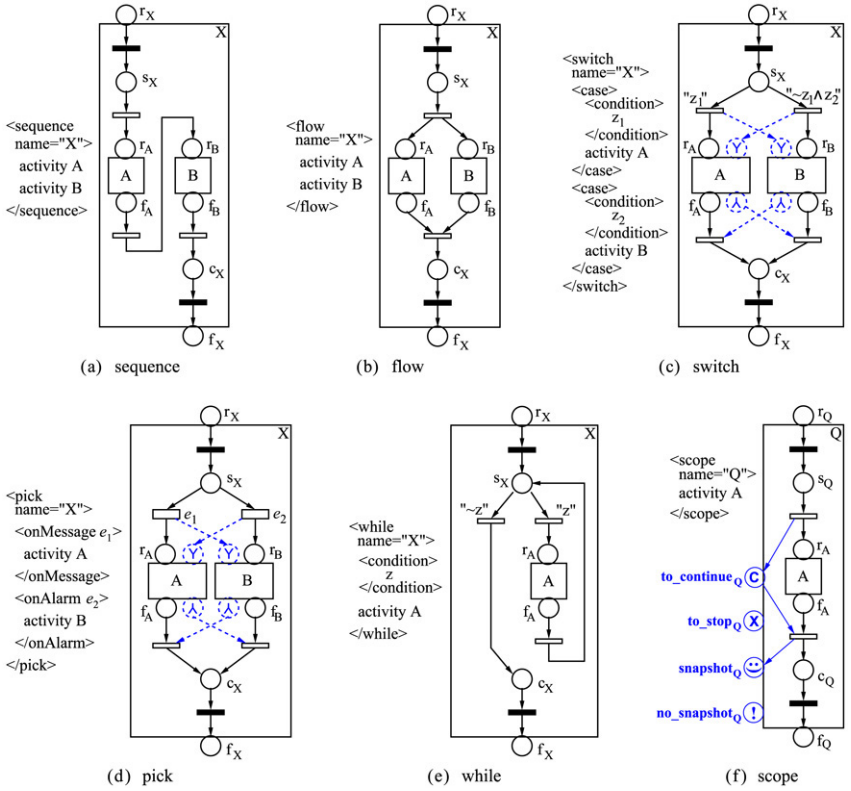


Fig. 3. Structured activities (normal behavior).

The skip path is used to facilitate the mapping of control links, which will be described in Section 3.2. 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. 3 depicts the mapping of structured activities with a focus on their normal behavior. Next to the mapping of each activity is a BPEL snippet of the activity. More τ-transitions (drawn as hollow bars) are introduced for the mapping of routing constructs. In Fig. 3 and subsequent figures, the skip path of the mapping is not shown if it is not used. Also, for simplicity, we use at most two sub-activities for illustration of the mapping of structure activities, and this mapping can be easily generalized to arbitrary number of sub-activities.

A *sequence* activity consists of one or more activities that are executed sequentially. A *flow* activity provides for the 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. Also, among the set of branches in a switch activity, the first

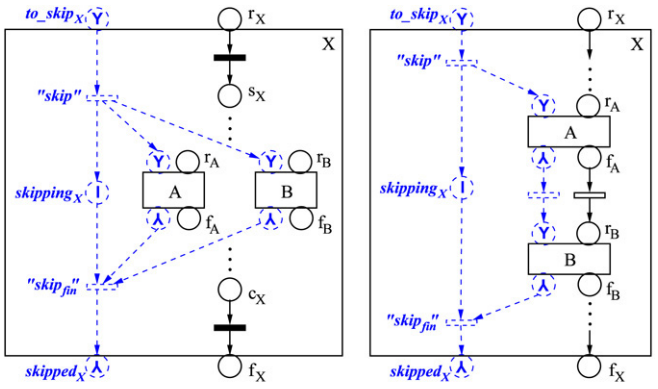


Fig. 4. Skipping structured activities.

branch whose condition holds will be taken. In Fig. 3(c), this is modeled by the two τ-transitions annotated with z_1 or $\sim z_1 \wedge z_2$, where z_1 and z_2 are Boolean expressions capturing conditions for the branches with activity A or B, respectively. Note that these are just annotations (i.e. comments), which are used to improve the readability of the net, and are not used in any analysis. Using plain Petri nets, the choice between executions of conditional branches is modeled non-deterministically.

A *pick* activity exhibits the conditional behavior where decision making is triggered by external events or system timeouts. 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 a timeout. In Fig. 3(d), a pick activity is modeled in a similar way to a switch activity, except for the two transitions, labeled e_1 or e_2, which model the corresponding events. As compared to the two *local* τ-transitions annotated with branching conditions in the mapping of a switch activity (in Fig. 3(c)), the event transitions e_1 and e_2 (in Fig. 3(d)) are *global* transitions enabled upon external or system triggers. Note that external events are part of the environment and system timeouts are concerned with the entire process. Both are not modeled at the level of BPEL activities, and thus are not shown in Fig. 3(d).

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 exit if the condition does not hold any more ($\sim z$).

A *scope* provides event and exception handling. It has a primary activity (i.e. main activity) that defines its normal behavior. To facilitate the mapping of exception handling, we define four flags for a scope, as shown in Fig. 3(f). 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 [7] which refers to the preserved state of a successfully completed uncompensated scope; and no_snapshot, indicating the absence of a scope snapshot.

The mapping of the skipping of an entire structured activity is shown in Fig. 4. To capture the control dependency generated by structural constructs like *sequence*, we define separately the mapping of the skipping of a non-sequence activity (i.e. flow, switch, pick, while, or scope) in Fig. 4(a), and the mapping of the skipping of 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 (A and B) 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. 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, in order to preserve control dependencies between these sub-activities.

3.2. Control links

Control links are non-structural constructs used to express synchronization dependencies between activities. Fig. 5 depicts the mapping of control links using an example of a basic activity. The given BPEL snippet specifies that

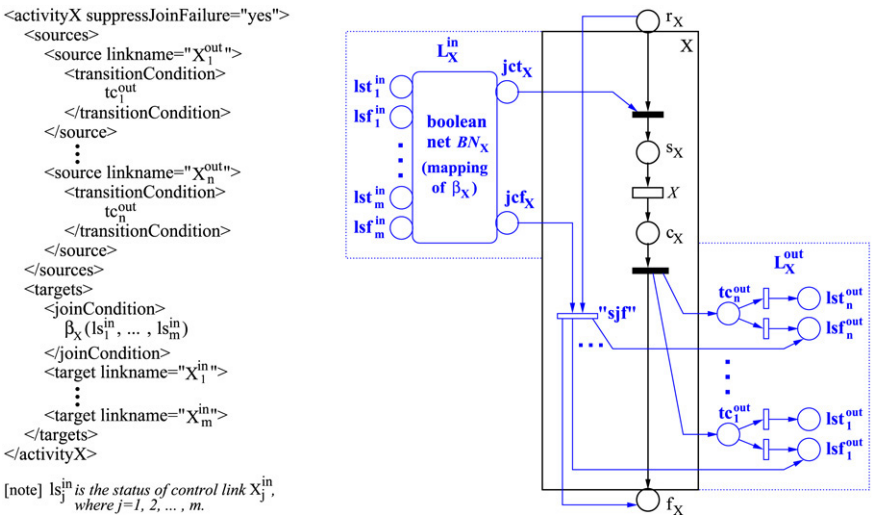


Fig. 5. A basic activity with control links.

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").

The subnet enclosed in the box labeled 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 modeled non-deterministically. Next, the subnet enclosed in the box labeled L_X^{in} specifies the mapping of incoming links to activity X. A *join condition* is defined as a Boolean expression (e.g. $\beta_X(ls_1^{in}, \ldots, ls_m^{in})$) in the set of variables representing the status of each of the incoming links. It is mapped to a Boolean net (BN_X), which takes the status of all incoming links as input and produces an evaluation result as output to place jct_X ("join condition true") or jcf_X ("join condition false"). The definition of this Boolean net is given in Appendix A.2.

If the join condition evaluates to true, activity X can start as normal. Otherwise, a fault of *join failure* occurs. A join failure can be handled in two different ways, as determined by the `suppressJoinFailure` attribute associated with X. If this attribute is set to `yes`, the join failure will be suppressed, as modeled 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, as if it has 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 fault handling procedure (see Section 3.4).

Fig. 6 depicts the mapping of skipping a basic activity with control links, where the skip path is drawn in dashed lines. Such an activity X, if asked to skip, cannot be skipped until the status of all incoming links have been determined. Since the value of incoming link status does not affect the skipping behavior of X, place jcv_X is used to collect either the true or false result of join condition evaluation when X needs to be skipped. In this way, we capture the control dependency between X and the source activity of each of the incoming links to X. As soon as activity X is skipped, the dead path elimination for any outgoing links from X is also captured.

We now extend the mapping of control links to structured activities. This is shown in Fig. 7, which includes mappings for both (a) normal behavior; and (b) skipping behavior of a non-sequence activity. The mapping for a sequence activity can be extended in a similar way. In Fig. 7(a) for the mapping of suppressing a join failure for activity X, place to_f_X is added to capture an intermediate state when X waits for its sub-activities A and B to be skipped, before X enters into the "finished" state (f_X) and initiates the dead path elimination for its outgoing links. The mapping of skipping a structured activity with control links in Fig. 7(b) can be easily followed according to the mapping for a basic activity shown in Fig. 6.

As an example, Fig. 8 depicts the mapping of the BPEL process shown in Fig. 1. The four transitions "tt", "ff", "tf" and "ft", together with their incoming arcs (from places "lst_{x1}", "lsf_{x1}", "lst_{x2}" and "lsf_{x2}") and outgoing arcs (to places

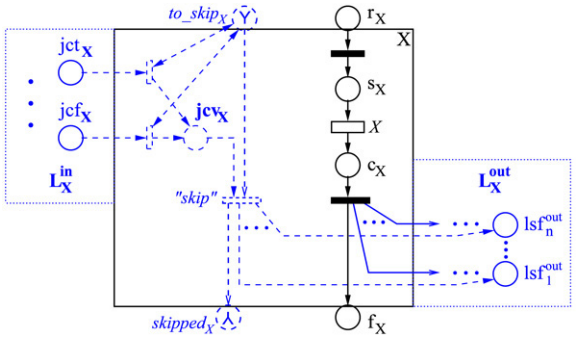


Fig. 6. Skipping a basic activity with control links.

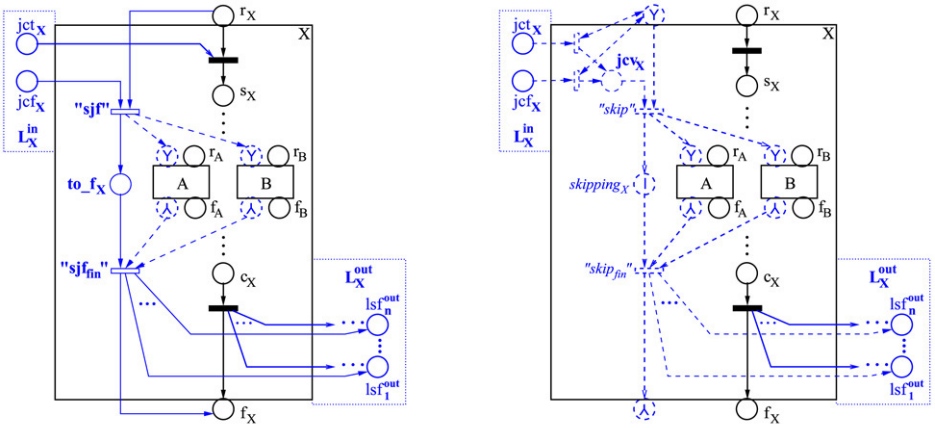


Fig. 7. A (non-sequence) structured activity with control links.

"jct_{A3}" and "jcf_{A3}"), constitute the Boolean net specifying the join condition for activity A3. Note that for simplicity, some skip paths (e.g. the skip path of activity A3) that are not used are not shown in this figure. Also, the execution of the entire Petri net shown in Fig. 8 will be described in Section 4.1.

3.3. 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, and the corresponding event handler can be invoked multiple times as long as the scope is active.

We discuss a couple of decisions made for the mapping of event handlers. First, since no control links are allowed to cross the boundary of event handlers, each event handler can be viewed as an independent unit within a scope. Second, the event handler is invoked if the expected event occurs no matter whether it is a message event or an alarm event. Thus, it is not necessary to distinguish between the mappings of two different types of event handlers.

Fig. 9 depicts the mapping of a scope (Q) with an event handler (EH). The four flags associated with the scope (see Fig. 3(f)) are omitted, as they will not be used in this mapping. 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 e_{normal} is *enabled* and may occur upon an environment or a system trigger. When e_{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 e_{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 e_{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. Hence, by using place enabled in the mapping, we are

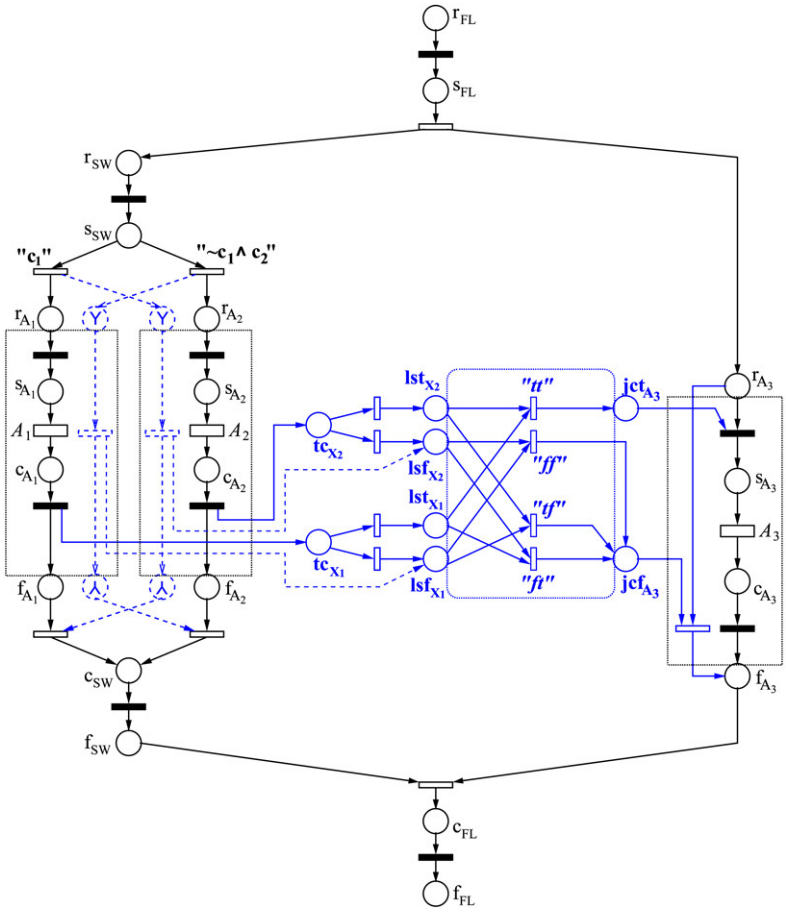


Fig. 8. Mapping of the BPEL process shown in Fig. 1.

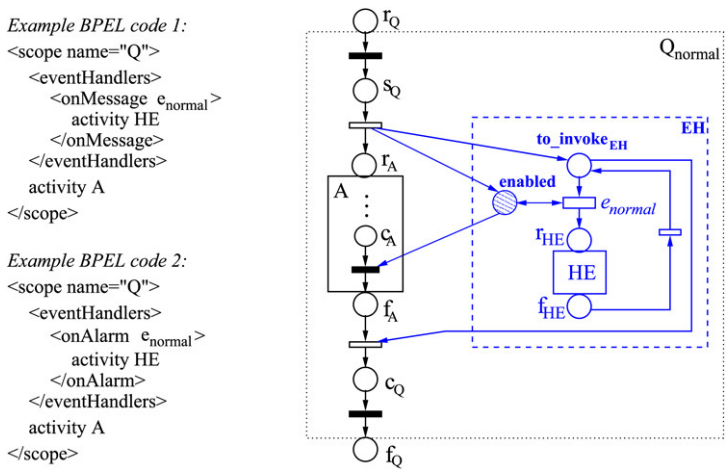


Fig. 9. A scope with an event handler.

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 a case violates the semantics of "disablement of events" defined in Section 13.5.5 of [7].

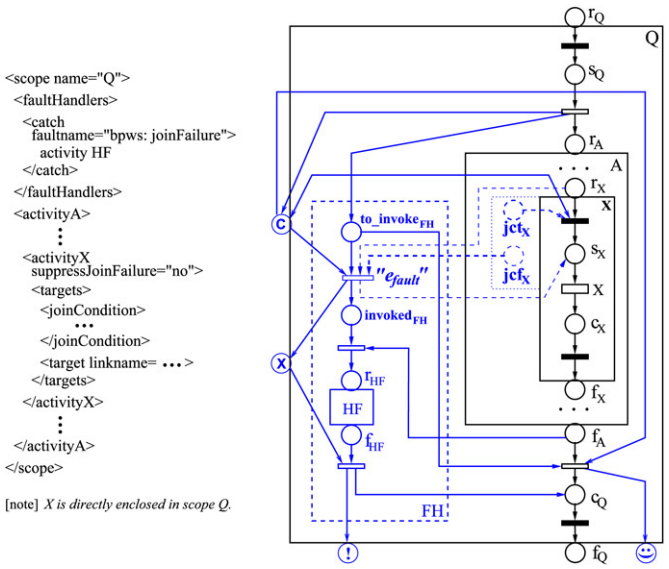


Fig. 10. A fault handler (with dashed arcs and places added for handling a join failure fault).

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 [7]). The mapping in Fig. 9 allows an event handler to have at most *one* active instance at a time. By adding an arc from transition e_{normal} to place $to\_invoked_{EH}$, multiple active instances of an event handler may be invoked simultaneously. This however will cause place $to\_invoked_{EH}$, and thus the net, to be unbounded, and the analysis of an unbounded net is a problem with a high computational complexity. To avoid an unbounded net, we consider another approach in which the subnet enclosed in the box labeled EH is duplicated to n subnets to capture at most n active instances of an event handler simultaneously. This results in a *1-safe net* [30], which can be analyzed based on the maximal number (n) of simultaneously active instances of an event handler allowed in the process. The interested readers may refer to [28] for the details of this net.

3.4. Fault handling

Three types of faults may arise during the process execution [7]. 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.

Each scope has at least one *fault handler*. If a fault occurs during the normal process of a scope, it will be caught by one of the fault handlers within the scope. The scope switches from normal processing mode to fault handling mode. A scope in which a fault has occurred is considered a faulty scope, no matter whether or not the fault can be handled successfully.

Fig. 10 depicts the mapping of a scope (Q) with a fault handler (FH), using the example of a "join failure" fault (see Section 3.2). The subnet enclosed in the box labeled FH, excluding the dashed arcs to/from transition "e_{fault}", specifies the general mapping of a fault handler. It has a similar structure to the mapping of an event handler with the following differences.

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 "e_{fault}" represents a such fault event, and upon its occurrence, the status of scope Q changes from $to\_continue$ to to_stop (see Fig. 3(f) for the definition of these two flags). 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 terminated in the associated scope. This results in an intermediate state, as captured by place $invoked_{FH}$, after the occurrence of e_{fault}, but before the execution of HF. We will not describe the mapping of activity termination here (see

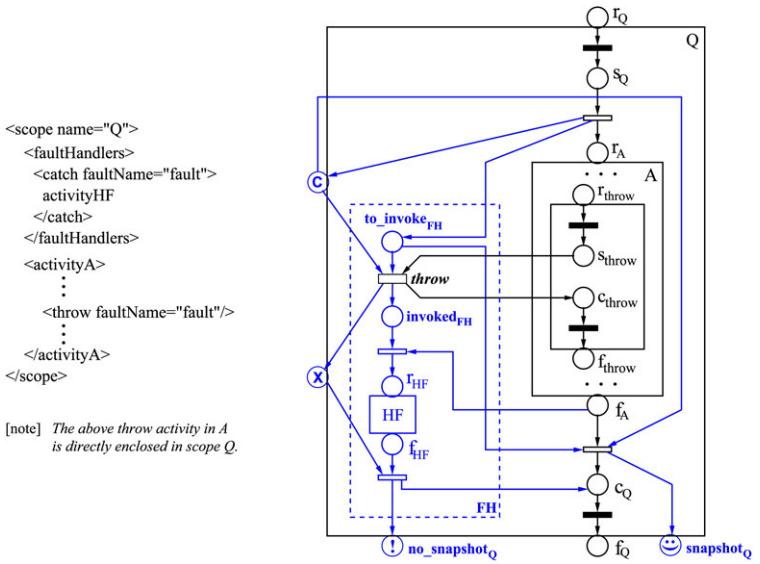


Fig. 11. A fault handler for handling a fault triggered upon the occurrence of a throw activity.

Section 3.6), except to mention that an activity being terminated will end up in the "finished" state. For example, in Fig. 10, if activity A is required to terminate, place f_A will get marked upon termination of A. The arc from place f_A to the input transition of place r_{HF}, ensures that activity HF cannot start until the normal process of scope Q has terminated.

Finally, if the fault has been handled successfully, any control links leaving from scope Q will be evaluated normally. Accordingly, in Fig. 10, place c_Q will get marked, whereas the status of Q will change from to_stop to no_snapshot (see Fig. 3(f) for the definition of these two flags) to indicate that a fault has occurred during Q's normal performance.

We instantiate the general mapping of a fault handler for a "join failure" fault (by adding dashed places and dashed arcs in Fig. 10). The occurrence of a fault event (e_{fault}) will be triggered if the join condition evaluates to false (place jcf_X being marked) and activity X is ready to start (place r_X being marked). The arc from transition "e_{fault}" to place s_X allows us to continue the flow in the normal process of scope Q. This is necessary in the mapping of activity termination (see Section 3.6), which requires a dry run of the uncompleted activities in a scope. Another example, as shown in Fig. 11, is the mapping of a fault handler for handling a fault triggered upon the occurrence of a *throw* activity. In this case, the action of throwing a fault (modeled by transition labeled throw) itself resembles the corresponding fault event.

The above mapping of fault handling can be extended to cover three more complicated situations. The resulting nets from the extended mappings are easy to obtain given the discussions below, and are thus not included here. Interested readers may refer to [28] or Appendix A.2 (attached to the end of this paper) for the details of these nets.

- *Handling faults thrown by fault handlers.* If a fault handler cannot resolve a fault being caught or another fault occurs during the fault handling, both faults need to be (re)thrown to the (parent) scope that directly encloses the current scope, therefore invoking the corresponding fault handlers associated with the parent scope.
- *Handling faults occurred in a scope with event handlers*. Event handlers are considered as part of the normal process of a scope. When a fault occurs in a scope, any active instance of an event handler attached to the scope needs to terminate. The mapping of handling a fault occurring in a scope with event handlers, can be basically obtained by combining the mapping of event handling (Fig. 9) and the general mapping of fault handling (Fig. 10) for the scope. Also, additional arcs are needed to capture that: (a) all instances of the event handler must terminate before the fault handling starts; and (b) no further instantiations of the event handler are allowed once the scope becomes faulty.
- *Dead path elimination for control links leaving from fault handlers*. We assume a fault handler FH within scope Q. There are four possible cases of dead path elimination for control links leaving from FH: (a) FH is invoked, and during its execution it is determined that the status of some of its outgoing links should be set to false; (b) scope Q is completed successfully, in which case, none of the fault handlers in Q can be executed and the status of all

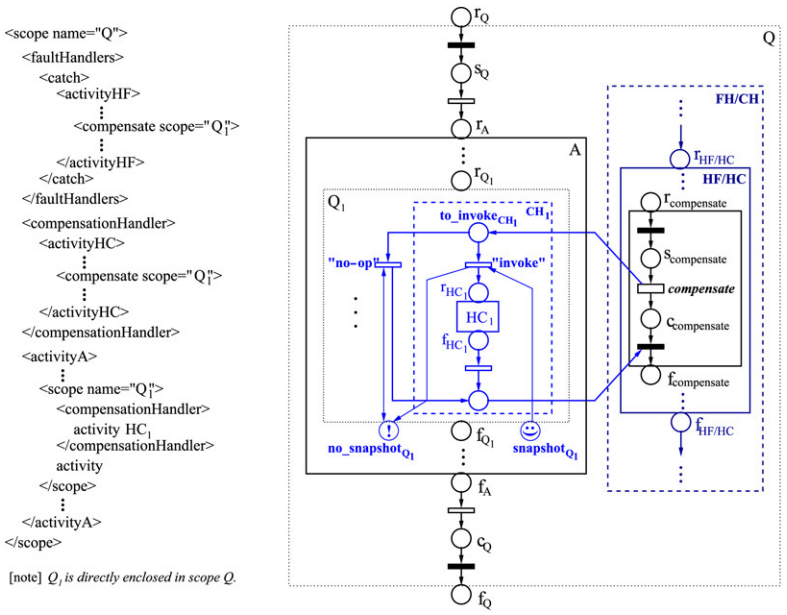


Fig. 12. Handling compensation invoked by the compensate activity.

control links leaving from these fault handlers are set to false upon completion of Q; (c) Q needs to be skipped, and again none of the fault handlers in Q may be executed and all outgoing link status are set to false when Q is being skipped; and (d) another fault handler FH′ is invoked in scope Q, which disables the execution of fault handler FH, and as a result, the status of all outgoing links from FH are set to false upon invocation of FH′.

3.5. Compensation

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 nested within this scope. The compensation handler of a scope is available for invocation only if the scope has a *scope snapshot* (see Section 3.1). Otherwise, the compensation handler will not be available, and invoking a compensation handler that is unavailable is equivalent to an empty activity (i.e. a "no-op" [7]).

Fig. 12 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, transition "invoke" will occur only if Q_1 has a scope snapshot, and 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 removed. 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.

Similarly to fault handlers, a compensation handler may also throw a fault that occurs during compensation. For example, in Fig. 12, a fault that occurs during the execution of activity HC_1, is treated as a fault within the scope (Q_c) that directly encloses the compensate activity used for the compensation of scope Q_1. Hence, if this fault is thrown, it will be caught by the corresponding fault handler within scope Q_c.

3.6. Termination

This subsection presents the mapping for BPEL constructs that terminate (i.e. cancel) the execution of activities: (a) termination of a scope due to a fault and (b) termination of an entire process upon the occurrence of an *exit* activity.

Termination of a scope due to a fault. The main activity of a scope needs to terminate when the scope is faulty. If the fault has been handled successfully, the scope will end as if it has completed normally, and thus the processing

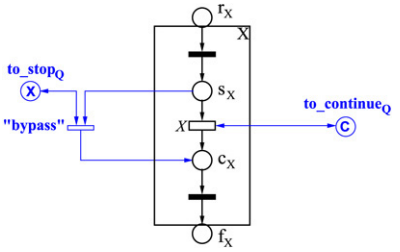
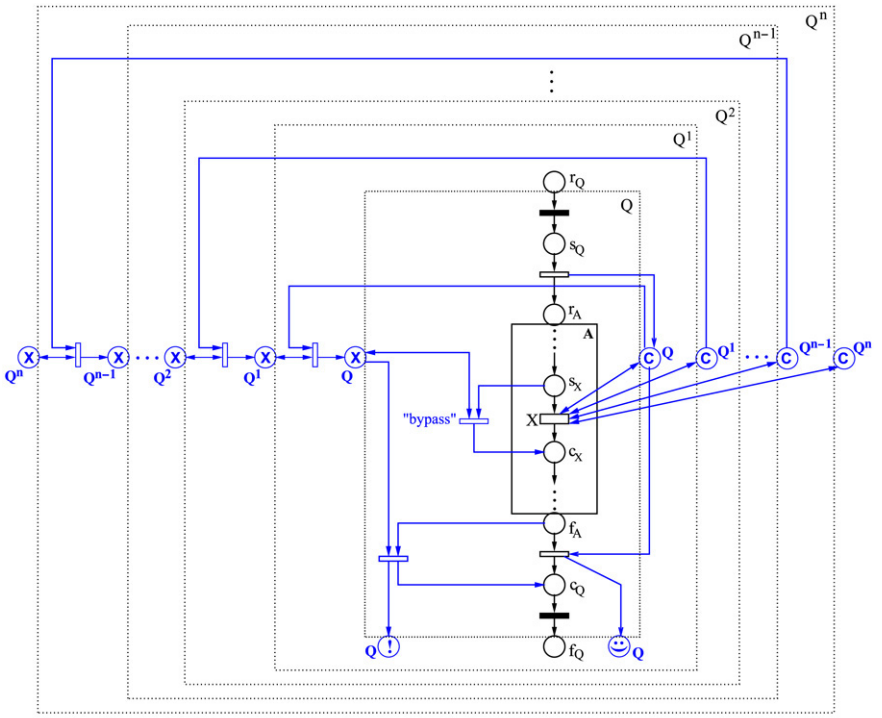


Fig. 13. Termination of a basic activity.



[note] 1. *Q is directly enclosed in Q^1, 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.*

Fig. 14. Termination of a basic activity enclosed within a hierarchy of scopes.

of its parent scope will not be affected. Also, the control dependencies should be preserved. Hence, for the mapping of activity termination, we adopt an approach of conducting a dry run of the activity without performing its concrete actions (i.e. the core action of each basic activity), nor allowing it to process any normal events.

We assume that a basic activity X is directly enclosed in scope Q. The mapping for the termination of activity X is illustrated in Fig. 13. The core action of X (modeled by transition X) can be performed only if scope Q is allowed to continue its normal process. Otherwise, if Q needs to stop, the core action of X is bypassed, as captured by the τ-transition "bypass". The mapping of bypassing a normal event can be defined in a similar way. As mentioned in Section 3.4, when a fault occurs in scope Q, the status of Q changes from *to_continue* to *to_stop*.

We now consider that the above scope Q is enclosed in a hierarchy of scopes from Q^1 (innermost) to Q^n (outermost). Every basic activity (e.g. activity X) that is directly enclosed in Q, needs to check if each of its ancestor scopes is allowed to continue or needs to stop, i.e., if a token is present in either the place to_continue or to_stop. Suppose that scope Q^n is required to terminate. When a token is put in the to_stop place of Q^n, a propagation process takes place whereby for each scope Q_i nested within Q^n (i.e. $Q_i \in \{Q, Q^1, Q^2, \ldots, Q^{n-1}\}$), the token in the to_continue place of Q_i is moved to the to_stop place of Q_i. This propagation process is schematically illustrated in Fig. 14.

Two things are worth noting. First, if a scope Q is already faulty when one of its ancestor scopes needs to terminate, "the already active fault handler (for Q) is allowed to complete" (Section 13.4.4 of [7]). This means that, in such a

case, the execution of a basic activity or the processing of a normal event within a fault handler will not be affected. Second, BPEL [7] defines for each scope a *termination handler*, which will be invoked if that scope is in normal process but is forced to terminate as one of its ancestor scopes needs to terminate due to a fault. The termination handler for scope Q is responsible for the compensation of all successfully completed scopes nested within Q. It may be viewed as a special fault handler that cannot throw any fault during its performance. The mapping shown in Fig. 14 can be easily extended with a subnet for each scope, which captures the semantics of the termination handler for that scope. Interested readers may refer to [28] or Appendix A.2 for more details.

Termination of a process due to an exit activity. In BPEL, the termination of an entire process is triggered by the execution of an exit activity within the process. When the 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 [7]). The process termination can be mapped 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, which changes the process status from *no_exit* to *to_exit*.
- Every basic activity or normal event needs to check for the presence of a token in the place no_exit or to_exit (which represents the corresponding flag), and will be bypassed if a token is present in the to_exit place. This applies to all basic activities and normal events in the entire process, including those within the fault handlers and the compensation handlers.

3.7. Interacting processes

The mapping presented in the previous subsections focuses on an individual BPEL process. Given that Petri nets are a formalism suitable for modeling communication aspects [8], it is not difficult to extend the mapping to capture the behavior of interacting BPEL processes.

In BPEL, there are three types of communication actions: *invoke*, *receive* and *reply*. An invoke activity may refer to: (a) an asynchronous "one-way" operation, in which case, the invoke represents an individual send action; or (b) a synchronous "request-response" operation, in which case, the invoke represents a pair of send and receive actions where a sender makes a request to a receiver and waits for a response before continuing to process. A receive activity allows the process to block and wait for a matching message to arrive, whilst a reply activity is used to send a response to a request that was previously accepted via a receive activity. Also, a message receipt event that triggers either an `onMessage` branch of a pick activity or an `onEvent` (message) event handler, is a *receive*-like construct and is thereby treated in much the same manner as a receive activity.

As an example, Fig. 15 shows a Petri net model of two interacting processes P1 and P2. Process P1 has an invoke activity (A) which invokes a "request-response" operation on process P2, and the operation corresponds to a couple of receive (B1) and reply (B2) activities in P2. Each of the three activities, A, B1 and B2, can be mapped to a Petri net in a similar way as illustrated in Fig. 2. The invoke activity A consists of two atomic actions: one for sending the request (modeled by transition invoke_s), and the other for receiving the response (modeled by transition invoke_r). The place w_A ("waiting") between transitions invoke_s and invoke_r models an intermediate state when P1 waits for the response from P2 after the request has been sent to P2. The interaction between P1 and P2 is captured by two places called request and response. The request place, with an incoming arc from transition invoke_s and an outgoing arc to transition receive, is used to convey the request from process P1 to P2, upon the execution of activity A. Similarly, the response place, with an incoming arc from transition reply and an outgoing arc to transition invoke_r, is used to convey the response from P2 to P1, upon the execution of activity B2. Finally, two transitions (drawn in dashed borders), namely request_lost and response_lost, model the loss of messages over a lossy communication medium.

4. 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 [29], built using *Woflan* [34], implements such a functionality when coupled with its companion *BPEL2PNML* tool. Fig. 16 depicts the role of WofBPEL and BPEL2PNML in the analysis of BPEL processes. The BPEL process

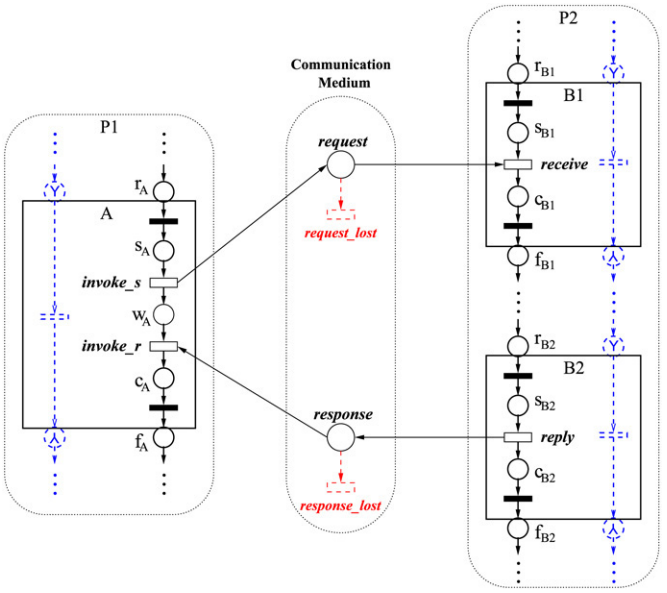


Fig. 15. A Petri net model of two interacting BPEL processes.

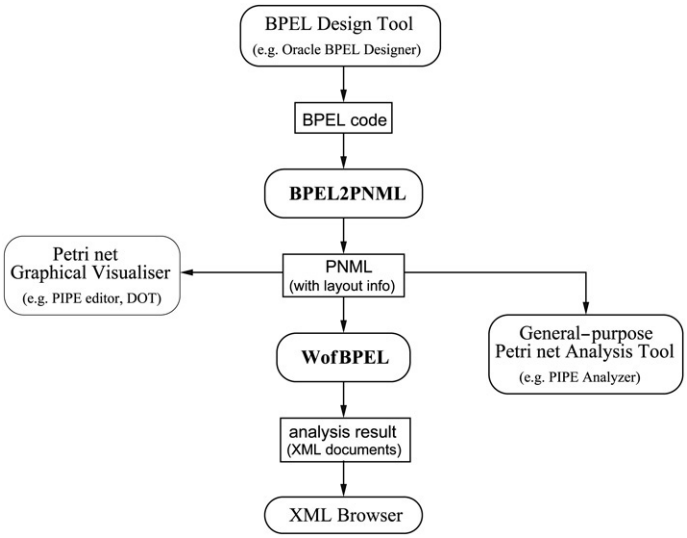


Fig. 16. Analyzing BPEL processes using WofBPEL and BPEL2PNML.

code may be manually written or generated from a BPEL design tool, e.g. Oracle's BPEL Designer. BPEL2PNML takes as input the BPEL code and produces a file conforming to the Petri Net Markup Language (PNML) syntax. This file can be given as input to WofBPEL which, depending on the selected options, applies a number of analysis methods and produces an XML file describing the analysis results. It may also be used as input to any general-purpose Petri net analysis tool, e.g. PIPE.[4] In addition, the PNML file obtained as the output from BPEL2PNML also includes layout information, and can thus be used to generate a graphical view of the corresponding Petri nets. 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 the following three types of analysis:

– Detecting unreachable activities in a BPEL process, such as the situation illustrated in Section 2. This analysis may be performed using two methods as discussed in Section 4.1.

[4] https://sourceforge.net/projects/petri-net.

– 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 Section 4.2.
– 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 by a BPEL engine 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 so that no future activity may consume the corresponding type of message. Details of this analysis are given in Section 4.3.

The above three types of analysis are concerned with analysis of individual processes. In addition, our mapping from BPEL to Petri nets can also be used to analyze interacting processes. This issue is discussed in Section 4.4.

4.1. Detection of unreachable activities

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. 8. Specifically, WofBPEL is able to detect that all possible runs starting from the initial state (represented by the marking with one token in the designated input place, e.g. r_{FL} in Fig. 8) and leading to the desired final state (represented by the marking with one token in the designated output place, e.g. f_{FL} in Fig. 8). If we assume that the goal of the Petri net is to move from the initial state to the desired final state, then transitions that are not covered by any runs clearly indicate an error, because they cannot contribute in any way to achieving this goal. As a result, WofBPEL will report that activity A3 in the original BPEL process (as modeled by transition A3 in Fig. 8) does not appear in any possible run starting from the initial state to the desired final state (i.e. it is unreachable). The reason for this is that transition "tt" will never fire. For "tt" to fire, there needs to be a token in both places lst_{x1} and lst_{x2}. However, the paths leading from the initial place (r_{FL}) to these two places are disjoint: an exclusive choice between these paths is made at place s_{SW}.

To perform this unreachability analysis, WofBPEL relies on two different methods, namely *relaxed soundness* and *transition invariants*. The former is complete, but more computationally expensive than the latter. Relaxed soundness [10] takes into account all possible runs to get from the initial state to the desired final state. Every transition which is covered by any of these runs is said to be relaxed sound. On the other hand, transitions that are not covered by these runs are called not relaxed sound. 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 behavior. Even switch and pick activities are mapped onto parallel behavior, as the unchosen branches need to be skipped. Hence, computing relaxed soundness might be a problem. To alleviate this state space problem, we can replace the relaxed soundness by transition invariants.

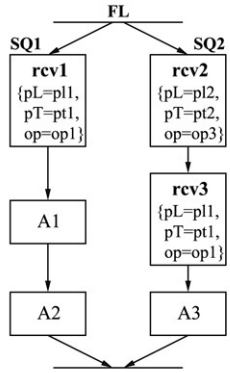
Transition invariants have been experimentally shown to outperform state space methods and are able to deal with complex processes [33]. Basically, a transition invariant is a multiset of transitions that cancel out; that is, when all transitions from the multiset are executed simultaneously, then the state does 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 run from the initial state to the desired 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 runs 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.

4.2. Detection of conflicting message-consuming activities

The BPEL specification [7] 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 [7]).[5] In

[5] For the purposes of this constraint, `onMessage` branches of a pick activity and `onEvent` (message) event handlers are equivalent to a receive activity.

```
<process name="competingMessages"
        targetNamespace="http://samples.otn.com"
        suppressJoinFailure="yes"
        xmlns:tns="http://samples.otn.com"
        xmlns:services="http://services.otn.com"
        xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">
  <flow name="FL" suppressJoinFailure="yes">
    <sequence name="SQ1">
      <receive name="rcv1" partnerLink="pl1" portType="pt1" operation="op1"/>
      <wait name="A1"> <until> '2005-09-01T00:00+01:00' </until> </wait>
      <invoke name="A2" partnerLink="pl2" portType="pt2" operation="op2"/>
    </sequence>
    <sequence name="SQ2">
      <receive name="rcv2" partnerLink="pl2" portType="pt2" operation="op3"/>
      <receive name="rcv3" partnerLink="pl1" portType="pt1" operation="op1"/>
      <reply name="A3" partnerLink="pl1" portType="pt1" operation="op2"/>
    </sequence>
  </flow>
</process>
```



[Abbr. : pL – partnerLink, pT – portType, op – operation]

Fig. 17. Example of a BPEL process with conflicting receive activities.

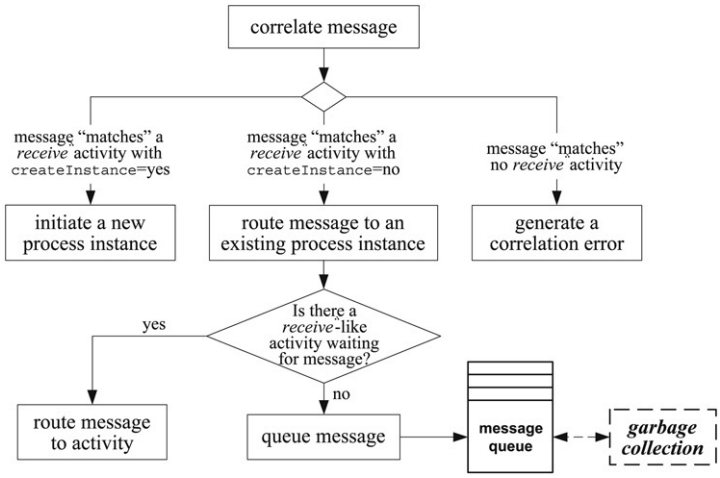


Fig. 18. A flow chart illustrating the message processing mechanism in a BPEL engine.

other words, activities that can consume the same message event may not be simultaneously enabled. Message events are considered the same if they have identical partner links, port types, operations, and optionally identical correlation sets. Activities that handle message events are receive activities, pick activities, and event handlers. Fig. 17 depicts an example of a BPEL process which involves two conflicting receive activities *rcv1* and *rcv3*. Correlation sets are not used in this example, and thus each message event is identified by a partner link, a port type and an operation.

To check this property, it is necessary to generate the full state space S_F. Then we can check for each $s \in S_F$ whether there exist (at least) two concurrently enabled transitions that represent the same message event. For example, in Fig. 17, the two receive activities *rcv1* and *rcv3* may be simultaneously enabled when the sequence activity SQ1 is about to start, and in SQ2 the execution of activity *rcv2* has completed. Also, for this property, we could alleviate the possible state space problem by using well-known Petri net reduction rules [26] (see Section 5.1). Except for the transitions that model the receipt of a message event, we could try to reduce every place and every transition before generating the state space.

4.3. Garbage collection of queued messages

Fig. 18 provides a flow chart illustrating message processing mechanisms in a BPEL engine. Upon each message's arrival, the BPEL engine routes the message to the correct process instance using BPEL's correlation mechanism. For simplicity, if the message has a correlation set that carries tokens with the same values as those specified by a receive activity in a process, we say the message "matches" that receive activity. The receive activity may be a start activity in the process, as indicated by its `createInstance` attribute set to `yes`. In this case, the occurrence of the activity

initiates a new instance of the process and consumes the message. In the second case, if the message "matches" a receive activity with `createInstance` set to `no`, it is routed to the corresponding existing process instance. The engine then checks whether there is a *receive*-like activity (i.e. receive, pick, or message event handler) waiting for the message. If so, the message is directed to the activity. Otherwise, the message is stored in a queue for possible use by further activities in the process instance. As the last case, if the message "matches" no receive activities, the engine generates a correlation error and discards the message.

Now, we look further into the second case. Consider the situation where a message stored in the queue can never be consumed in a running process instance. For example, this may result from the fact that the corresponding *receive*-like activity is on a conditional branch which is not taken in the process instance. In the BPEL engine, the queue of inbound messages is normally retained until the process instance to which the message is associated completes. However, from the long-run viewpoint of resource management, we would like to discard (i.e. garbage-collect) the message as soon as it is detected that no further activity may consume the corresponding type of message. Below, we discuss our mechanism for garbage collection on queued messages.

Again using the full state space, we can compute, for each activity a in a BPEL process, a set of message types MT_a such that a message type mt is in MT_a iff it is possible in the state space to consume a message of type mt after execution of a. In other words, each basic activity a is associated with a set of message types MT_a such that for each $mt \in MT_a$, there exists a run of the process where an activity that consumes a message of type mt is executed after a. Assume that activity a has just been executed, a message m is present in the queue, and the type of m is *not* in MT_a. As a result, message m cannot be consumed anymore (by any activity). Thus, it can be removed from the queue (i.e. it can be garbage collected).

By computing the set of message types for every basic activity in the BPEL process model, and piggy-backing it in the process definition that is handed over to a BPEL engine, the engine can use this information to remove redundant messages from its queue, thus optimizing resource consumption. Accordingly, we built a post-processor to link WofBPEL with a BPEL engine. This post processor, which is also available at http://www.bpm.fit.qut.edu.au/projects/babel/tools, takes as input the original BPEL process code and the corresponding output from WofBPEL, and produces an annotated version of the BPEL process. In this annotated BPEL process, each basic activity a is associated with a set of message types (identified by a partner link, a port type, an operation and optionally a correlation set) corresponding to MT_a. Given these annotations, the BPEL engine can, after executing activity a, compare the set of message types (MT_a) associated to a with the current set of messages in the queue (M_q) and discard all messages in $M_q \backslash MT_a$.

Consider a concrete BPEL process[6] depicted in Fig. 19. This process, namely *FlightBookingFlow*, provides a flight booking service. Upon receiving a flight booking request from a client, it sets the flight price (e.g. $500), sends the offer to the client, and waits for the client's response. The client may approve the offer, in which case, the booking will be confirmed by the *FlightBookingFlow* process. Alternatively, the client may reject the offer, or no response may be received from the client within 30 min after the offer has been sent. In both cases, the client's booking request will be canceled. Finally, the *FlightBookingFlow* process replies to the client with his/her flight booking details including the flight price, the booking status (i.e. confirmed or canceled), and optionally the confirmation identifier (if the booking is confirmed).

In Fig. 19, the graphical representation depicts the above BPEL process, where each basic activity is annotated by a set of message types. Before the pick activity *handleOfferResponse* is executed, each activity is annotated with a set of two message types corresponding to the messages that may be received later. These two message types are: *FlightBookingApproveMessage* (identified by the tuple (client, tns:FlightBooking, approve)), and *FlightBookingCancelMessage* (identified by the tuple (client, tns:FlightBooking, cancel)). After one of the branches is taken in activity *handleOfferResponse*, no more messages are expected to be consumed until the end of the process. Thus, the remaining activities are annotated with an empty set of message types.

4.4. Analysis of interacting processes

For verification of the two interacting BPEL processes shown in Fig. 15, it is expected that each of the processes will deadlock under certain situations. In the requesting process P1, the invoke activity A will not be able to finish if

[6] This is a revised version of the Hotwireflow process demo available within the Oracle BPEL Process Manager download package at http://www.oracle.com/technology/software/products/ias/bpel/index.html.

```
<process name="FlightBookingFlow"
  targetNamespace="http://samples.otn.com"
  suppressJoinFailure="yes"
  xmlns:tns="http://samples.otn.com"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns:bpws="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

 <partnerLinks>
  <partnerLink name="client"
               partnerLinkType="tns:FlightBooking"
               myRole="FlightBookingProvider"
               partnerRole="FlightBookingRequester"/>
 </partnerLinks>

 <variables>
  <variable name="input" messageType="tns:FlightBookingRequestMessage"/>
  <variable name="offer" messageType="tns:FlightBookingOfferMessage"/>
  <variable name="approved" messageType="tns:FlightBookingApproveMessage"/>
  <variable name="canceled" messageType="tns:FlightBookingCancelMessage"/>
  <variable name="output" messageType="tns:FlightBookingResultMessage"/>
 </variables>

 <sequence>
  <receive name="receiveInput" partnerLink="client"
           portType="tns:FlightBooking" operation="initiate"
           variable="input" createInstance="yes"/>

  <assign name="setOffer">
   <copy>
    <from expression="number(500)"/>
    <to variable="offer" query="/tns:offer"/>
   </copy>
  </assign>

  <invoke name="sendOffer" partnerLink="client"
          portType="tns:FlightBookingCallback" operation="onOffer"
          inputVariable="offer"/>

  <assign name="recordOffer">
   <copy>
    <from variable="offer" query="/offer"/>
    <to variable="input" query="/tns:flightRequest/tns:price"/>
   </copy>
  </assign>

  <pick name="handleOfferResponse">
   <onMessage partnerLink="client" portType="tns:FlightBooking"
              operation="approve" variable="approved">
    <assign name="clientApproved">
     <copy>
      <from expression="string('Approved')"/>
      <to variable="input" query="/tns:flightRequest/tns:status"/>
     </copy>
     <copy>
      <from expression="string('12345')"/>
      <to variable="input" query="/tns:flightRequest/tns:confirmationId"/>
     </copy>
    </assign>
   </onMessage>
   <onMessage partnerLink="client" portType="tns:FlightBooking"
              operation="cancel" variable="canceled">
    <assign name="clientCanceled">
     <copy>
      <from expression="string('Canceled')"/>
      <to variable="input" query="/tns:flightRequest/tns:status"/>
     </copy>
    </assign>
   </onMessage>
   <onAlarm for="'PT30M'"> <!-- wait for 30 minutes -->
    <assign name="autoCanceled">
     <copy>
      <from expression="string('Canceled')"/>
      <to variable="input" query="/tns:flightRequest/tns:status"/>
     </copy>
    </assign>
   </onAlarm>
  </pick>

  <assign name="generateOutput">
   <copy> <from variable="input"/> <to variable="output"/> </copy>
  </assign>

  <invoke name="replyOutput" partnerLink="client"
          portType="tns:FlightBookingCallback" operation="onResult"
          inputVariable="output"/>

 </sequence>
</process>
```

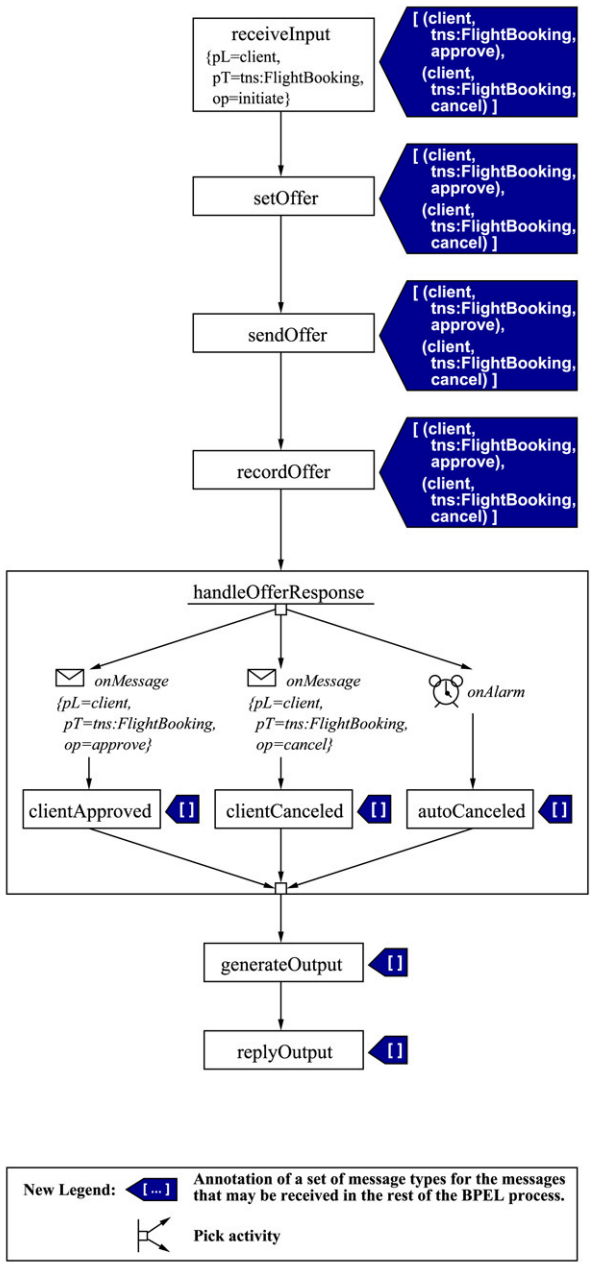


Fig. 19. Example of a BPEL process and diagrammatic representation of the corresponding annotations for garbage collection.

any of the following happens: (a) the receive activity B1 is skipped, and as a result the corresponding reply activity B2 is never performed either; (b) B1 is executed, but B2 is skipped due to some reason (e.g., the current process instance of P2 is forced to terminate after B1 has completed but before B2 starts); and (c) the request/response message is lost during transmission over a lossy medium. In all these three scenarios, a request has been sent by P1, but the corresponding response will never be returned to P1, and thus the execution of P1 will stop in the intermediate state

modeled by place w_A. On the other hand, in the responding process P2, activity B1 cannot be executed if either activity A is skipped or the request message sent from P1 is lost during transmission. Intuitively, this means that the expected message from P1 never reaches P2, thus causing P2 to be deadlock.

Again, the above verification of interacting BPEL processes can be conducted automatically using BPEL2PNML and WofBPEL. WofBPEL, by applying the available analysis techniques implemented in Woflan, can be used to check the soundness of a BPEL process, including such as the absence of deadlocks.

5. Empirical evaluation

This section presents experimental results aimed at validating the feasibility of applying the presented analysis techniques in terms of execution time. As a preamble, we present net reduction techniques which we have been found to be necessary in order to maintain the produced WF-nets within manageable sizes, and to avoid the state-explosion problem when analyzing realistic BPEL process definitions.

5.1. Net reduction

WofBPEL can be used to perform reductions on the Petri nets produced by BPEL2PNML. These are: (i) removal of the "idle" skip fragments introduced in the proposed mapping; and (ii) removal of unnecessary silent transitions (i.e. τ-transitions) from the nets.

At first, in order to facilitate the mapping of control links and fault handlers, and to be consistent in the way structured activities are mapped in BPEL, we have assumed in our mapping that any activity may be skipped. As a result, a skip path is generated for every activity in BPEL2PNML. However, not every activity can actually be skipped. A straightforward counterexample is the root's activity (i.e., the top-level process scope). As another, more detailed, example, Fig. 6 and Fig. 7(b) show the skip fragments using dashed lines. The underlying assumption for these mappings is that any activity Y that requires another activity X to be skipped, puts a token into place $\mathsf{to\_skip}_X$, waits for a token to arrive in place $\mathsf{skipped}_X$, removes the token from that place, and then continues. However, if no other activity can put a token into $\mathsf{to\_skip}_X$, then the entire fragment forming the skip path will never be executed, and therefore can be removed. By removing these idle skip fragments, the Petri net obtained from BPEL2PNML is also converted to a so-called Workflow net (WF-net) [1], on which the soundness property has been defined [1] that can be checked by the Woflan tool.[7]

Second, WofBPEL applies behavior preserving reduction rules based the ones as given by Murata [26]. This way, the size of the net can be significantly reduced by removing unnecessary silent transitions and redundant places. There is one important difference between the rules given by Murata and the rules used in WofBPEL: in our case the non-silent transitions (represented by labeled transitions) should never be removed, since we aim at preserving not only liveness and soundness, but also the observable behavior.

Fig. 20 visualizes the reduction rules used in WofBPEL, where only silent transitions (represented by unlabeled transitions) can be removed. The first rule shows that a (silent) transition connecting two places may be removed by merging the two places, provided that tokens in the first place can only move to the second place. The second rule shows that multiple alterative silent transitions can be reduced to a single one. After applying the second rule, one may be able to apply the first rule provided that the first place has only one output arc (see Fig. 20). The third rule shows that self-loops can be removed if the transition involved is silent. When applying the rules, one should clearly differentiate between silent and non-silent transitions. For example, in the fourth rule at least one of the transitions should be silent, otherwise the rule should not be applied (as indicated). In this rule the execution of y is inevitable once the silent transition has been executed. Therefore, it is only possible to postpone its occurrence. The two last rules do not remove any transitions, but remove places. Note that x and y may be or may not be silent. The reduction rules shown in Fig. 20 do not preserve the moment of choice and therefore assume trace semantics rather than branching/weak bisimulation [18].

[7] A WF-net is a Petri net that models a workflow process definition. It has one input place (called *source place*) and one output place (*sink place*). A token in the source place corresponds to a case (i.e. process instance) which needs to be handled, and a token in the sink place corresponds to a case which has been handled. Also, in a WF-net there are no dangling tasks and/or conditions. Tasks are modeled by transitions and conditions by places. Hence, every transition/place should be located on a path from the source place to the sink place.

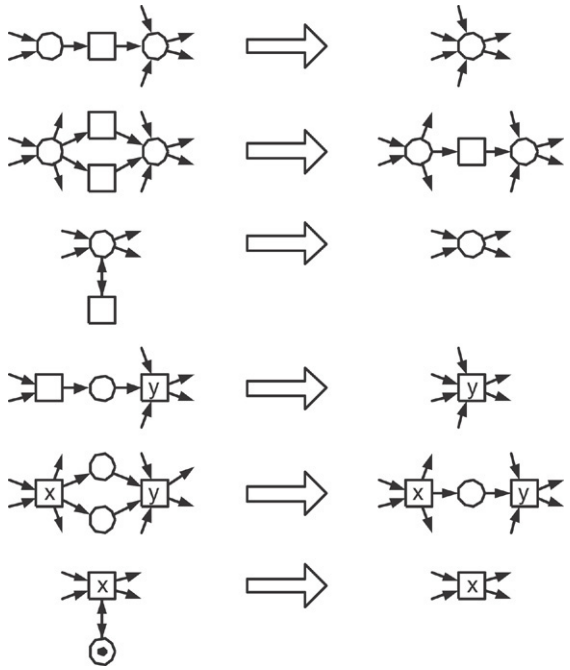


Fig. 20. Behavior preserving reduction rules used in WofBPEL.

Table 1
Statistics of 9 BPEL processes, their original Petri-net mappings produced by BPEL2PNML, and the corresponding reduced WF-nets produced by WofBPEL

BPEL process				Original Petri net		Reduced WF-net	
Name	Activities	Nest. levels	E/F/C-H	Places	Trans.	Places	Trans.
UnreachableActivity	5	3	0	59	43	34	32
ConflictingReceive	6	2	0	44	38	18	15
FlightBookingFlow	10	2	0	86	87	27	31
SalesForceFlow	14	6	0	112	112	38	43
HelpDeskServiceRequest	18	7	1	264	259	97	98
ResilientFlow	23	9	3	184	169	49	46
VacationRequest	26	7	1	191	188	58	64
OrderApproval	37	13	2	277	272	85	90
LoanPlusFlow	49	9	2	379	366	107	117

5.2. Measuring execution times

We apply BPEL2PNML and WofBPEL to the translation and analysis of 9 BPEL processes. Three of these processes, namely *UnreachableActivity*, *ConflictingReceive*, and *FlightBookingFlow*, have been presented in Sections 4.1 and 4.3. The other six processes are the largest samples that are distributed with the Oracle BPEL Process Manager (version 10.1.2). Table 1 provides the size of these 9 BPEL processes, of the Petri nets produced by BPEL2PNML, and of the corresponding reduced WF-nets produced by WofBPEL after applying the reduction rules. The size of a BPEL process is measured in terms of the number of activities (including structured activities such as *sequence* and *flow*), the maximal levels of nesting of these activities, and the number of handlers (event handlers, fault handlers and compenstation handlers) defined in the process. The size of a Petri net or a WF-net is measured in the number of places and transitions. For example, the *LoanPlusFlow* process has 49 activities, maximally 9 levels of nesting, and 2 fault handlers. The Petri net generated by BPEL2PNML contains 379 places and 366 transitions, while the reduced WF-net generated by WofBPEL contains 107 places and 117 transitions. The results highlight the importance of applying reduction rules to the nets produced by BPEL2PNML.

Table 2
Execution time (in seconds) for performing reduction on the original Petri nets and then conducting each of the four types of analysis on the reduced WF-nets for the 9 BPEL processes listed in Table 1

BPEL process	Behavioral	Structural	Competing-messages	Look-ahead
UnreachableActivity	0.051	0.052	0.051	0.047
ConflictingReceive	0.047	0.041	0.042	0.040
FlightBookingFlow	0.081	0.078	0.080	0.078
SalesForceFlow	0.120	0.117	0.120	0.120
HelpDeskServiceRequest	0.404	0.210	0.365	0.417
ResilientFlow	0.224	0.214	0.209	0.209
VacationRequest	0.273	0.227	0.260	0.238
OrderApproval	0.521	0.328	0.509	0.500
LoanFlowPlus	14.06	0.907	14.01	14.35

Table 2 shows the execution time for performing reductions on the original Petri nets and then conducting each of the four types of analysis on the reduced WF-nets for the above 9 BPEL processes.[8] There are four types of analysis. *Behavioral* analysis gives all transitions that are not relaxed sound, i.e., all transitions that are not covered by any path from start to end. *Structural* analysis approximates the set of non-relaxed-sound transitions by using transition invariants, and can be useful if the state space of the WF-net (which is needed by the behavioral analysis) cannot be constructed within reasonable time. Both types of analysis can be used to detect unreachable activities. Next, the conflicting message-consuming activities can be detected via *competing-messages* analysis, and the computation of a set of message types for each activity in a process can be performed via *look-ahead* analysis.

From Table 2, it can be seen that with the reduced WF-nets for the 9 BPEL processes shown in Table 1, all the analyses take less than half a minute (most of them takeless than half second). Also, we can observe that the structural analysis performs an order of magnitude better than the behavioral analysis. In particular, for the *LoanPlusFlow* example, the execution of the structural analysis is 14 times faster than that of the behavioral analysis. This provides evidence to support that transition invariants outperform state space methods, especially when dealing with complex processes. Also, as a comparison, we have conducted the same types of analyses on the original Petri nets obtained from BPEL2PNML for these 9 processes, and the results have shown that the execution time increases significantly. For example, the behavioral analysis for the *LoanPlusFlow* process takes 167 s, the competing-messages analysis takes 167 s, and look-ahead analysis takes more than 5 min.

Finally, during the above experiments, no actual errors were found in the process definitions. For the *ResilientFlow*, *OrderApproval* and *LoanPlusFlow* examples, the behavioral and structural analyses reported some unreachable transitions. However, these turned out to come from fault handlers in these process definitions which were meant to catch internal fault types (i.e. faults that are not explicitly thrown by the process, but may be thrown by the BPEL engine). Given a catalog of recognized internal fault types, it would be possible to eliminate such false warnings.

6. Choreography conformance checking

In this section, we show that the application of our mapping from BPEL to Petri nets is not restricted to design-time analysis, but can also be deployed to check properties at run-time. The mapping defined in Section 3 has thereby been used for choreography conformance checking between a running process and the corresponding desired choreography specification that is written in BPEL. In [2] the authors of this paper presented an approach for choreography conformance checking based on BPEL and Petri nets. Below, we briefly describe what "choreography conformance checking" is and how it applies the mapping from BPEL to Petri nets. Interested readers are referred to [2] for details.

To coordinate a collection of interacting Web services, the concept of *choreography* defines collaborations between interacting parties, i.e., the coordination process of interconnected Web services that all partners need to agree on. A choreography specification is used to describe the desired behavior of interacting parties. A language like BPEL can be used to define a desired choreography specification (for this purpose, abstract BPEL processes are used).

[8] Experiments were run on a laptop with a Pentium M processor 750, 1.86 GHz, 1 GB RAM, running Windows XP SP2 and Cygwin (using the "time" command). Each type of analysis was executed four times in a row. The time measurement of the first execution was discarded, and the average of the remaining three measurements was computed.

Assuming that there is a running process and a choreography specification, it is interesting to check whether each partner (exposed as Web service) is well behaved. Note that partners have no control over each others' services. Moreover, partners will not expose the internal structure and state of their services. This triggers the question of conformance: "Do all parties involved operate as described?" The term "choreography conformance checking" is then used to refer to this question. To address the question, one can assume the existence of both a process model which describes the desired choreography and a message log which records the actual observed behavior, i.e. an actual choreography.

Based on a process model described in terms of BPEL abstract processes, a Petri net description of the intended choreography can be created by using the translation defined in Section 3. Conformance checking is then performed by comparing a Petri net and an message log. The message log is obtained by mapping SOAP messages exchanged between different services onto MXML, a unified format for process mining [2]. Both BPEL2PNML and WofBPEL provide tool support for conformance checking of BPEL processes. In particular, WofBPEL is used to obtain a reduced WF-net from the original Petri net produced by BPEL2PNML. This WF-net, together with the above MXML log, can be taken as input to the *ProM Conformance Checker*,[9], which provides a tool used to actually measure conformance.

7. Related work

Recently, several groups have been working on providing formal semantics for BPEL with the goal of providing some form of analysis. The software described in this paper (WofBPEL) was released early 2005 and presented at some conferences [29,27] in 2005. It uses a Petri-net-based analysis tool dedicated to the analysis of workflow process. This tool (Woflan) was developed over the last decade [34]. In parallel with the development of WofBPEL, several other groups worked on providing formal semantics and analysis tools for BPEL processes, cf. [11,11–17,19,21–24,31]. In this section we position our work with respect to these papers. It is impossible to discuss all of them in detail. Therefore, we focus on three of the most relevant tools in this area: *WSAT*, *LTSA-WS/BPEL4WS*, and *Tools4BPEL*.

WSAT (Web Service Analysis Tool) [16,17] is a formal specification, verification, and analysis tool for web service compositions based on so-called Guarded Automata (GA). The tool has been developed at the University of California at Santa Barbara. BPEL specifications are translated to guarded automata. These are then mapped onto Promela, the input language of the well-know model-checker SPIN. Using SPIN, a variety of properties can be checked as long as the mapping yields a finite system. The authors have focused on interacting BPEL web services [16,17] using concepts such as synchronizability (i.e., when can asynchronous communication be replaced by synchronous communication).

LTSA-WS/BPEL4WS [14,15] (also known as the LTSA WS-Engineer plug-in for Eclipse) is an extension to the Labelled Transition System Analyser (LTSA) which allows models to be described by translation of the BPEL4WS implementations and WS-CDL descriptions. It has been developed at the Imperial College London. LTSA-WS/BPEL4WS is able to map BPEL specifications onto labeled transition systems and perform various checks including deadlock freedom, safety, and progress properties (all provided by LTSA). Since model checking is used, also other properties can be investigated. Interestingly, the tool also allows for the synthesis of Message Sequence Charts (MSCs) and compare the synthesized result with the BPEL specification [15].

Tools4BPEL is a tool set which consists of *BPEL2oWFM* and *Fiona*. It has been developed at the Humboldt-Universität zu Berlin. BPEL2oWFM maps BPEL specifications onto Petri nets, and is a successor of the BPEL2PN tool based on the translation described in [19,31]. The focus of the work is twofold. On the one hand, different properties are being verified using model checking techniques (through LOLA) [19]. On the other hand, the tool Fiona can check for controllability (i.e., is there an environment that can interact properly) and, if so, generate the operating guidelines (i.e., instructions on how to use the service) [22].

Table 3 provides a more structured overview of related work. It should be noted that some of these efforts focus on only subsets of BPEL. The columns of the Table 3 correspond to the following criteria:

– *Tech* indicates the formalization technique used: FSM for Finite State Machines, PA for Process Algebra, ASM for Abstract State Machines, HPN for High-level Petri Nets and PN for plain (i.e. low-level) Petri Nets.

[9] This tool has been developed in the context of the *ProM framework* which offers a wide range of tools related to process mining. Both documentation and software can be downloaded from http://www.processmining.org.

Table 3
A comparative summary of related work on formalization and analysis of BPEL

Related work	Tech	SA	CL	EFC	TAV	FDM	Comm
Farahbod et al. [11]	ASM	+	+/−	+	−	+	−
Ferrara [12]	PA	+	−	+	−	−	−
Fisteus et al. [13]	FSM	+	−	+/−	+/−	−	−
Foster et al. [14,15]	FSM	+	+	+/−	+	−	+
Fu et al. [16,17]	FSM	+	+	+/−	+	−	+
Hinz et al. [19], Lohmann et al. [22], Martens [23,24], Stahl [31]	HPN	+	+	+	+	+	+
Koshkina and van Breugel [21]	PA	+	+	−	+/−	+	−
Our work	PN	+	+	+	+	+	+

- *SA* indicates whether the formalization covers Structured Activities fully (+), partially (+/−) or not at all (−). It can be seen that all the approaches referred to cover this subset of BPEL.
- *CL* indicates whether the formalization covers Control Links. Here a +/− rating is given for formalizations that cover control link constructs, but do not fully cover join conditions.
- *EFC* indicates whether the formalization covers Event handling, Fault handling and Compensation. Some references cover fault handling, but do not cover compensation and/or event handling, in which case, a +/− rating is assigned.
- *TAV* (Tool for Automatic Verification) indicates whether a verification tool is provided. Here a +/− rating is given in the case of some efforts [13,21] where the authors claim to have developed and/or used a tool to verify deadlock-freeness of BPEL processes. However, no example has been given of a deadlocking BPEL process (abstracting from process interactions which can create deadlocks across processes), and hence such analysis is unnecessary. Apart from this, some of the cited references refer to the possibility of performing formal verification [11,12], but do not develop automated means of doing so (− rating). It should also be noted that most of the tools that provide support simply export the state space to some general model checker (e.g., [15] exports to LTSA, [22] exports to LOLA, and [16,17] exports to SPIN). Our approach differs in the sense that it uses a dedicated checker (Woflan) with analysis techniques tailored towards (relaxed) soundness and other properties specific for workflow-like processes.
- *FDM* (Formally Defined Mapping) indicates whether an abstract syntax for BPEL is provided and the mapping from BPEL to the target language is formally defined. Here, a + rating is given if the formally defined mapping is complete with respect to the scope of the work claimed by the author(s). A − rating does not imply that it does not exist, however, it has not been made public (as far as we can tell).
- *Comm* indicates whether or not the translation covers the interactions (i.e. communications) between BPEL processes. Tools such as WSAT, LTSA-WS/BPEL4WS, Tools4BPEL, and WofBPEL provide support. However, each of the tools aims at different questions with respect to interaction, e.g., BPEL2oWFM and Fiona in Tools4BPEL focus on checking "controllability" and generating "operating guidelines", WSAT focuses on "synchronizability", and LTSA-WS/BPEL4WS also addresses the synthesis of requirements from example MSCs. In this paper we showed that our approach can be used to answer questions related to (1) detection of unreachable activities; (2) detection of conflicting message-consuming activities; (3) garbage collection on queued messages; (4) analysis of interacting processes (i.e., is the composed process sound); and (5) choreography conformance checking.

Our approach is most related to the work that has led to the BPEL2PN and Tools4BPEL toolsets [19,22,31]. Our mapping is different, but the final goal is similar: generating a Petri net based on a BPEL process definition and using this net for analysis purposes. There are two technical differences in the approaches. First, our mapping has been designed to yield directly low-level nets, whereas the mapping defined in [19,31], proceeds in two steps: first a high-level net is created, and then this high-level net is expanded into a low-level net by abstracting from the data in the tokens and expanding the net accordingly. This abstraction is not described in detail, making it difficult to determine its implications on the structure of the resulting net and on the complexity of subsequent analysis techniques. In particular, our approach maps join conditions directly into a structure composed of plain places and transitions, while in [19,31], join conditions are treated as expressions attached to transitions (a high-level Petri net feature). Second, the set of net reduction rules employed are different. Our approach relies on the liveness and soundness-preserving rules presented in [26], while the proposal in [22] uses four reduction rules specifically defined for this purpose: two

of these rules are meant to detect structurally dead places (which are also removed by WofBPEL), while the other two are special cases of Murata's reduction rules.

This paper builds upon on our previous work on formalizing BPEL. A less complete and earlier version of the formalization (without the tool support) can be found in [32], while a mapping of a small subset of BPEL consisting of control links and join conditions can be found in [20].

8. Conclusions

BPEL is gaining increasing adoption as a process-oriented service composition language, as reflected by the large number of implementations (see http://en.wikipedia.org/wiki/BPEL for a compendium). However, current tools lack the ability to statically detect undesirable situations such as unreachable activities or pairs of activities that may compete for the same message. Also, current BPEL implementations are not optimized with respect to the 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 its coverage of control-flow constructs and communication actions. In particular, it is the first attempt at providing formal semantics for "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, which translates BPEL code into PNML code, and WofBPEL, which 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. In this paper, we also discussed the application of this mapping to choreography conformance checking. Specifically, we showed that by mapping abstract BPEL processes onto Petri nets and SOAP messages onto message logs, it is possible to detect deviations between the specification in BPEL and the actual behavior observed.

Our future work aims at extending this mapping to cover data manipulation aspects. This will allow us to apply simulation techniques to check properties for which a 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 [4].

Appendix A

This appendix provides formal definitions of the syntax and the semantics of WS-BPEL. We first introduce a Boolean function and an 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 {`true`, `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*: $F \times Assign(F) \to \mathbb{B}$.

A.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 $\mathcal{W} = (\mathcal{A}, \mathcal{E}, \mathcal{C}, \mathcal{L}, \mathsf{HR}, \mathsf{type}_{\mathcal{A}}, \mathsf{type}_{\mathcal{E}}, \mathsf{instance}, \mathsf{name}, <_{seq}, <_{swt}, \mathsf{serialscp}, \mathtt{process}, \mathsf{trigger}_f, \mathsf{scp}_c, \mathsf{trigger}_c, \mathsf{scp}_t, \mathsf{trigger}_{tf}, \mathsf{LR}, \mathsf{joincon}, \mathsf{supjoinf}, \mathsf{trigger}_{jf})$ 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 $\mathsf{HR}_\mathsf{p} = \pi_{1,3}\mathsf{HR}$ (projection of HR on two activity sets), $\mathsf{children}(a) = \{a' \in \mathcal{A} \mid \mathsf{HR}_\mathsf{p}(a, a')\}$ 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,
- $\mathsf{type}_\mathcal{A}$: $\mathcal{A} \to \mathcal{T}_\mathcal{A}$ is a function that assigns types to activities taken from the set of activity types $\mathcal{T}_\mathcal{A} = \{$`sequence`, `flow`, `pick`, `switch`, `while`, `scope`, `invoke`, `receive`, `reply`, `wait`, `assign`, `empty`, `throw`, `compensate`, `exit`$\}$,
- $\forall t \in \mathcal{T}_\mathcal{A}$, $\mathcal{A}_t = \{a \in \mathcal{A} \mid \mathsf{type}_\mathcal{A}(a) = t\}$ is a set of all activities of type t,
- $\mathsf{type}_\mathcal{E}$: $\mathcal{E} \to \mathcal{T}_\mathcal{E}$ is a function that assigns types to events taken from the set of event types $\mathcal{T}_\mathcal{E} = \{$`message`, `alarm`, `fault`, `compensation`, `termination`$\}$.
- $\forall t \in \mathcal{T}_\mathcal{E}$, $\mathcal{E}_t = \{e \in \mathcal{E} \mid \mathsf{type}_\mathcal{E}(e) = t\}$ is a set of all events of type t,
- instance: $\mathcal{A}_{receive} \cup \mathcal{A}_{pick} \to \mathbb{B}$ is a function which assigns a Boolean value to the attribute `createInstance` 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. the internal nodes of the HR tree,
- let $\mathcal{A}^{basic} = \mathcal{A}_{invoke} \cup \mathcal{A}_{receive} \cup \mathcal{A}_{reply} \cup \mathcal{A}_{wait} \cup \mathcal{A}_{assign} \cup \mathcal{A}_{empty} \cup \mathcal{A}_{throw} \cup \mathcal{A}_{compensation} \cup \mathcal{A}_{exit}$ be a set of basic activities, $\forall s \in \mathcal{A}^{basic}$, $\mathsf{children}(s) = \varnothing$, i.e. the leaves of the HR tree,
- given $\mathcal{A}' = \mathcal{A}_{sequence} \cup \mathcal{A}_{flow}$, $\mathsf{HR} \cap (\mathcal{A}' \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}' \times \{\bot\} \times \mathcal{A})$,
- $\forall s \in \mathcal{A}_{sequence}$, $\exists$ an order $<^s_{seq}$ which is a strict total order over 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 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 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 $\neg\exists a \in \mathsf{children}(s)$, $\mathsf{last}(s) <^s_{swt} a$, and let $c \in \mathcal{C}$, $\mathsf{HR}(s, c, \mathsf{last}(s)) \Rightarrow \forall assign(c) \in Assign(C)$, $eval(c, assign(c)) =$ `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,
- $\mathsf{HR} \cap (\mathcal{A}_{scope} \times \mathcal{B} \times \mathcal{A}) = \mathsf{HR} \cap (\mathcal{A}_{scope} \times (\mathcal{E} \cup \{\bot\}) \times \mathcal{A})$, 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., a scope has at least one fault handler,
  - $|\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{compensation} \times \mathcal{A})| \leqslant 1$, i.e., a scope has at most one compensation handler,
  - $|\mathsf{HR} \cap (\{s\} \times \mathcal{E}_{termination} \times \mathcal{A})| \leqslant 1$, i.e., a scope has 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 behavior 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,
- serialscp: $\mathcal{A}_{scope} \to \mathbb{B}$ is a function assigning a Boolean value to the `variableAccessSeriali-zable` attribute of a scope. $\forall s \in \mathcal{A}_{scope}$, $\forall a \in \mathsf{descendants}(s)$, $(\mathsf{serialscp}(s) =$ `true` $\wedge\ a \in \mathcal{A}_{scope}) \Rightarrow \mathsf{serialscp}(a) =$ `false`, i.e., serializable scopes cannot be nested,
- `process` $\in \mathcal{A}_{scope}$ is the root of the HR tree, and $\mathsf{serialscp}($`process`$) =$ `false`,

($*$ fault handling $*$)

- $\mathsf{trigger}_{tf}$: $\mathcal{A}_{throw} \to \mathcal{E}_{fault}$ is a function which maps each throw activity to a (process-defined) fault event triggered by that activity,

– $\mathcal{A}_{throw}^{<re>} = \mathcal{A}_{throw} \cap (\bigcup_{s \in \mathcal{A}_{scope}} (\mathcal{A}_{\mathcal{H}}^{fault}(s) \cap \mathcal{A}^{directenc}(s)))$ is the set of throw activities used to throw faults that cannot be solved during the fault handling. 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,
– $\mathcal{A}_{throw} \cap (\mathcal{A}_{\mathcal{H}}^{fault}(\texttt{process}) \cap \mathcal{A}^{directenc}(\texttt{process})) = \varnothing$, i.e., a fault handler of the process scope cannot throw any fault further,

(∗ compensation ∗)

– scp_c: $\mathcal{E}_{compensation} \to \mathcal{A}_{scope} \backslash \{\texttt{process}\}$ 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,
– $\mathsf{trigger}_c$: $\mathcal{A}_{compensate} \to \mathcal{E}_{compensation}$ 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 behavior of a scope, that is, it may be used only for exception handling and termination,
– $\forall s \in \mathcal{A}_{scope}$, let $\mathcal{A}_{\mathcal{H}}^{fct}(s) = \mathcal{A}_{\mathcal{H}}^{fault}(s) \cup \mathcal{A}_{\mathcal{H}}^{compensation}(s) \cup \mathcal{A}_{\mathcal{H}}^{termination}(s)$, then $(a \in \mathcal{A}_{compensate} \cap (\mathcal{A}_{\mathcal{H}}^{fct}(s) \cap \mathcal{A}^{directenc}(s))) \Rightarrow \mathsf{scp}_c(\mathsf{trigger}_c(a)) \in \mathcal{A}^{normal}(s)$, 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: $\mathcal{E}_{termination} \to \mathcal{A}_{scope} \backslash \{\texttt{process}\}$ is an injective function mapping a termination event to a scope such that the occurrence of that event invokes 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 $dom(\mathsf{trigger}_t) = \{(e, a) \in \mathcal{E}^{ft} \times \mathcal{A}_{scope} \mid \exists (s, e, x) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft} \times \mathcal{A}$, such that $a \in \mathsf{clan}(x)\}$,
– $\forall s \in \mathcal{A}_{scope}$, $\mathcal{A}_{throw} \cap (\mathcal{A}_{\mathcal{H}}^{termination}(s) \cap \mathcal{A}^{directenc}(s)) = \varnothing$, i.e., a throw activity cannot be used when handling the termination for a scope. A termination handler is a special type of fault handler that cannot throw any fault unresolved,

(∗ control links ∗)

– $\mathsf{LR} \subseteq \mathcal{A} \times \mathcal{L} \times \mathcal{A}$ 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:
  • $\forall (a, l, a') \in \mathsf{LR}$, $a \notin \mathsf{clan}(a')$, i.e., a control link cannot connect an activity to any of its ancestors,
  • $\forall s \in \mathcal{A}_{sequence}$, $\forall \{x, x'\} \subseteq \mathsf{children}(s)$, $\forall a \in \mathsf{clan}(x)$, $\forall a' \in \mathsf{clan}(x')$, $x <_{seq}^{s} x' \Rightarrow \neg \exists l \in \mathcal{L}$ such that $\mathsf{LR}(a', l, a)$, 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 $\mathcal{A}_{scope}^{serial} = \{s \in \mathcal{A}_{scope} \mid \mathsf{serialscp}(s) = \texttt{true}\}$ be a set of serializable scopes, and $\mathcal{A}' = \mathcal{A}_{while} \cup \mathcal{A}_{scope}^{serial}$, $\forall s \in \mathcal{A}', \forall a \in \mathsf{descendants}(s), \forall a' \in \mathcal{A} \backslash \mathsf{clan}(s), \neg \exists l \in \mathcal{L}$ such that $\mathsf{LR}(a, l, a') \vee \mathsf{LR}(a', l, a)$, 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 $\mathcal{A}_h^{event}(s, e) = \{a \in \mathcal{A}_{\mathcal{H}}^{event}(s) \mid \exists x \in \mathcal{A}_{\mathcal{H}}^{event}(s), \mathsf{HR}(s, e, x) \wedge a \in \mathsf{clan}(x)\}$ be the set of activities in an event handler triggered upon the occurrence of event e in scope s, then $\forall a \in \mathcal{A}_h^{event}(s, e)$, $\forall a' \in \mathcal{A} \backslash \mathcal{A}_h^{event}(s, e), \neg \exists l \in \mathcal{L}$ such that $\mathsf{LR}(a, l, a') \vee \mathsf{LR}(a', l, a)$, i.e. a control link cannot cross the boundary of an event handler,
  • $\forall s \in \mathcal{A}_{scope}, \forall a, a' \in \mathcal{A}_{\mathcal{H}}^{compensation}(s)$ and $a \neq a'$, $\neg \exists l \in \mathcal{L}$ such that $\mathsf{LR}(a, l, a') \vee \mathsf{LR}(a', l, a)$, 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 $\mathcal{A}_{\mathcal{H}}^{ft} = \mathcal{A}_{\mathcal{H}}^{fault} \cup \mathcal{A}_{\mathcal{H}}^{termination}$, $\mathcal{A}_h^{ft}(s, e) = \{a \in \mathcal{A}_{\mathcal{H}}^{ft}(s) \mid \exists x \in \mathcal{A}_{\mathcal{H}}^{ft}(s), \mathsf{HR}(s, e, x) \wedge a \in \mathsf{clan}(x)\}$ be the set of activities for handling a fault (or termination) event e in scope s, then $\forall a \in \mathcal{A}_h^{ft}(s, e), \forall a' \in \mathcal{A} \setminus \mathcal{A}_h^{ft}(s, e), \neg \exists l \in \mathcal{L}$ such that $\mathsf{LR}(a', l, a)$, i.e., a control link that crosses the boundary of a fault handler (or termination handler), must be outbound,
– let $\mathcal{A}^{source} = \{a \in \mathcal{A} \mid \exists l \in \mathcal{L}, (a, l) \in \pi_{1,2}\mathsf{LR}\}$ be a set of source activities of all control links, and $\mathcal{A}^{target} = \{a \in \mathcal{A} \mid \exists l \in \mathcal{L}, (l, a) \in \pi_{2,3}\mathsf{LR}\}$ be a set of target activities of all control links, then $\forall a \in \mathcal{A}^{source}$, $\mathcal{L}_{out}(a) = \{l \in \mathcal{L} \mid \exists a' \in \mathcal{A}, \mathsf{LR}(a, l, a')\}$ is a set of all outgoing control links from a, and $\forall a \in \mathcal{A}^{target}$, $\mathcal{L}_{in}(a) = \{l \in \mathcal{L} \mid \exists a' \in \mathcal{A}, \mathsf{LR}(a', l, a)\}$ is a set of all incoming control links to a,

– let $a \in \mathcal{A}^{target}$, $\mathsf{joincon}(a)$, which expresses the join condition of incoming control links at a, is a Boolean function over $\mathcal{L}_{in}(a)$ (i.e. $Var(\mathsf{joincon}(a)) = \mathcal{L}_{in}(a)$),
– $\mathsf{supjoinf}$: $\mathcal{A} \to \mathbb{B}$ is a function assigning a Boolean value to the `suppressJoinFailure` attribute of each activity,
– let $\mathcal{A}_{SJF} = \{a \in \mathcal{A} \mid \mathsf{supjoinf}(a) = \texttt{true}\}$ and $\mathcal{A}_{TJF} = \{a \in \mathcal{A} \mid \mathsf{supjoinf}(a) = \texttt{false}\}$, the function $\mathsf{trigger}_{jf}$: $\mathcal{A}^{target} \cap \mathcal{A}_{TJF} \to \mathcal{E}_{fault}$ maps a target activity of which the attribute `suppressJoinFailure` is set to `false`, to the corresponding join failure fault event.

A.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: $\forall\, s \in \mathcal{A}_{sequence}$,

– $\mathsf{head}(s) \in \mathsf{children}(s)$ is the first activity to occur in s, i.e. $\neg\,\exists\, a \in \mathsf{children}(s),\ a <^{s}_{seq} \mathsf{head}(s)$,
– $\mathsf{tail}(s) \in \mathsf{children}(s)$ is the last activity to occur in s, i.e. $\neg\,\exists\, a \in \mathsf{children}(s),\ \mathsf{tail}(s) <^{s}_{seq} a$,
– the relation $\lessdot^{s}_{seq} = \{(a, a') \in <^{s}_{seq} \mid \neg\,\exists\, x \in \mathsf{children}(s),\ (a <^{s}_{seq} x) \wedge (x <^{s}_{seq} a')\}$ 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 $\mathcal{W}$, $\forall\, a \in \mathcal{A}^{target}$, $Var(\mathsf{joincon}(a)) = \mathcal{L}_{in}(a)$. Let $\mathsf{b}_a = assign(\mathsf{joincon}(a))$, 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, $\mathcal{L}_{OUT}(sA) = \{l \in \mathcal{L} \mid \exists\, a \in sA, \exists\, a' \in \mathcal{A} \backslash sA$, such that $\mathsf{LR}(a, l, a')\}$ 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: $\mathcal{A}_{scope} \to \mathcal{A}$ maps each scope activity to its main activity, such that $\forall\, s \in \mathcal{A}_{scope}$, $\mathsf{HR}(s, \bot, \mathsf{main}(s))$.

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 $e \in \mathcal{E}^{ft}$, $\mathcal{A}^{ft}_{h}(s, e)$ (see Definition 1) is a set of activities for handling a fault event or a termination event e in scope s; $\mathcal{E}^{ft}_{scp}(s) = \{e \in \mathcal{E}^{ft} \mid (s, e) \in \pi_{1,2}\mathsf{HR}\}$ is a set of fault events or termination events associated with scope s; and $\mathcal{L}^{ft}_{OUT}(s) = \bigcup_{e \in \mathcal{E}^{ft}_{scp}(s)} \mathcal{L}_{OUT}(\mathcal{A}^{ft}_{h}(s, e))$ 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 $s \in \mathcal{A}_{scope}$, $\mathcal{A}^{fct}_{\mathcal{H}}(s)$ (see Definition 1) is a set of activities for handling exceptions and termination in s, and $\mathcal{A}^{nfct}(s) = \mathcal{A}^{normal}(s) \setminus (\bigcup_{x \in \mathcal{A}^{normal}(s) \cap \mathcal{A}_{scope}} \mathcal{A}^{fct}_{\mathcal{H}}(s))$ 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:

$$
\begin{aligned}
P_{\mathcal{W}} = {} & \{r_x \mid x \in \mathcal{A}\} \cup && \text{– activity ready}\\
& \{s_x \mid x \in \mathcal{A}\} \cup && \text{– activity started}\\
& \{c_x \mid x \in \mathcal{A}\} \cup && \text{– activity completed}\\
& \{f_x \mid x \in \mathcal{A}\} \cup && \text{– activity finished}\\
& \\
& \{to\_skip_x \mid x \in \mathcal{A}\} \cup && \text{– to skip activity}\\
& \{skipped_x \mid x \in \mathcal{A}\} \cup && \text{– skipped activity}\\
& \{skipping_x \mid x \in \mathcal{A}^{structured}\} && \text{– skipping activity}
\end{aligned}
$$

$\{tc_l \mid l \in \mathcal{L}\} \cup$ – 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
$\{to\_f_x \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}\} \cup$ – skipping to "finished" state

$\{to\_continue_x \mid x \in \mathcal{A}_{scope}\} \cup$ – 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
– ($*$ scope status collected $*$)
$\{scp\_stat\_collected_x^{\text{NRM}} \mid x \in \mathcal{A}_{scope}\} \cup$ – on normal path
$\{scp\_stat\_collected_x^{\text{SKP}} \mid x \in \mathcal{A}_{scope}\} \cup$ – on skip path
$\{scp\_stat\_collected_x^{\text{SJF}} \mid x \in \mathcal{A}_{scope}\} \cup$ – on suppress join failure path

$\{to\_invoke_{x,e}^{\text{EH}} \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal}\} \cup$ – 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
$\{to\_invoke_{x,e}^{\text{FH}} \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$ – ready to invoke fault handler
$\{invoked_{x,e}^{\text{FH}} \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$ – fault handler invoked

$\{to\_invoke_x^{\text{CH}} \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$ – intend to invoke compensation
$\{endc_x \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$ – end of compensation handling

$\{to\_invoke_x^{\text{TH}} \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$ – ready to invoke termination
$\{invoked_x^{\text{TH}} \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$ – termination handler invoked

$\{to\_c_x^{\text{STP}} \mid x \in \mathcal{A}_{pick} \cap (\bigcup_{s \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(s))\} \cup$
$\{to\_c_x^{\text{EXT}} \mid x \in \mathcal{A}_{pick}\} \cup$ – skipping to "completed" state
$\{to\_exit, no\_exit\}$ – 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
$\{SC_{y,y'}^x \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot_{seq}^x y'\} \cup$ – sequence continue
$\{SE_x \mid x \in \mathcal{A}_{sequence} \cup \mathcal{A}_{scope}\} \cup$ – sequence/scope end

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

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

$\{SKP_x \mid x \in \mathcal{A}\} \cup$ – skip activity
$\{SKPF_x \mid x \in \mathcal{A}^{structured}\} \cup$ – skipping activity finish
$\{SKP\_CS_{y,y'}^x \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot_{seq}^x y'\} \cup$ – 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
$\{JCE^{\mathsf{b}_x} \mid x \in \mathcal{A}^{target} \wedge \mathsf{b}_x \in \mathbb{B}^{\mathcal{L}_{in}(x)}\} \cup$ – join condition evaluation
$\{SJF_x \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{SJF}\} \cup$ – suppress join failure (SJF)
$\{SJFF_x \mid x \in \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$ – suppressing join failure finish
$\{CLT\_JCT_x \mid x \in \mathcal{A}^{target}\} \cup$ – collect join condition true
$\{CLT\_JCF_x \mid x \in \mathcal{A}^{target}\} \cup$ – collect join condition false

$\{CLT\_SNP\_NRM_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot on normal path
$\{CLT\_NSNP\_NRM_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot on normal path
$\{CLT\_SNP\_SKP_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot on skip path
$\{CLT\_NSNP\_SKP_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot on skip path
$\{CLT\_SNP\_SJF_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot on SJF path
$\{CLT\_NSNP\_SJF_x \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot on SJF path

$\{E_{x,e} \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \times \mathcal{E}^{ntfnc}\} \cup$ – event of non-thrown fault nor – compensation

$\{HB_{x,e} \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A} \times (\mathcal{E} \setminus \mathcal{E}^{normal})\} \cup$ – handling exception 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
$\{NOP_x \mid x \in \mathcal{A}_{scope}\} \cup$ – "no-op" in compensation

$\{IGN\_STP_x \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap (\bigcup_{s \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(s))\} \cup$ – ignore join failure
$\{BYP\_STP_x \mid x \in (\mathcal{A}^{basic} \cup \mathcal{A}_{pick}) \cap (\bigcup_{s \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(s))\} \cup$
$\{BYP\_EXT_x \mid x \in \mathcal{A}^{basic} \cup \mathcal{A}_{pick}\} \cup$ – bypass basic activity or pick
$\{BYPF\_STP_x \mid x \in \mathcal{A}_{pick} \cap (\bigcup_{s \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(s))\} \cup$
$\{BYPF\_EXT_x \mid x \in \mathcal{A}_{pick}\}$ – bypassing pick finish

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

$F_{\mathcal{W}} = \{(s_x, A_x) \mid x \in \mathcal{A}^{basic}\} \cup \{(A_x, c_x) \mid x \in \mathcal{A}^{basic}\} \cup$ – 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

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

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

$\{(s_x, XS_{x,y}) \mid x \in \mathcal{A}_{switch} \wedge y \in \mathsf{children}(x)\} \cup$ – (∗ switch/pick ∗)
$\{(XS_{x,y}, r_y) \mid x \in \mathcal{A}_{switch} \wedge y \in \mathsf{children}(x)\} \cup$ – XOR-split

$\{(s_x, E_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{pick} \times \mathcal{E}^{normal}\} \cup$
$\{(E_{x,e}, r_y) \mid (x, e, y) \in \mathsf{HR} \cap \mathcal{A}_{pick} \times \mathcal{E}^{normal} \times \mathcal{A}\} \cup$ – deferred XOR-split
$\{(XS_{x,y}, to\_skip_{y'}) \mid x \in \mathcal{A}_{switch} \wedge y \in \mathsf{children}(x) \wedge y' \in \mathsf{children}(x) \backslash \{y\}\} \cup$
$\{(E_{x,e}, r_{y'}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{pick} \times \mathcal{E}^{normal} \wedge y' \in \mathsf{children}(x) \wedge (x, e, y') \notin \mathsf{HR}\} \cup$ – skip unchosen branches
$\{(skipped_{y'}, XJ_{y,x}) \mid x \in \mathcal{A}_{switch} \cup \mathcal{A}_{pick} \wedge y \in \mathsf{children}(x) \wedge y' \in \mathsf{children}(x) \backslash \{y\}\} \cup$ – skipped unchosen branches
$\{(f_y, XJ_{y,x}) \mid x \in \mathcal{A}_{switch} \cup \mathcal{A}_{pick} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(XJ_{y,x}, c_x) \mid x \in \mathcal{A}_{switch} \cup \mathcal{A}_{pick} \wedge y \in \mathsf{children}(x)\} \cup$ – 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
$\{(to\_skip_x, SKP_x) \mid x \in \mathcal{A}\} \cup$ – (∗ skip path ∗)
$\{(SKP_x, skipped_x) \mid x \in \mathcal{A}^{basic}\} \cup$ – skipped basic activity
$\{(SKP_x, skipping_x) \mid x \in \mathcal{A}^{structured}\} \cup$ – skip structured activity

$\{(SKP_x, to\_skip_y) \mid x \in \mathcal{A}^{structured}_{nonseq} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(skipped_y, SKPF_x) \mid x \in \mathcal{A}^{structured}_{nonseq} \wedge y \in \mathsf{children}(x)\} \cup$ – skipping non-seq. activity
$\{(SKP_x, to\_skip_y) \mid x \in \mathcal{A}_{sequence} \wedge y = \mathsf{head}(x)\} \cup$
$\{(skipped_y, SKP\_CS^x_{y,y'}) \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot^x_{seq} y\} \cup$
$\{(SKP\_CS^x_{y,y'}, to\_skip_{y'}) \mid x \in \mathcal{A}_{sequence} \wedge y \lessdot^x_{seq} y\} \cup$
$\{(skipped_y, SKPF_x) \mid x \in \mathcal{A}_{sequence} \wedge y = \mathsf{tail}(x)\} \cup$ – skipping sequence activity
$\{(skipping_x, SKPF_x) \mid x \in \mathcal{A}^{structured}\} \cup$
$\{(SKPF_x, skipped_x) \mid x \in \mathcal{A}^{structured}\} \cup$ – skipped structured activity

$\{(lst_l, JCE^{\mathsf{b}_x}) \mid x \in \mathcal{A}^{target} \wedge l \in \mathcal{L}_{in}(x) \wedge \mathsf{b}_x(l) = \mathtt{true}\} \cup$ – (∗ control link ∗)
$\{(lsf_l, JCE^{\mathsf{b}_x}) \mid x \in \mathcal{A}^{target} \wedge l \in \mathcal{L}_{in}(x) \wedge \mathsf{b}_x(l) = \mathtt{false}\} \cup$ – join condition evaluation
$\{(JCE^{\mathsf{b}_x}, jct_x) \mid x \in \mathcal{A}^{target} \wedge eval(\mathsf{joincon}(x), \mathsf{b}_x) = \mathtt{true}\} \cup$ – join condition to true
$\{(JCE^{\mathsf{b}_x}, jct_x) \mid x \in \mathcal{A}^{target} \wedge eval(\mathsf{joincon}(x), \mathsf{b}_x) = \mathtt{false}\} \cup$ – join condition to false

$\{(jct_x, PRE_x) \mid x \in \mathcal{A}^{target}\} \cup$ – pre-condition positive
$\{(PST_x, tc_l) \mid x \in \mathcal{A}^{source} \wedge l \in \mathcal{L}_{out}(x)\} \cup$ – post-condition evaluation
$\{(tc_l, SET\_LST_l) \mid l \in \mathcal{L}\} \cup \{(SET\_LST_l, lst_l) \mid l \in \mathcal{L}\} \cup$ – link status to true
$\{(tc_l, SET\_LSF_l) \mid l \in \mathcal{L}\} \cup \{(SET\_LSF_l, lsf_l) \mid l \in \mathcal{L}\} \cup$ – link status to false

$\{(jcf_x, SJF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{SJF}\} \cup$ – 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)
$\{(SJF_x, to\_f_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$ – SJF_s (for structured activity)

$\{(SJF_x, to\_skip_y) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}_{nonseq} \cap \mathcal{A}_{SJF} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(SJF_x, to\_skip_y) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{sequence} \cap \mathcal{A}_{SJF} \wedge y = \mathsf{head}(x)\} \cup$ – SJF_s: to skip sub-activities
$\{(to\_f_x, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF}\} \cup$
$\{(skipped_y, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}^{structured}_{nonseq} \cap \mathcal{A}_{SJF} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(skipped_y, SJFF_x) \mid x \in \mathcal{A}^{target} \cap \mathcal{A}_{sequence} \cap \mathcal{A}_{SJF} \wedge$

$$y = \mathsf{tail}(x)\} \cup$$ – 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

$$\{(SJF_x, lsf_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{basic} \cap \mathcal{A}_{SJF} \wedge l \in \mathcal{L}_{out}(x)\} \cup$$
$$\{(SJFF_x, lsf_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{structured} \cap \mathcal{A}_{SJF} \wedge l \in \mathcal{L}_{out}(x)\} \cup$$ – dead path elimination (DPE)

$$\{(to\_skip_x, CLT\_JCT_x) \mid x \in \mathcal{A}^{target}\} \cup$$ – ($*$ skip control link $*$)
$$\{(CLT\_JCT_x, to\_skip_x) \mid x \in \mathcal{A}^{target}\} \cup$$
$$\{(jct_x, CLT\_JCT_x) \mid x \in \mathcal{A}^{target}\} \cup$$
$$\{(CLT\_JCT_x, jcv_x) \mid x \in \mathcal{A}^{target}\} \cup$$ – collect join condition true
$$\{(to\_skip_x, CLT\_JCF_x) \mid x \in \mathcal{A}^{target}\} \cup$$
$$\{(CLT\_JCF_x, to\_skip_x) \mid x \in \mathcal{A}^{target}\} \cup$$
$$\{(jcf_x, CLT\_JCF_x) \mid x \in \mathcal{A}^{target}\} \cup$$
$$\{(CLT\_JCF_x, jcv_x) \mid x \in \mathcal{A}^{target}\} \cup$$ – collect join condition false
$$\{(jcv_x, SKP_x) \mid x \in \mathcal{A}^{target}\} \cup$$ – skip join condition value
$$\{(SKP_x, lsf_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{basic} \wedge l \in \mathcal{L}_{out}(x)\} \cup$$
$$\{(SJFF_x, lsf_l) \mid x \in \mathcal{A}^{source} \cap \mathcal{A}^{structured} \wedge l \in \mathcal{L}_{out}(x)\} \cup$$ – DPE in case of skipping

$$\{(s_x, SB_x) \mid x \in \mathcal{A}_{scope}\} \cup \{(SB_x, to\_continue_x) \mid x \in \mathcal{A}_{scope}\} \cup$$ – ($*$ 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
$$\{(to\_continue_x, SE_x) \mid x \in \mathcal{A}_{scope}\} \cup$$
$$\{(SE_x, snapshot_x) \mid x \in \mathcal{A}_{scope}\} \cup$$ – to_continue $\rightarrow$ to_stop
$$\{(SKPF_x, no\_snapshot_x) \mid x \in \mathcal{A}_{scope}\} \cup$$ – skipped scope

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

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

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

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

$$\{(jcf_y, E_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{jf} \wedge y \in \mathcal{A}^{target} \wedge \mathsf{trigger}_{jf}(y) = e\} \cup$$
$$\{(r_y, E_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{jf} \wedge$$

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

$$\{(SB_x, to\_invoke_x^{\mathrm{TH}}) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$ – ready to invoke TH
$$\{(to\_invoke_x^{\mathrm{TH}}, SE_x) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$ – not to invoke TH
$$\{(to\_stop_x, E_{y,e}) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{scope} \cap \mathcal{A}^{directenc}(x) \wedge$$
$$e \in \mathcal{E}_{termination} \wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$$
$$\{(E_{y,e}, to\_stop_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{scope} \cap \mathcal{A}^{directenc}(x) \wedge$$
$$e \in \mathcal{E}_{termination} \wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$$ – 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
$$\{invoked_{x,e}^{\mathrm{FH}}, HB_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault}\} \cup$$ – to start FH/TH
$$\{(to\_invoke_{x,e}^{\mathrm{EH}}, HB_{x,e'}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{normal} \wedge$$
$$(x, e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$$ – not invoke any more EH
$$\{(to\_invoke_{x,e}^{\mathrm{FH}}, HB_{x,e'}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{fault} \wedge$$
$$(x, e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times (\mathcal{E}^{ft} \backslash \{e\})\} \cup$$ – not invoke any other FH
$$\{(to\_invoke_x^{\mathrm{TH}}, HB_{x,e'}) \mid (x, e') \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}^{ft}\} \cup$$ – 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

$$\{(A_c, to\_invoke_x^{\mathrm{CH}}) \mid c \in \mathcal{A}_{compensate} \wedge x \in \mathcal{A}_{scope} \wedge$$ – (∗ compensation handler ∗)
$$\mathsf{scp}_c(\mathsf{trigger}_c(c)) = x\} \cup$$ – intend to invoke CH
$$\{(snapshot_x, HB_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$$ – CH is available
$$\{(HB_{x,e}, no\_snapshot_x) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$$ – CH to be unavailable
$$\{(to\_invoke_x^{\mathrm{CH}}, HB_{x,e}) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$$
$$\{(HB_{x,e}, r_y) \mid (x, e, y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation} \times \mathcal{A}\} \cup$$ – CH start
$$\{(f_y, HF_{x,e}) \mid (x, e, y) \in \mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation} \times \mathcal{A}\} \cup$$
$$\{(HF_{x,e}, endc_x) \mid (x, e) \in \pi_{1,2}\mathsf{HR} \cap \mathcal{A}_{scope} \times \mathcal{E}_{compensation}\} \cup$$ – CH finish

$$\{(no\_snapshot_x, NOP_x) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$ – CH is unavailable
$$\{(to\_invoke_x^{\mathrm{CH}}, NOP_x) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$
$$\{(NOP_x, endc_x) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$ – "no-op" in compensation
$$\{(NOP_x, no\_snapshot_x) \mid x \in \mathcal{A}_{scope} \backslash \{\texttt{process}\}\} \cup$$ – CH remains unavailable
$$\{(endc_x, PST_c) \mid x \in \mathcal{A}_{scope} \wedge c \in \mathcal{A}_{compensate} \wedge$$
$$\mathsf{scp}_c(\mathsf{trigger}_c(c)) = x\} \cup$$ – resume compensate activity

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

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

$\{(A_t, lsf_l) \mid t \in \mathcal{A}_{throw} \wedge \exists_{x \in \mathcal{A}_{scope}}((x, \mathsf{trigger}_{tf}(t)) \in \pi_{1,2}\mathsf{HR} \wedge$
$l \in \mathcal{L}^{ft}_{OUT}(x) \backslash \mathcal{L}_{OUT}(\mathcal{A}^{ft}_h(x, \mathsf{trigger}_{tf}(t))))\} \cup$ – DPE: thrown fault

– (∗∗ prepare for update ∗∗)[10]

$\{(PRE_{\mathtt{process}}, no\_snapshot_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – initialize scope status
$\{(s_x, CLT\_SNP\_NRM_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_SNP\_NRM_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(snapshot_x, CLT\_SNP\_NRM_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot status
$\{(CLT\_SNP\_NRM_x, scp\_stat\_collected^{\mathrm{NRM}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on normal path
$\{(s_x, CLT\_NSNP\_NRM_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_NSNP\_NRM_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(no\_snapshot_x, CLT\_NSNP\_NRM_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot status
$\{(CLT\_NSNP\_NRM_x, scp\_stat\_collected^{\mathrm{NRM}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on normal path
$\{(s_x, CLT\_SNP\_SKP_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_SNP\_SKP_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(snapshot_x, CLT\_SNP\_SKP_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot status
$\{(CLT\_SNP\_SKP_x, scp\_stat\_collected^{\mathrm{SKP}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on skip path
$\{(s_x, CLT\_NSNP\_SKP_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_NSNP\_SKP_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(no\_snapshot_x, CLT\_NSNP\_SKP_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot status
$\{(CLT\_NSNP\_SKP_x, scp\_stat\_collected^{\mathrm{SKP}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on skip path
$\{(s_x, CLT\_SNP\_SJF_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_SNP\_SJF_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(snapshot_x, CLT\_SNP\_SJF_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect snapshot status
$\{(CLT\_SNP\_SJF_x, scp\_stat\_collected^{\mathrm{SJF}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on SJF path
$\{(s_x, CLT\_NSNP\_SJF_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(CLT\_NSNP\_SJF_x, s_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(no\_snapshot_x, CLT\_NSNP\_SJF_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – collect no_snapshot status
$\{(CLT\_NSNP\_SJF_x, scp\_stat\_collected^{\mathrm{SJF}}_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – on SJF path
$\{(scp\_stat\_collected^{\mathrm{NRM}}_x, SB_x) \mid x \in \mathcal{A}_{scope}\} \cup$
$\{(scp\_stat\_collected^{\mathrm{SKP}}_x, SKPF_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – clean snapshot/no_snapshot
$\{(scp\_stat\_collected^{\mathrm{SJF}}_x, SJFF_x) \mid x \in \mathcal{A}_{scope}\} \cup$ – before another execution

$\{(to\_continue_x, A_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – (∗ termination due to a fault ∗)
$\{(A_y, to\_continue_x) \mid x \in \mathcal{A}_{scope} \wedge$
$y \in (\mathcal{A}^{basic} \backslash \mathcal{A}_{throw}) \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(A_t, to\_continue_x) \mid x \in \mathcal{A}_{scope} \wedge$
$t \in \mathcal{A}_{throw} \cap \mathcal{A}^{nfct}(x) \cap (\mathcal{A} \backslash \mathcal{A}^{directenc}(x))\} \cup$ – can perform basic activity
$\{(to\_continue_x, SET\_LST_l) \mid x \in \mathcal{A}_{scope} \wedge$
$(\exists y \in \mathcal{A}^{nfct}(x) \cap \mathcal{A}^{directenc}(x)$
such that $l \in \mathcal{L}_{out}(y))\} \cup$
$\{(SET\_LST_l, to\_continue_x) \mid x \in \mathcal{A}_{scope} \wedge$
$(\exists y \in \mathcal{A}^{nfct}(x) \cap \mathcal{A}^{directenc}(x)$
such that $l \in \mathcal{L}_{out}(y))\} \cup$ – can set link status to true
$\{(to\_continue_x, E_{y,e}) \mid x \in \mathcal{A}_{scope} \wedge e \in \mathcal{E}^{normal} \wedge$
$y \in (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \cap (\{x\} \cup \mathcal{A}^{nfct}(x))$
$\wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$
$\{(E_{y,e}, to\_continue_x) \mid x \in \mathcal{A}_{scope} \wedge e \in \mathcal{E}^{normal} \wedge$
$y \in (\mathcal{A}_{scope} \cup \mathcal{A}_{pick}) \cap (\{x\} \cup \mathcal{A}^{nfct}(x))$
$\wedge (y, e) \in \pi_{1,2}\mathsf{HR}\} \cup$ – can process e_{normal}

[10] To keep $PN_{\mathcal{W}}$ 1-safe, we assume that when a scope is executed multiple times, the status of the scope is updated upon each execution.

$\{(to\_continue_x, LB_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(LB_y, to\_continue_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{while} \cap \mathcal{A}^{nfct}(x)\} \cup$ – can continue loop in while

$\{(to\_stop_x, IGN\_STP_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(IGN\_STP_y, to\_stop_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(jcf_y, IGN\_STP_y) \mid y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$ – ignore join failure
$\{(r_y, IGN\_STP_y) \mid y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$
$\{(IGN\_STP_y, s_y) \mid y \in \mathcal{A}^{target} \cap \mathcal{A}_{TJF} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$ – continue dry-run of activity
$\{(to\_stop_x, BYP\_STP_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$
$\{(BYP\_STP_y, to\_stop_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}^{basic} \cap \mathcal{A}^{nfct}(x)\} \cup$ – to bypass basic activity
$\{(s_y, BYP\_STP_y) \mid y \in \mathcal{A}^{basic} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$
$\{(BYP\_STP_y, c_y) \mid y \in \mathcal{A}^{basic} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$ – bypassed basic activity

$\{(to\_stop_x, BYP\_STP_y) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{nfct}(x) \cup$ – (** bypass pick **)
$\{(BYP\_STP_y, to\_stop_x) \mid x \in \mathcal{A}_{scope} \wedge y \in \mathcal{A}_{pick} \cap \mathcal{A}^{nfct}(x) \cup$
$\{(s_y, BYP\_STP_y) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$
$\{(BYP\_STP_y, to\_c_y^{\mathrm{STP}}) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$ – to bypass e_{normal} & branches
$\{(BYP\_STP_y, to\_skip_z) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x)) \wedge z \in \mathsf{children}(y)\} \cup$
$\{(skipped_z, BYPF\_STP_y) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x)) \wedge z \in \mathsf{children}(y)\} \cup$ – bypassed all branches
$\{(to\_c_y^{\mathrm{STP}}, BYPF\_STP_y) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$
$\{(BYPF\_STP_y, c_y) \mid y \in \mathcal{A}_{pick} \cap (\bigcup_{x \in \mathcal{A}_{scope}} \mathcal{A}^{nfct}(x))\} \cup$ – bypassing finish

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

$\{(no\_exit, LB_x) \mid x \in \mathcal{A}_{while}\} \cup$
$\{(LB_x, no\_exit) \mid x \in \mathcal{A}_{while}\} \cup$ – check no_exit in while

$\{(to\_exit, BYP\_EXT_x) \mid x \in \mathcal{A}^{basic}\} \cup$
$\{(BYP\_EXT_x, to\_exit) \mid x \in \mathcal{A}^{basic}\} \cup$ – to bypass basic activity
$\{(s_x, BYP\_EXT_x) \mid x \in \mathcal{A}^{basic}\} \cup$
$\{(BYP\_EXT_x, c_x) \mid x \in \mathcal{A}^{basic}\} \cup$ – bypassed basic activity

$\{(to\_exit, BYP\_EXT_x) \mid x \in \mathcal{A}_{pick}\} \cup$ – (** bypass pick **)
$\{(BYP\_EXT_x, to\_exit) \mid x \in \mathcal{A}_{pick}\} \cup$
$\{(s_x, BYP\_EXT_x) \mid x \in \mathcal{A}_{pick}\} \cup$
$\{(BYP\_EXT_x, to\_c_x^{\mathrm{EXT}}) \mid x \in \mathcal{A}_{pick}\} \cup$ – bypassed e_{normal}
$\{(BYP\_EXT_x, to\_skip_y) \mid x \in \mathcal{A}_{pick} \wedge y \in \mathsf{children}(x)\} \cup$
$\{(skipped_y, BYPF\_EXT_x) \mid x \in \mathcal{A}_{pick} \wedge y \in \mathsf{children}(x)\} \cup$ – skipped all branches
$\{(to\_c_x^{\mathrm{EXT}}, BYPF\_EXT_x) \mid x \in \mathcal{A}_{pick}\} \cup$
$\{(BYPF\_EXT_x, c_x) \mid x \in \mathcal{A}_{pick}\} \cup$ – bypassing finish

References

[1] W.M.P. van der Aalst, Verification of workflow nets, in: P. Azéma, G. Balbo (Eds.), Proceedings of 18th International Conference on Application and Theory of Petri Nets, in: Lecture Notes in Computer Science, vol. 1248, Springer-Verlag, Toulouse, France, June 1997, pp. 407–426.

[2] W.M.P. van der Aalst, M. Dumas, C. Ouyang, A. Rozinat, H.M.W. Verbeek, Conformance checking of service behavior, in: Middleware for Service-Oriented Computing, ACM Transactions on Internet Technology (February) 2008 (in press) (special issue). A technical report version is available via http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-25.pdf.
[3] W.M.P. van der Aalst, K.M. van Hee, Workflow Management: Models, Methods, and Systems, MIT press, Cambridge, Massachusetts, 2002.
[4] W.M.P. van der Aalst, A.H.M. ter Hofstede, YAWL: Yet another workflow language, Information Systems 30 (4) (2004) 245–275.
[5] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services: Concepts, Architectures and Applications, Springer-Verlag, 2003.
[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana, Business Process Execution Language for Web Services Version 1.1. BEA Systems, IBM Corporation, Microsoft Corporation, SAP AG, Siebel Systems, May 2003.
[7] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C.K. Liu, S. Thatte, P. Yendluri, and A. Yiu (Eds.), Web Services Business Process Execution Language Version 2.0, Working Draft, WS-BPEL TC OASIS, May 2005. Available via http://www.oasis-open.org/committees/download.php/12791/.
[8] J. Billington, M. Diaz, G. Rozenberg (Eds.), Application of Petri Nets to Communication Networks, in: Lecture Notes in Computer Science, vol. 1605, Springer-Verlag, 1999.
[9] F. Casati, M.-C. Shan, Dynamic and adaptive composition of e-services, Information Systems 26 (3) (2001) 143–162.
[10] J. Dehnert, A methodology for workflow modelling: From business process modelling towards sound workflow specification, Ph.D. Thesis, Technische Universität Berlin, Berlin, Germany, August 2003.
[11] R. Farahbod, U. Glässer, 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.
[12] A. Ferrara, Web services: A process algebra approach, in: Proceedings of 2nd International Conference on Service Oriented Computing, ACM Press, New York, NY, USA, 2004, pp. 242–251.
[13] J.A. Fisteus, L.S. Fernández, C.D. Kloos, Formal verification of BPEL4WS business collaborations, in: Proceedings of 5th International Conference on Electronic Commerce and Web Technologies, EC-Web'04, in: Lecture Notes in Computer Science, vol. 3180, Springer-Verlag, Zaragoza, Spain, August 2004, pp. 76–85.
[14] H. Foster, S. Uchitel, J. Magee, J. Kramer, Model-based verification of web service composition, in: Proceedings of 18th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, Montreal, Canada, October 2003, pp. 152–161.
[15] H. Foster, S. Uchitel, J. Magee, J. Kramer, Tool support for model-based engineering of web service compositions, in: Proceedings of 2005 IEEE International Conference on Web Services, ICWS 2005, IEEE Computer Society, Orlando, FL, USA, July 2005, pp. 95–102.
[16] X. Fu, T. Bultan, J. Su, Analysis of interacting BPEL web services, in: Proceedings of 13th International Conference on World Wide Web, ACM Press, New York, NY, USA, 2004, pp. 621–630.
[17] X. Fu, T. Bultan, J. Su, WSAT: A tool for formal analysis of web services, in: Proceedings of 16th International Conference on Computer Aided Verification, CAV, in: Lecture Notes in Computer Science, vol. 3114, Springer-Verlag, 2004, pp. 510–514.
[18] R.J. van Glabbeek, W.P. Weijland, Branching time and abstraction in bisimulation semantics, Journal of the ACM 43 (3) (1996) 555–600.
[19] S. Hinz, K. Schmidt, C. Stahl, Transforming BPEL to Petri nets, in: W.M.P. van der Aalst, B. Benatallah, F. Casati, F. Curbera (Eds.), Proceedings of the International Conference on Business Process Management, BPM 2005, in: Lecture Notes in Computer Science, vol. 3649, Springer-Verlag, Nancy, France, September 2005, pp. 220–235.
[20] B. Kiepuszewski, A.H.M. ter Hofstede, W.M.P. van der Aalst, Fundamentals of control flow in workflows, Acta Informatica 39 (3) (2003) 143–209.
[21] M. Koshkina, 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.
[22] N. Lohmann, P. Massuthe, C. Stahl, D. Weinberg, Analyzing interacting BPEL processes, in: S. Dustdar, J.L. Fiadeiro, A.P. Sheth (Eds.), Proceedings of the International Conference on Business Process Management, BPM 2006, Vienna, Austria, September 2006, in: Lecture Notes in Computer Science, vol. 4102, Springer-Verlag, 2006, pp. 17–32.
[23] A. Martens, Verteilte Geschäftsprozesse—Modellierung und Verifikation mit Hilfe von Web Services (In German), Ph.D. Thesis, Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany, 2003.
[24] A. Martens, Analyzing web service based business processes, in: M. Cerioli (Ed.), Proceedings of 8th International Conference on Fundamental Approaches to Software Engineering, FASE 2005, in: Lecture Notes in Computer Science, vol. 3442, Springer-Verlag, 2005, pp. 19–33.
[25] J. Martinez, M. Silva, A simple and fast algorithm to obtain all invariants of a generalised Petri net, in: C. Girault, W. Reisig (Eds.), Application and Theory of Petri Nets: Selected Papers from the First and the Second European Workshop, in: Informatik Fachberichte, vol. 52, Springer-Verlag, Berlin, 1982, pp. 301–310.
[26] T. Murata, Petri nets: Properties, analysis and applications, Proceedings of the IEEE 77 (4) (1989) 541–580.
[27] C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, WofBPEL/BPEL2PNML, in: Tools Sessions in 2nd International Workshop on Web Services and Formal Methods, WS-FM05, Versailles, France, 2–3 September 2005.
[28] C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, Formal semantics and analysis of control flow in WS-BPEL, Technical Report BPM-05-15, BPMcenter.org, September 2005. Available via http://is.tm.tue.nl/staff/wvdaalst/BPMcenter/reports/2005/BPM-05-15.pdf.
[29] C. Ouyang, H.M.W. Verbeek, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, WofBPEL: A tool for automated analysis of BPEL processes, in: Proceedings 3rd International Conference on Service Oriented Computing (Demonstration Session), in: Lecture Notes in Computer Science, vol. 3826, Springer-Verlag, Amsterdam, The Netherlands, December 2005, pp. 484–489.
[30] J.L. Peterson, Petri Net Theory and the Modeling of Systems, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.
[31] C. Stahl, A Petri net semantics for BPEL, Technical Report 188, Humboldt-Universität zu Berlin, June 2005.

[32] H.M.W. Verbeek, W.M.P. van der Aalst, Analyzing BPEL processes using Petri nets, in: D.C. Marinescu (Ed.), Proceedings of 2nd International Workshop on Applications of Petri Nets to Coordination, Workflow and Business Process Management, June 2005, pp. 59–78.
[33] H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, Verifying workflows with cancellation regions and OR-joins: An approach based on invariants, in: BETA Working Paper Series, WP 156, Eindhoven University of Technology, Eindhoven, The Netherlands, 2006.
[34] H.M.W. Verbeek, T. Basten, W.M.P. van der Aalst, Diagnosing workflow processes using Woflan, The Computer Journal 44 (4) (2001) 246–279.