# A Petri Net Semantics for BPEL

Christian Stahl
Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
D-10099 Berlin

stahl@informatik.hu-berlin.de

We present a pattern-based Petri net semantics for the Business Process Execution Language for Web Services (BPEL). Our semantics is complete – it covers the standard behaviour of BPEL as well as the exceptional behaviour (e.g. faults, events, compensation). Therefore every business process specified in BPEL can be transformed into a Petri net.

# Contents

# List of Figures

# 1 Introduction

The *Business Process Execution Language for Web Services* (BPEL) is part of ongoing activities to standardize a family of technologies for web services. A textual specification [CGK$^+$03] appeared in 2003 and is subject to further revisions. The language contains features from previous languages, for instance IBM's WSFL [Ley01] and Microsoft's XLANG [Tha01]. The textual specification is, of course, not suitable for formal methods such as computer aided verification. With computer aided verification, in particular model checking, it would be possible to decide crucial properties such as composability of processes, soundness, and controllability (the possibility to communicate with the process such that the process terminates in a desired end state). For a formal treatment, it is necessary to resolve the ambiguities and inconsistencies of the language which occurred particularly due to the unification of rather different concepts in WSFL and XLANG.

Several groups have proposed formal semantics for BPEL. Among the existing attempts, there are some based on finite state machines [FFK04, FBS04], process algebras [Fer04], and abstract state machines [Fah05, FR05, FGV04]. Though all of them are successful in unravelling weaknesses in the informal specification, they are of different significance for formal verification. The semantics based on abstract state machines are feature-complete. However, Petri nets provide a much broader basis for computer aided verification than abstract state machines. Most of the other approaches typically do not support some of BPEL's most interesting features such as fault handling, compensation handling, and event handling.

In this paper, we consider a *Petri net semantics* for BPEL. The semantics is *complete* (i.e., covers all the standard and exceptional behaviour of BPEL), and *formal* (i.e., feasible for model checking). With Petri nets, several elegant technologies such as the theory of workflow nets [vdA98], a theory of controllability [Mar04, Sch04], a long list of verification techniques [Sch00] and tools [RWL$^+$03, SR00, Sch00] become directly applicable. The Petri net semantics provides patterns for each BPEL activity. Compound activities contain slots for the patterns of their subactivities. This way, it is possible to translate BPEL processes automatically into Petri nets. Using high-level Petri nets, data aspects can be fully incorporated while these aspects can as well be ignored by switching to low-level Petri nets.

The paper is structured as follows: In Sec. 2 we explain the general concepts of BPEL. Next, in Sec. 3 we introduce the principles of our Petri net semantics for BPEL. In the following sections we transform BPEL's elementary activities (Sec. 4), structured activities (Sec. 5), and the semantics of links (Sec. 6). BPEL's specific structured activity, scope, is transformed in Sec. 7. Afterwards we explain the transformation of the complex components of a scope: event handler, fault handler, and compensation handler (Sections 8 – 10). Finally, conclusions are drawn in Sec. 11.

## 2 Introduction to BPEL

BPEL is a language for describing the behaviour of business processes based on web services. Such a business process can be described in two different ways: either as an *executable business process* or as a *business protocol*. An executable business process which is the focus of this paper models the behaviour and the interface of a *partner* (a participant), in a business interaction. A business protocol, in contrast, only models the interface and the message exchange of a partner. The rest of its internal behaviour is hidden. Throughout this paper we will use the term *BPEL process* instead of "executable business process specified in BPEL". To execute a BPEL process means to create an *instance* of this process which is executed.

For the specification of the internal behaviour of a business process, BPEL provides two kinds of *activities*. An activity is either an *elementary activity* or a *structured activity*. The set of elementary activities includes: `empty` [1] (doing nothing), `wait` (waiting for some time), `assign` (copying a value from one place to another), `receive` (waiting for a message from a partner), `invoke` (invoking a partner), `reply` (replying a message to a partner), `throw` (signalling a fault), and `terminate` (terminating the entire process instance).

A structured activity defines a causal order on the elementary activities. It can be nested with other structured activities. The set of structured activities includes: `sequence` (activities ordered sequentially), `flow` (activities ordered parallel), `while` (while loop), `switch` (selects one control path depending on data), and `pick` (selects one control path depending either on timeouts or external messages). The most important structured activity is a `scope` which links an activity to a transaction management. It provides a `fault handler`, a `compensation handler`, an `event handler`, `correlation sets` and data `variables`. A `process` is a special `scope`. More precisely, it is the outmost `scope` of the business process.

A `fault handler` is a component that provides methods to handle faults that may occur during the execution of its enclosing `scope`. In contrast, a `compensation handler` is used to reverse some effects that happened during the execution of activities. With the help of an `event handler`, external message events and specified timeouts can be handled. A `correlation set` is used for identifying the instance of a BPEL `process` only by a message's content. Thus, a `correlation set` is an identifier – more precisely, it is a collection of properties – and all messages of an instance have to contain it. It is either initialized by the first incoming or outgoing message.

Another important concept in BPEL are `links`. A `link` can be used to define an order between two concurrent activities in a `flow`. It has a *source* activity and a *target* activity. The source may specify a boolean expression, the status of the `link`. The target may also specify a boolean expression which evaluates the status of all incoming `links`. BPEL provides *dead-path-elimination* [LR99], i.e. the status of all outgoing `links` of a source activity that is not executed anymore is set to negative. Consider, for instance, an activity within a branch that is not taken in a `switch` activity.

[1] We use this type-writer font for BPEL constructs.

In the forthcoming sections we transform BPEL into *algebraic high-level Petri nets* [Rei91], a specific class of high-level Petri nets. We assume the basics of Petri nets to be known by the reader. Thus, there is no section introducing this formalism. We will only explain specific concepts of algebraic high-level Petri nets if necessary.

# 3 Transformation of BPEL into Petri Nets

Our goal is to translate every BPEL process into a Petri net. The translation is guided by the syntax of BPEL. In BPEL, a `process` is built up by plugging language constructs together. Therefore we translate each construct of the language into a Petri net. Such a net forms a *pattern* of the respective BPEL construct. Each pattern has an *interface* for joining it with other patterns as it is done with BPEL constructs. Some of the patterns are used with a parameter, e.g. there are some constructs that have inner constructs. The respective pattern must be able to carry any number of inner constructs as its equivalent in BPEL can do. We aim at keeping all properties of the constructs in the patterns. The collection of patterns forms our *Petri net semantics* for BPEL.

## 3.1 Transformation Approach

Let us have a more detailed look at the general pattern's design. Figure 1 depicts the pattern for the BPEL's `receive` activity. `receive` is responsible for receiving a partner's request. To identify whether the request is sent to this receive pattern and not to another instance of the `process`, BPEL's `receive` specifies at least one `correlation set`. The pattern in Fig. 1 presents a `receive` with one `correlation set` which is already initialized. More precisely, attribute *initiate* is set to "no". The pattern of BPEL's `receive` where a `correlation set` is initialized by the incoming message, i.e. initiate is set to "yes", is depicted in Fig. 2.

Before we discuss details of the receive pattern we give some general comments on the notion of patterns. Firstly, we use the common graphical notations for Petri nets. Places and transitions are labelled with an identifier, e.g. p1 [2] or t1 which are depicted (contrary to common notation) inside the respective Petri net node. In addition, some nodes have a second label depicted outside the node, e.g. initial. This label is used to show the purpose of the node in the net. Secondly, a variable in small letters in arc inscriptions, e.g. fault, symbolizes a single variable and a variable in capital letters, e.g. X, symbolizes a tupel of variables. Thirdly, there are transitions, e.g. t2 that have a transition guard. Such a transition can only fire when its guard, a boolean expression, is evaluated to true. A guard is depicted (in braces) next to the transition it belongs to, e.g. {!guard}.

In general, a pattern is framed by a dashed box. Inside the frame, the structure of the corresponding BPEL construct is modelled. The interface is established by the nodes depicted directly on the frame. Positive control flows from top to bottom while communication between processes flows horizontally. In Fig. 1 the positive control flow starts with a token on initial and it ends either with a token on finish or failed. Outside the frame, there are external objects, e.g. obj1. An object is either a place of a scope pattern (`variable`, `correlation set`) or of the process pattern (channel). An activity's pattern as the receive pattern in Fig. 1 relates to those places. The label on the top of an object defines its sort whereas the role is defined at the bottom of the object. A

[2] We use this serif-free font for labels in a Figure.

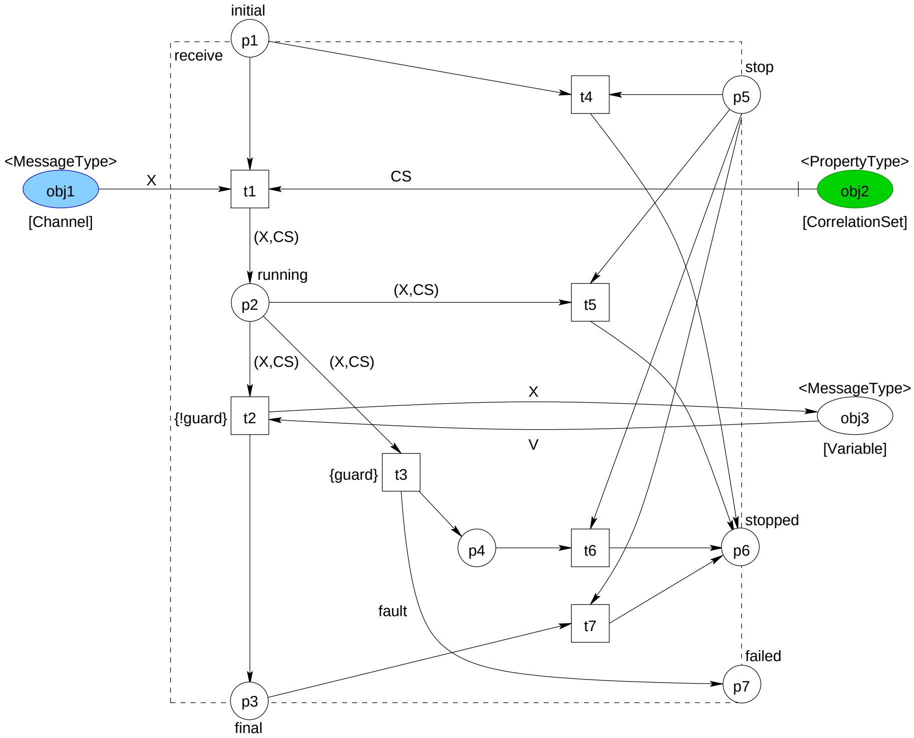


Figure 1: Pattern for BPEL's `receive` in case of initiate="no".

sort is the domain of the tokens lying on and arriving at this place. The object's role is independent of its sort.

When the pattern depicted in Fig. 1 is activated, it is executed in two steps. Firstly, the message is taken from the channel (obj1) and the `correlation set` (obj2) is read[3] (t1). Both values are saved in variables X and CS, respectively. In the second step this information is analyzed. Either the message is saved in the `variable` (t2) or a fault occurs (t3) because of a mismatch between the values of the `receive`'s `correlation set` and the `correlation set` in the message or some other error. With it variable V holds the old value of obj3 and fault holds the fault information. In both cases, the pattern is finished.

The meaning of place stop, stopped and failed in Fig. 1 needs to be explained. In BPEL, a process is forced to stop its positive control flow, e.g., when a fault occurs or activity `terminate` is activated. However, the BPEL specification [CGK$^+$03] tells only informally the requirements how to stop a scope. For instance, activity `receive` "is interrupted and terminated prematurely" [CGK$^+$03, p. 79]. The specification does not describe how to realize those requirements. Thus, we had to make some modelling decisions in our model: The pattern of BPEL's `scope` is extended by a stop pattern (see Sect. 9.1 for more details), which has no equivalent construct in BPEL. If a `scope` needs to be stopped, the stop pattern controls this procedure. Our idea is to remove all tokens from

[3] The arc between obj2 and t1, depicted by the vertical line, is a read arc [Web03].

the patterns, embedded in the scope pattern; thus the patterns of BPEL's activities and `event handler` contain a subnet – a so called *stop component*. In contrast, the patterns of BPEL's `compensation handler` and `fault handler` do not contain a stop component, because they both need not to be stopped. In [Sta04] we proved that every process can be stopped using stop components. In the case of Fig. 1, the stop component is established by transitions `t4` – `t7` using the interface stop and stopped. Throughout this paper, we will call this the *negative control flow* of an activity.

In order to explain how a stop component works, consider a `scope` that contains just a `receive` and the latter throws a fault. This leads to place failed being marked – the token is an object that consists of the fault's name. This place is joined with a place in the stop pattern; thus this pattern gets the control of the `scope`. First of all it stops the inner activity of the `scope` and consequently a token is produced on the `receive`'s stop place. Transition `t6` fires and stopped is marked. This place is also joined with a place in the stop pattern. In contrast, transitions `t4`, `t5`, `t7` consume the token on stop by stopping the receive pattern wherever the control flow is in this pattern. As a result, a token is produced on stopped, too. One might assume that `t4` obtains priority before `t1` and `t5` before `t2`. Indeed, this would destroy the model's asynchronous behaviour without changing the possible set of runs. We use this asynchronous behaviour in our patterns to model the aspect that sending the stop signal needs time, too. Consider, for instance, two receive patterns executed sequentially. It is possible that the first `receive` is finished (and so the second `receive` is activated) exactly in the moment signal stop is sent. In our patterns, however, this possibility is taken into account. Alternatively, a different modelling approach is possible: A transition of the receive pattern's positive control flow is only enabled when no fault has been occurred in the surrounding scope pattern. This fact could be modelled by a place marked when no fault has been occurred. But this, of course, would destroy the asynchronous character of any BPEL process.

As we already mentioned above, in BPEL it is possible to define an activity `receive` where a `correlation set` is initialized by the incoming message. Therefore BPEL's `receive` specifies the attribute initiate, which is either set to "yes" or "no". That means, the `correlation set` can either be set or be read. Thus, for all constructs that specify initiate we have built two patterns, one for "yes" and one for "no". Figure 2 depicts BPEL's `receive` in case of initiate="yes". In contrast to the pattern shown in Fig. 1, it takes a message from the channel and the `correlation set` is initialized, while `c` is a function extracting the `correlation set` from a message. Afterwards, either the message is stored in a `variable` or a fault is thrown as explained before in Fig. 1.

## 3.2 Objects

Next, we present the four objects (*message*) *channel*, *correlation set*, *variable*, and *clock* used throughout the patterns in more detail.

A message can only be sent or received through a message channel in BPEL. In a BPEL's process instance a channel is defined by a `portType`, an `operation`, the direction of the communication (input or output) and a `partnerLink`. The latter is automatically mapped to a concrete address. In our model a message channel is defined

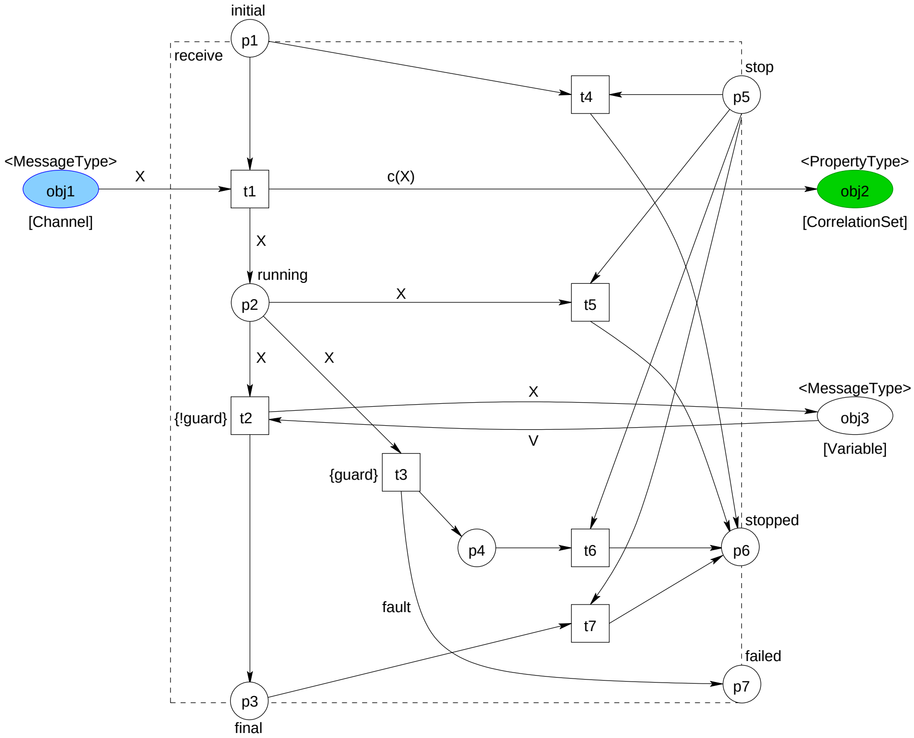


Figure 2: Pattern for BPEL's `receive` in case of initiate="yes".

by a role [Channel] and a sort <MessageType>. Thereby the information about BPEL's channel definition is encoded into the role. A channel will be depicted on the right of the pattern's structure, if the `process` sends a request to a partner. Otherwise, if the `process` is requested by a partner, it is depicted on the left side.

A `correlation set` is used for identifying the instance of a BPEL process only by the message's content. Thus, a `correlation set` is an identifier – more precisely, it is a collection of properties – and all messages of an instance have to contain it. It is initialized by the first incoming or outgoing message. In our model a `correlation set` is defined by a role [CorrelationSet] and a sort <propertyType>. If the `correlation set` is already initialized, it is only read (see Fig. 1). Therefore the token is permanently on the respective place. Otherwise, the `correlation set` has to be initialized by the first incoming or outgoing message. Throughout the paper we will use a function `c` (see Fig. 2) which extracts the `correlation set` of an incoming or outgoing message (a token of sort <propertyType>) produced on the respective object. We do not define this function and only refer to the work of Fahland [Fah05], modelling data aspects in more detail.

In BPEL a `variable` is used for either holding data or a message. An activity only gets a copy of the `variable`'s value, but a `variable` can be overwritten (just a part or the whole value). In our model, a `variable` is a place defined by the role [Variable] and the data it holds is modelled by a token. So the `variable`'s sort is either <Data> or

<MessageType>. To get a copy of the `variable`'s value we use a read-arc. Overwriting means that the token is first consumed and afterwards it is edited and produced, i.e. a loop is used in the respective patterns. In order to realize access on non-initialized `variables` each `variable` is preinitialized to zero. This is done for syntactical reasons only, whereas BPEL's semantics is not changed.

Furthermore we need a timer to model waiting, for instance. We decided to use a system clock, which is modelled by a place. The token's value – the time – is incremented continuously. This clock is defined globally in the process pattern and all constructs of the `process` can read it. Therefore the outgoing arcs of the respective object are read-arcs. Reading the token just means watching the time. The clock is defined by the role [Clock] and the sort <Time>.

# 4 Transformation of BPEL's Elementary Activities

## 4.1 Empty

The pattern for BPEL's `empty` is depicted in Fig. 3. Doing nothing, which is the activity's semantics is modelled by firing a transition (t1). The pattern's stop component is established by transitions t2 and t3.

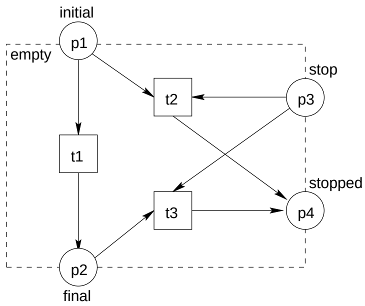


Figure 3: Pattern for BPEL's `empty`.

## 4.2 Wait

"The wait activity allows a business process to specify a delay for a certain period of time or until a certain deadline is reached" [CGK+03, p. 57]. Therefore we build two patterns – one for each wait condition – visualized in Figures 4 and 5. To model waiting,

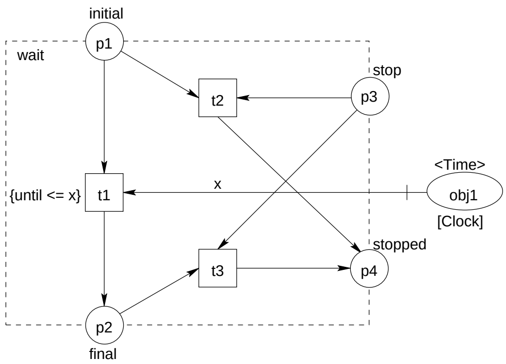


Figure 4: Pattern for BPEL's `wait` with an until condition.

we use a transition guard; thus control flows when the respective boolean expression holds.

In Fig. 4 the pattern for BPEL's `wait` with an until condition is depicted. If initial is marked and the deadline saved in variable until is lower or equal than the current time saved in variable x, the guard holds and transition t1 can fire. Furthermore the pattern's stop component is established by transitions t2 an t3.

The pattern for BPEL's `wait` with a for condition is visualized in Fig. 5. It is a little bit more complex than the former pattern. After having read a time stamp (t1), the control flow has to wait for a certain period of time saved in variable for. If the sum of x

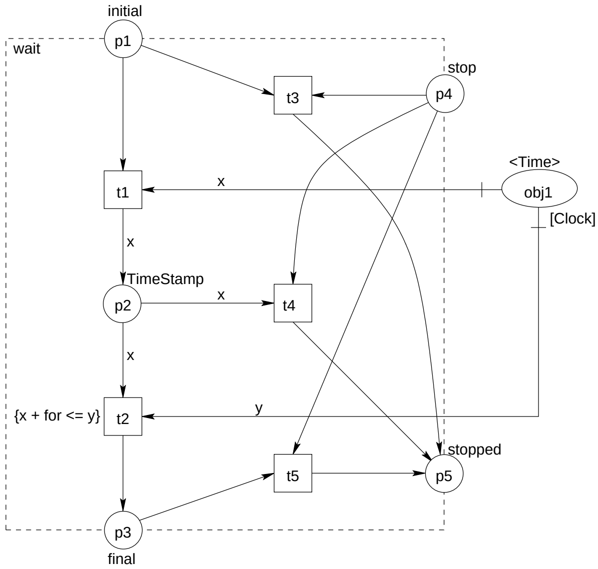


Figure 5: Pattern for BPEL's `wait` with a for expression.

and for is lower or equal than the current time stamp (y), the guard holds. Thus, t2 can fire. The pattern's stop component is established by transitions t3 – t5.

## 4.3 Assign

BPEL's activity `assign` either copies the value of a `variable` into another `variable` or it copies the value of an expression into a `variable`. Because of our abstraction of WSDL (see Sec. 3.2) it is not necessary to model all other actions of `assign` being specified in [CGK+03].

The pattern depicted in Fig. 6 copies the value of one `variable` into another. For that purpose, first obj1 is read and its value is saved in variable X. In the second step the transition guard, specifying the occurrence of a mismatched assignment or some other error, is evaluated. If it holds, a fault is thrown (t3). Otherwise, obj2 is updated by the value of X. In case of Fig. 6, the pattern's stop component is established by transitions t4 – t7.

The pattern of BPEL's `assign` that copies the value of an expression into a `variable` is depicted in Fig. 7. Again, the transition guard specifies the occurrence of a mismatched assignment or some other error. If it holds, t2 fires and a fault is thrown. Otherwise, t1 fires and the old value of the `variable` obj1 is read. f specifies a function that replaces the respective part in Y with expression x. The pattern's stop component is established by transitions t3 – t5.

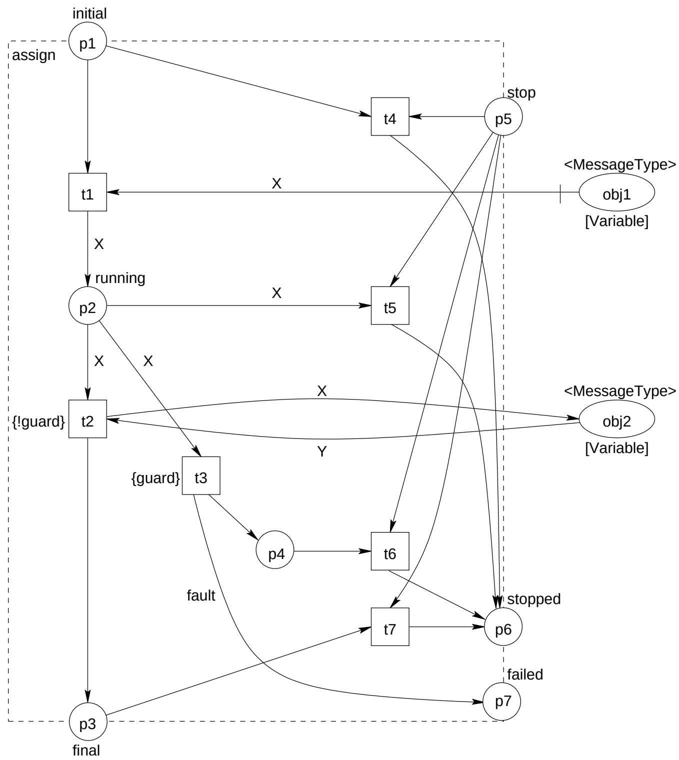


Figure 6: Pattern for BPEL's `assign` with a from `variable`.

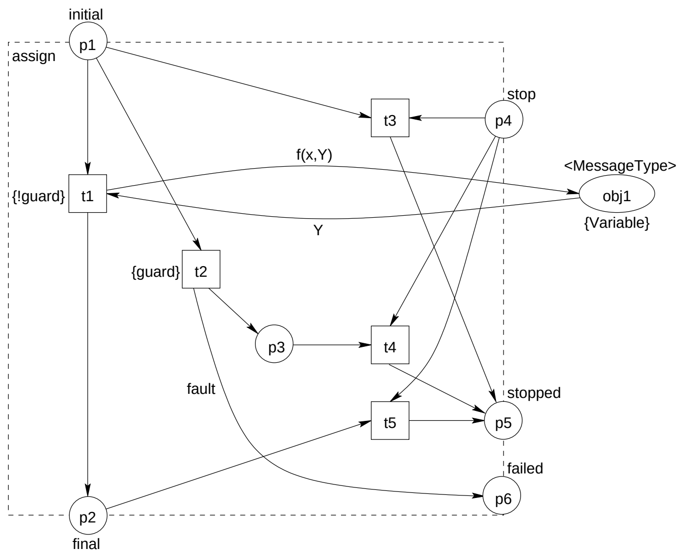


Figure 7: Pattern for BPEL's `assign` without a from `variable`.

## 4.4 Reply

BPEL's `reply` allows a business process to send a message holt in a `variable` in `reply` to a message that was received through an activity `receive`. Similar to the construct `receive` it specifies an attribute initiate. Therefore we again have built two patterns: Fig. 8 shows initiate="no" and Fig. 9 shows initiate="yes". Each of them is very similar to its respective receive pattern that was presented in Figures 1 and 2.

Let us take a look at Fig. 8. At first, a message is taken from the `variable` and the `correlation set` is read. In the next step, the transition guard, specifying the

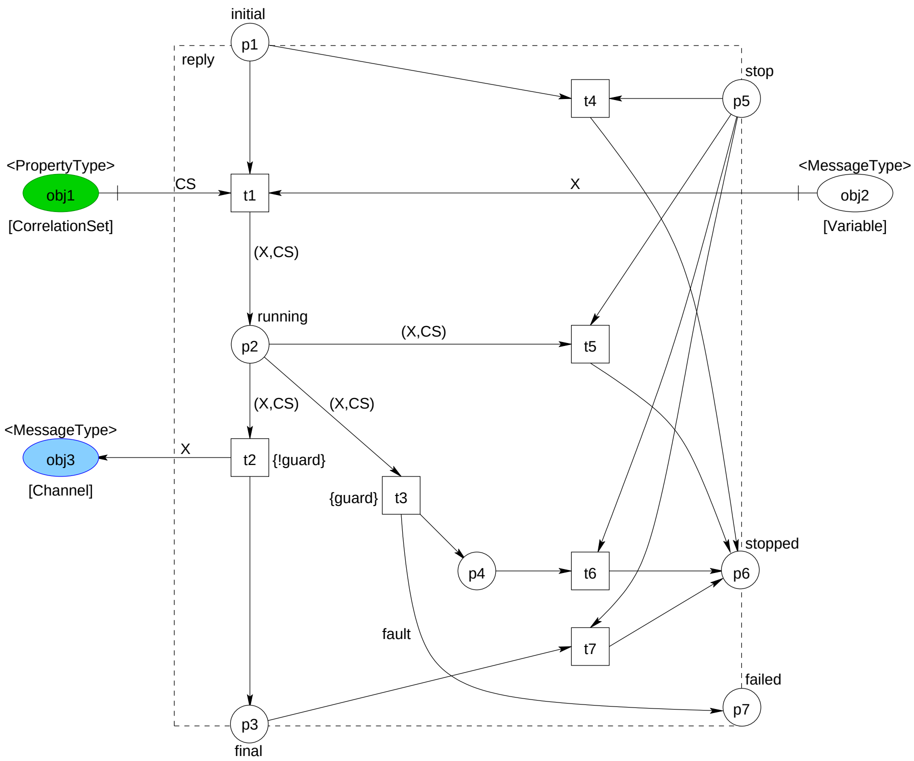


Figure 8: Pattern for BPEL's `reply` in case of initiate="no".

occurrence of a conflicting request or some other error is evaluated. If it holds, a fault occurs (`t3`). Otherwise the message is sent (`t2`).

In contrast, the pattern in Fig. 9 takes a message from the `variable` and the `correlation set` is initialized. In both figures the pattern's stop component is made up by transitions `t4` – `t7`.

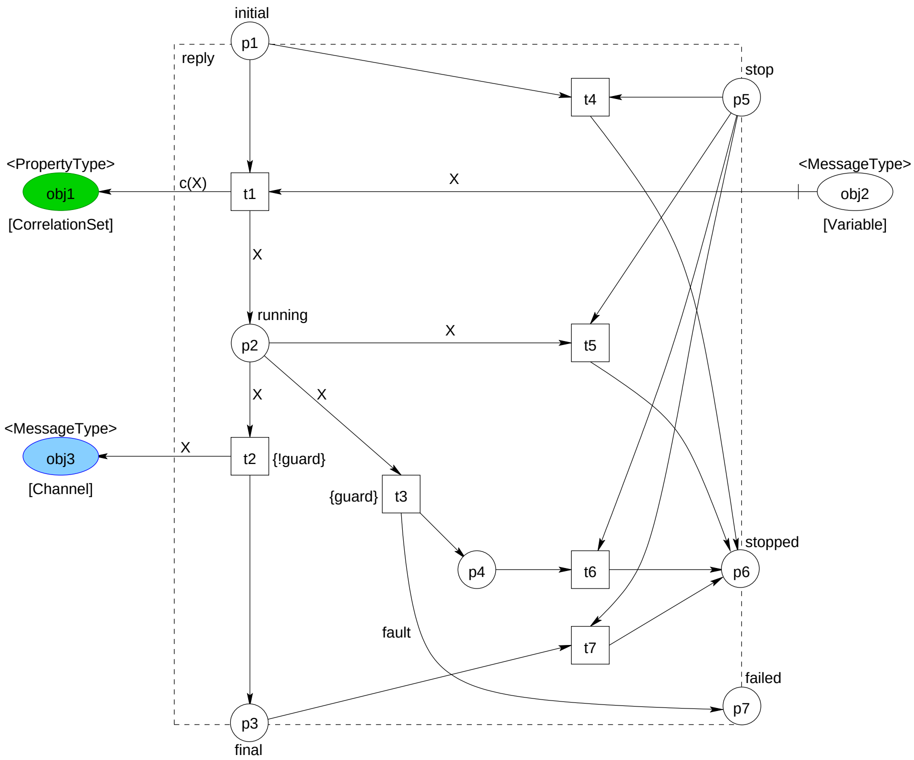


Figure 9: Pattern for BPEL's `reply` in case of initiate="yes".

## 4.5 Asynchronous Invoke

Just like the `reply` activity, an asynchronous `invoke` sends a message which is stored in a `variable`. In contrast to the `reply` activity though, the message is not sent in reply to a received message but it is sent as a request to a partner.

This activity also specifies an attribute initiate. So we also have built two patterns. They are visualized in Fig. 10 (initiate="no") and Fig. 11 (initiate="yes"). There is only one difference between each of them and its respective reply pattern. In the asynchronous invoke pattern the `variable` is placed on the left side and the channel on the right side whereas in the reply pattern it is the other way round. As explained in Sec. 3.2, if a `process` is called, the channel is placed on the right side. Otherwise, if an activity reacts to a partner's request, the corresponding channel is placed on the left side of the pattern.

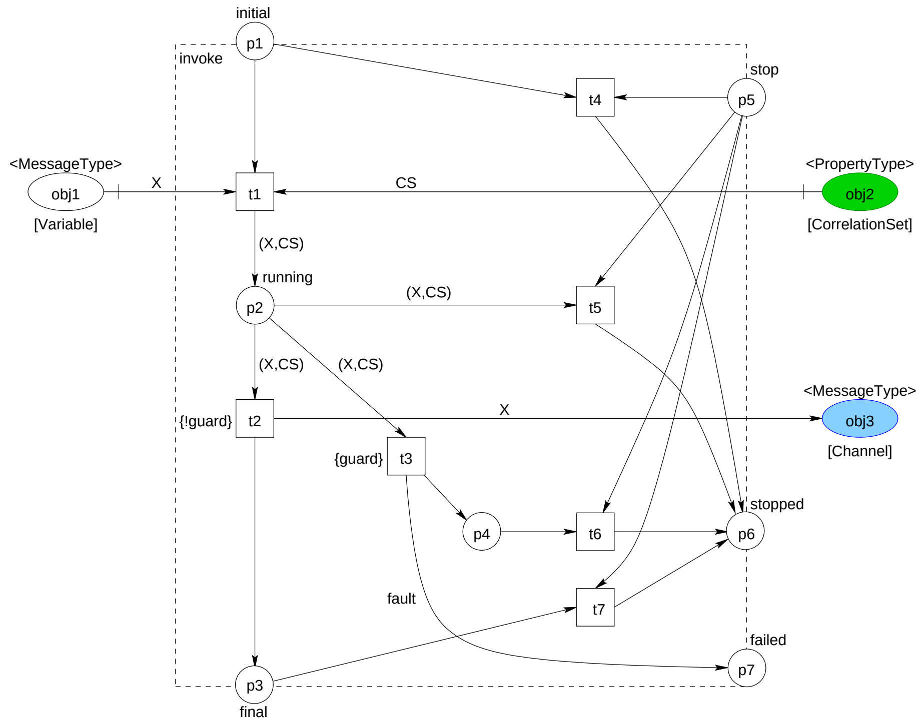


Figure 10: Pattern for BPEL's asynchronous `invoke` in case of initiate="no".

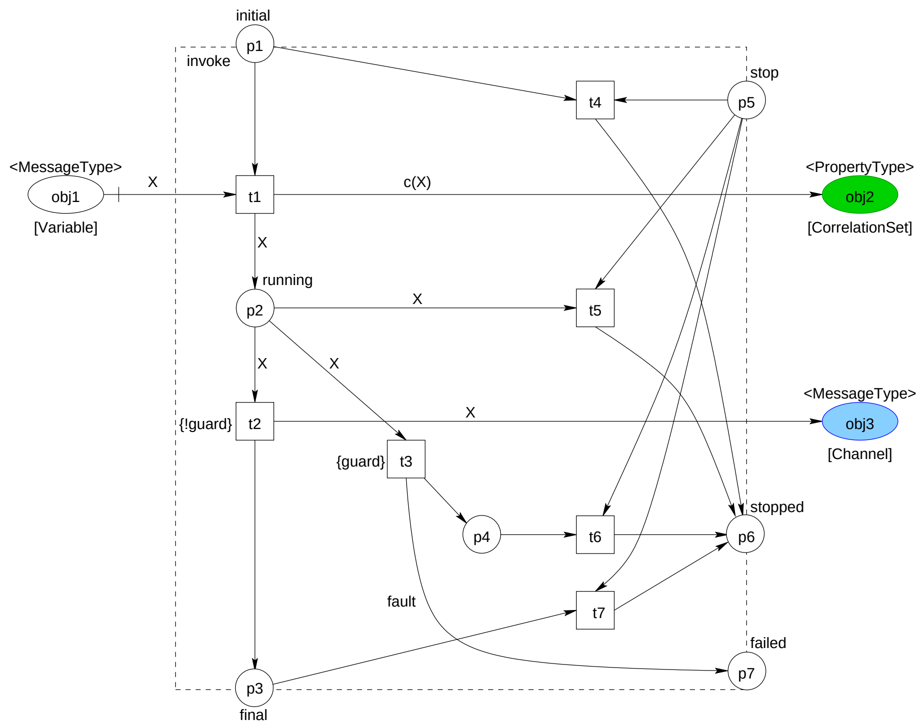


Figure 11: Pattern for BPEL's asynchronous `invoke` in case of initiate="yes".

## 4.6 Synchronous Invoke

The semantic of a synchronous `invoke` activity is to send a request to a partner and to wait for a result afterwards. In fact, it is an sequential execution of an asynchronous `invoke` and a `receive` activity. Therefore the pattern of BPEL's synchronous `invoke` is a composition of these two known patterns. It also specifies at least one attribute initiate. So again, we have built two patterns. The case of initiate="no" is visualized in

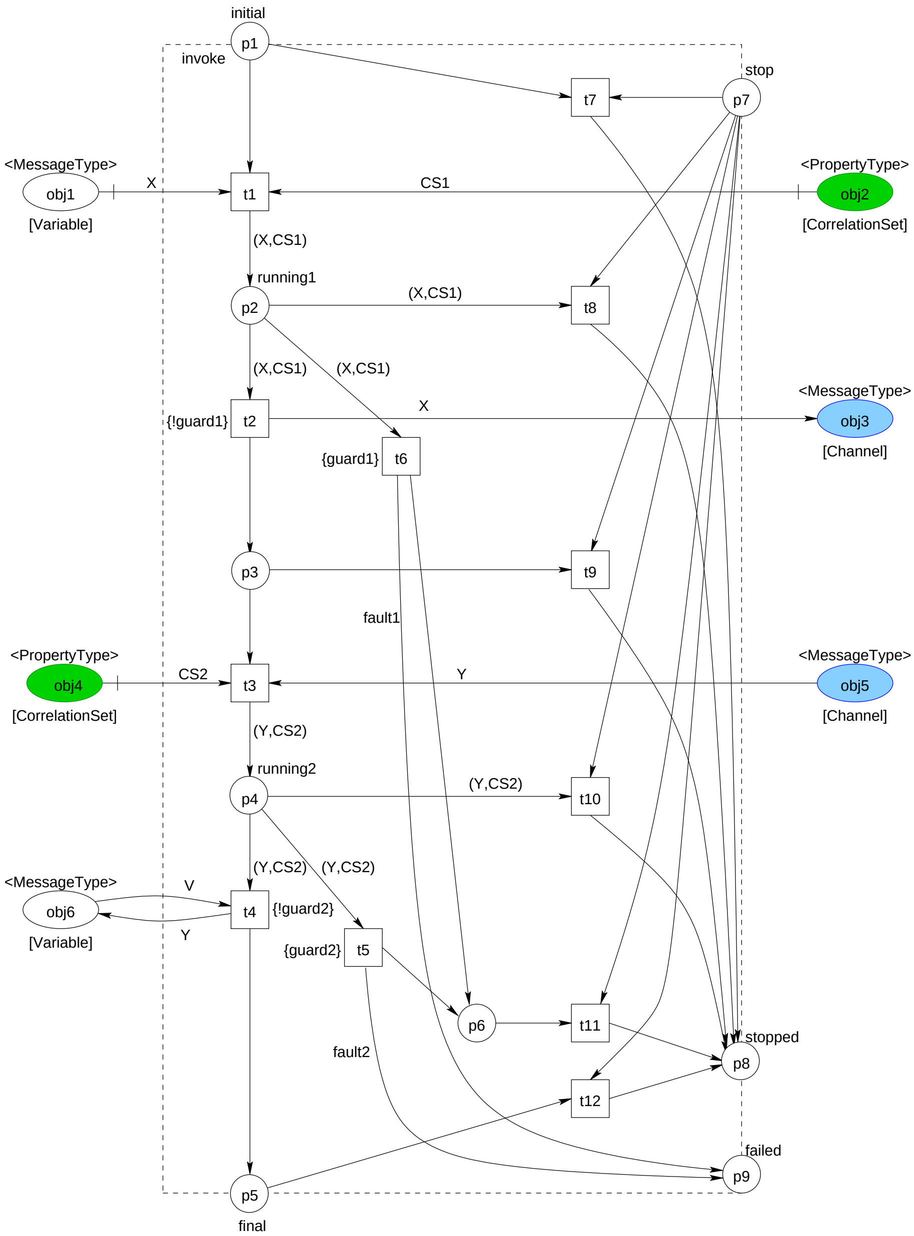


Figure 12: Pattern for BPEL's synchronous invoke in case of initiate="no".

Fig. 12. The pattern's stop component is established by transitions t7 – t12.

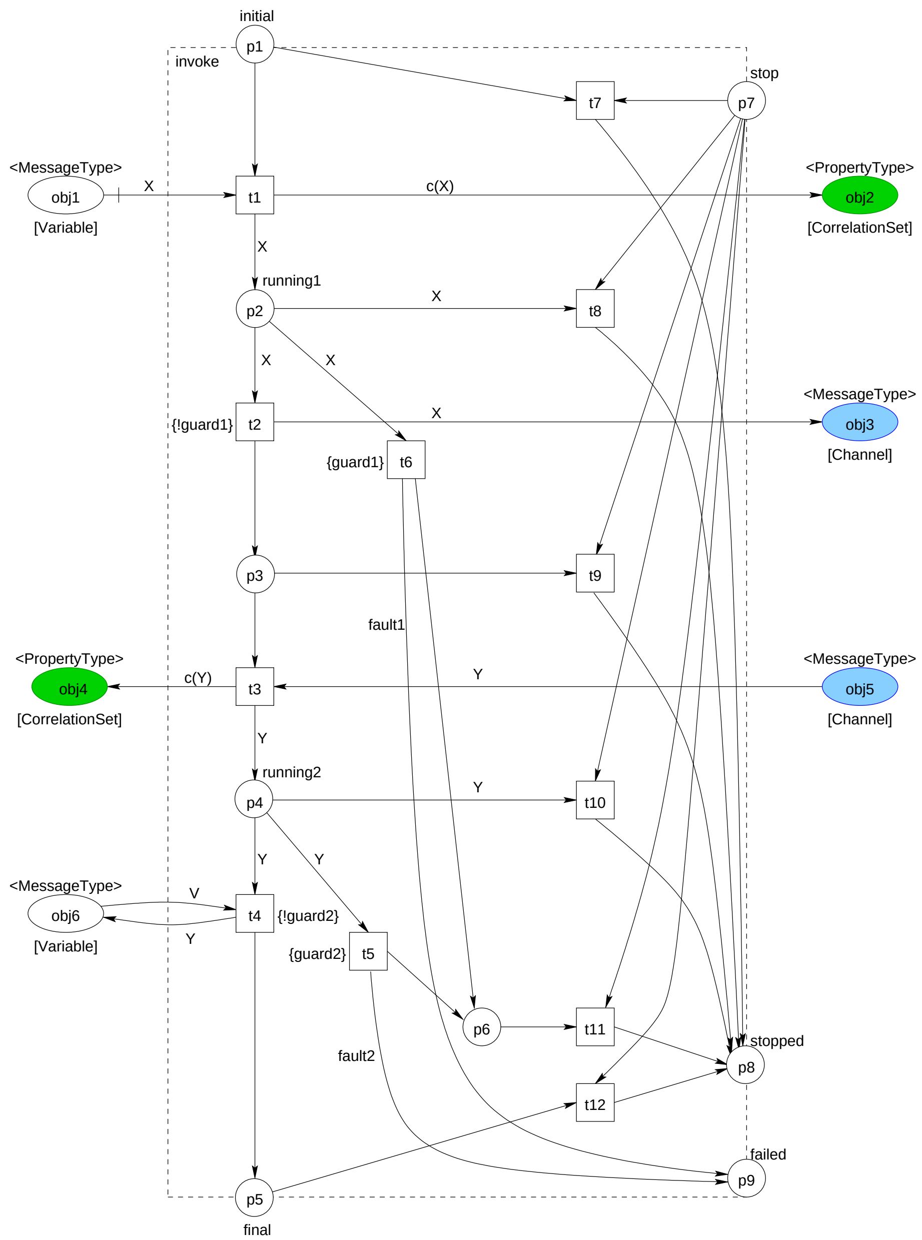


Figure 13: Pattern for BPEL's synchronous `invoke` in case of initiate="yes".

The case that `correlation sets` have to be set, i.e. initiate="yes" is depicted in Fig. 13. In this model we assume that two `correlation sets` are defined (one for sending a message and one for receiving a message) and both have to be initialized.

## 4.7 Throw

If a business process should generate an internal fault, the activity `throw` can be used. The respective pattern is depicted in Fig. 14. It puts the name of the fault into place failed by firing `t1`. In this pattern there is no need for a place final, because after signalling the fault, the whole `scope` is finished by the stop pattern. For this reason, the control flow neither needs to be passed to the successor activity nor to the enclosing `scope`. The pattern's stop component is established by transitions `t2` and `t3`.

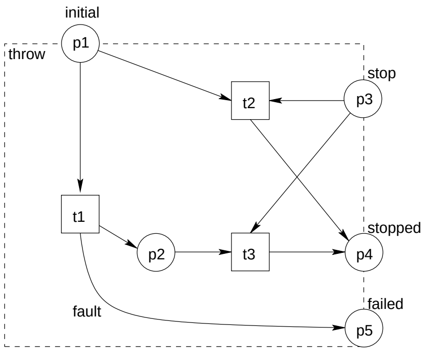


Figure 14: Pattern for BPEL's `throw`.

## 4.8 Terminate

In this section the pattern of BPEL's `terminate` is presented (see Fig. 15). The activity `terminate` is executed to terminate the whole `process` instance. In our model, the `process` is transferred to the state Terminated and the stop pattern is used for execut-

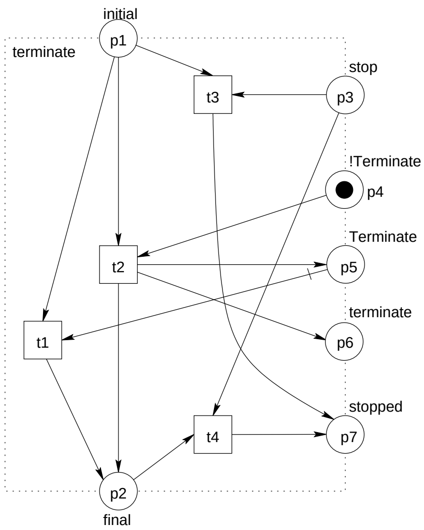


Figure 15: Pattern for BPEL's `terminate`.

ing the termination of the `process`. Two scenarios are possible: Either the `process` is already in state Terminated, i.e. another `terminate` activity has been executed before, and nothing has to be done (t1) or the termination of the `process` is started by transferring the `process` to state Terminated (t2). The pattern's stop component is made up by transitions t3 and t4.

In contrast to the throw pattern, we do need a place final in this pattern for the activity `terminate`. Consider the activity `terminate`, being embedded in a `fault handler`, is executed. An active `fault handler` has to be fully executed in order to guarantee firstly, the removing of the tokens and secondly, to ensure finishing of the patterns. After having executed the `fault handler`, the termination process gets control of stopping the `process`. Otherwise, if the activity `throw` is executed inside of a `fault handler`, the generated fault stops the respective `fault handler`. Afterwards the fault is rethrown to the immediately enclosing `scope`. Furthermore t1 is also needed, if two `terminate` activities are executed in a `fault handler` concurrently. Otherwise, our model would deadlock.

# 5 Transformation of BPEL's Structured Activities

In the following we transform BPEL's structured activities. A structured activity embeds at least one activity. Such an embedded activity (we will also call it an *inner activity*) can be any BPEL activity. But how should such an inner activity be visualized in our patterns?

In our model it is most important to see how the interfaces of an inner activity and its enclosing activity are joined. Thus, only the interface of an embedded pattern is visualized and all other information of the pattern is hidden. Therefore only the frame and places *initial*, *final*, *stop*, *stopped* and if needed *negLink* are visible. This interface concept allows plugging patterns together as it is done in BPEL. This notation is a simplification and a generalization of an inner activity. It means, the inner activity can be any BPEL construct, even a `throw` (although the pattern of BPEL's `throw` has no final place).

*negLink* is an abbreviation of negative link. It is an optional place that is only part of a link pattern's interface or of a structured activity pattern's interface when it embeds at least one activity that is source of a `link`. With the help of *negLink* the status of all source `links` of an inner activity that are not executed anymore is set to negative. Consider, for instance, an activity within a branch that is not taken in a `switch` activity. In other words, *negLink* is a place for modelling *dead-path-elimination* [LR99]. If *negLink* is depicted in a pattern, we suppose that the respective activity contains at least one activity that is source of a `link`.

The patterns of the structured activities become larger than most of the basic activity's pattern. Thus, we extend our graphical notation in order to simplify the respective Petri nets: Two places with the same identifier are joined.

## 5.1 Sequence

BPEL's `sequence` "contains one or more activities that are performed sequentially" [CGK+03, p. 59]. The general pattern of a `sequence`, which can carry a number of $n$ inner activities is depicted in Fig. 16.

As already mentioned, places with the same identifiers are joined, e.g. initial of the `sequence` and initial of innerActivity1 (p2), final of innerActivity1 (p4) and initial of innerActivity2 (p4) and so on.

There are two possible interleavings, because either initial is marked or negLink: Either innerActivity1, ..., innerActivityn are executed sequentially or the status of all source `links` embedded in the `sequence` is set to negative (t1).

If there is a token on stop, the `sequence` and its embedded activities will be stopped. The places stop and stopped are joined with all stop places and stopped places of the inner activities.

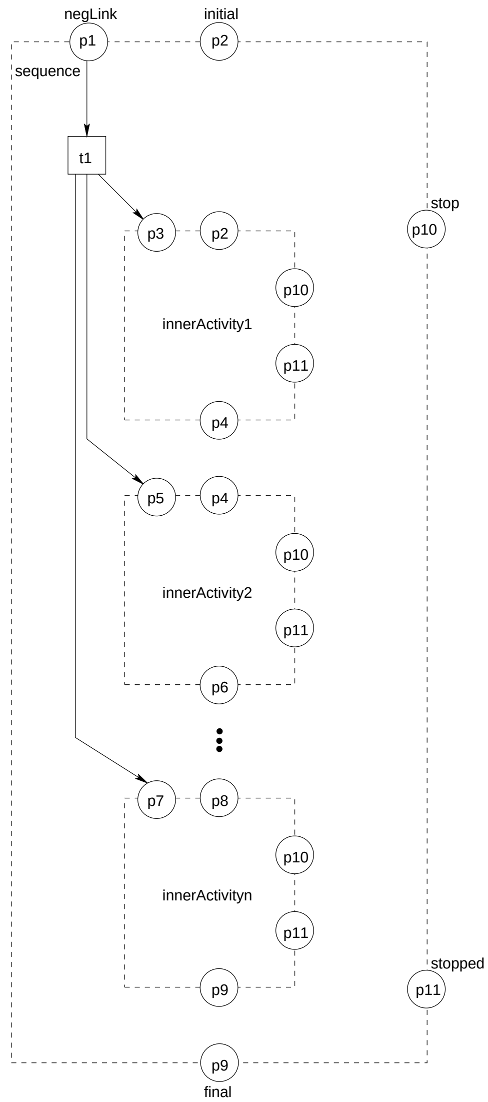


Figure 16: Pattern for BPEL's `sequence` embeds $n$ activities.

## 5.2 Flow

A `flow` provides an environment for performing one or more activities concurrently. After each of them has been finished, they are synchronized. The general pattern of BPEL's `flow`, which can carry a number of $n$ inner activities is shown in Fig. 17.

Again, either `initial` or `negLink` is marked. So we have to look at two possible scenarios: Either all inner activities are executed concurrently (`t2`) and afterwards they are synchronized (`t3`) or the status of all source `links` embedded in the `flow` is set to negative (`t1`). The pattern's stop component is established by transitions `t4` – `t7`.

If there is a token on `stop`, the `flow` and its embedded activities will be stopped. After

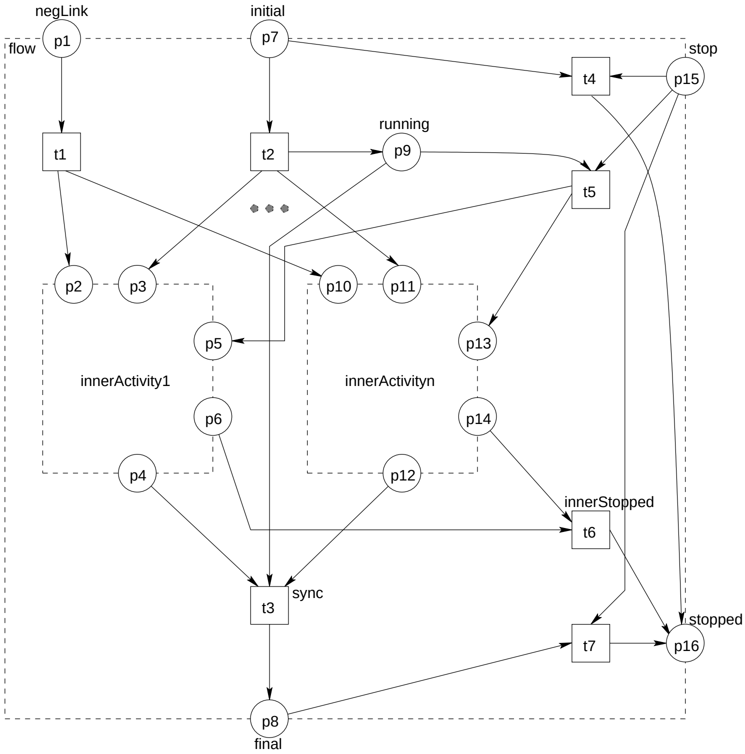


Figure 17: Pattern for BPEL's `flow` embeds $n$ activities.

t5 has fired the token lying on running is consumed; thus t3 cannot be activated. Furthermore the stop place of each inner activity is marked. So innerActivity1, ..., innerActivityn can be stopped concurrently. Firing t6 synchronizes them.

## 5.3 While

BPEL's `while` "supports iterative performance of a specified iterative activity. The iterative activity is performed until the given boolean while condition no longer holds true" [CGK+03, p. 60]. No `link` must cross the boundary of a `while` activity; thus the respective pattern visualized in Fig. 18 does not contain a place negLink.

Firstly, the data being stored in a `variable` is read (t2).[4] Afterwards there are three possible scenarios depending on the evaluation of the guards guard (describes a fault case) and cond (describes the loop condition): Either a fault occurs, e.g. because of a selection failure (t5) or the loop condition does not hold (t1). In the third scenario the loop condition holds (t3). Therefore innerActivity is performed and the loop is repeated (t4). The pattern's stop component is established by transitions t6 – t9.

[4] Reading a message holt in a variable is also possible, but not shown. Then the `variable` is of sort <MessageType>.

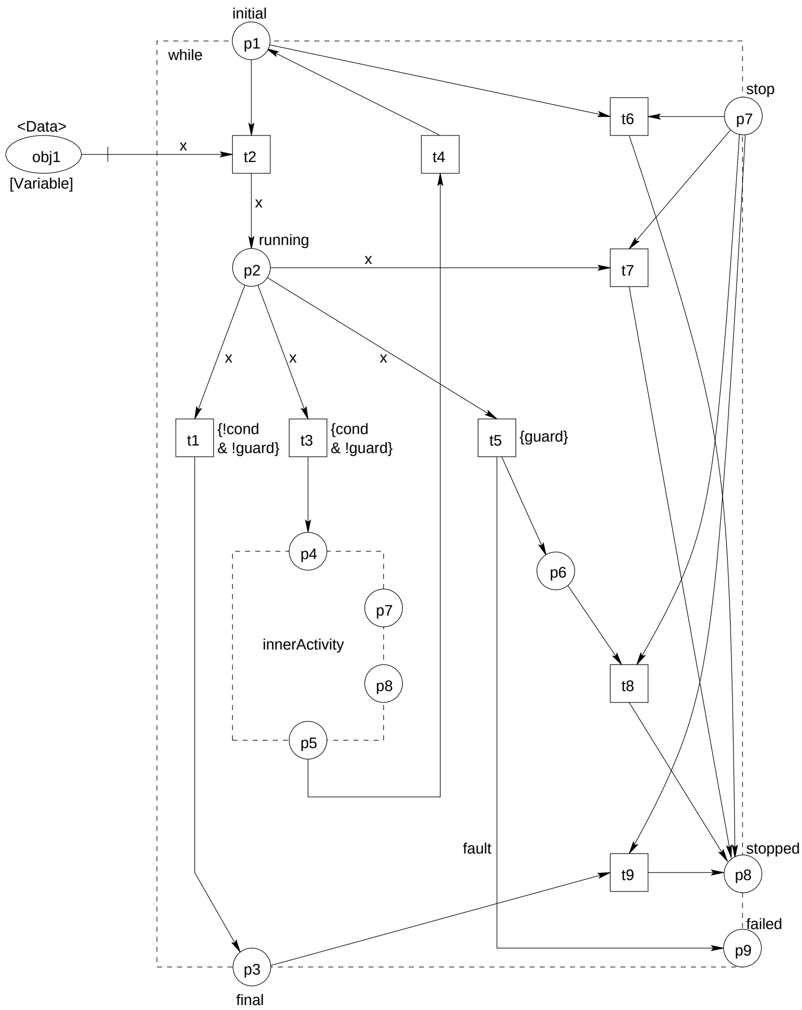


Figure 18: Pattern for BPEL's `while`.

## 5.4 Switch

BPEL's `switch` "...consists of an ordered list of one or more conditional branches defined by case elements, followed optionally by an otherwise branch. The case branches of the switch are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the otherwise branch is taken" [CGK+03, p. 59]. In Fig. 19 the pattern of a `switch` that consists of two case branches is shown.

Two scenarios are possible, because either place initial or place negLink is marked. If initial is marked (scenario 1), data being stored in a `variable` is read (t3).[5] Afterwards there are three possible sub-scenarios depending on the evaluation of guards guard (describes a fault case) and cond (describes the condition of the respective case branch).

[5]Reading a message holt in a `variable` is also possible, but not shown.

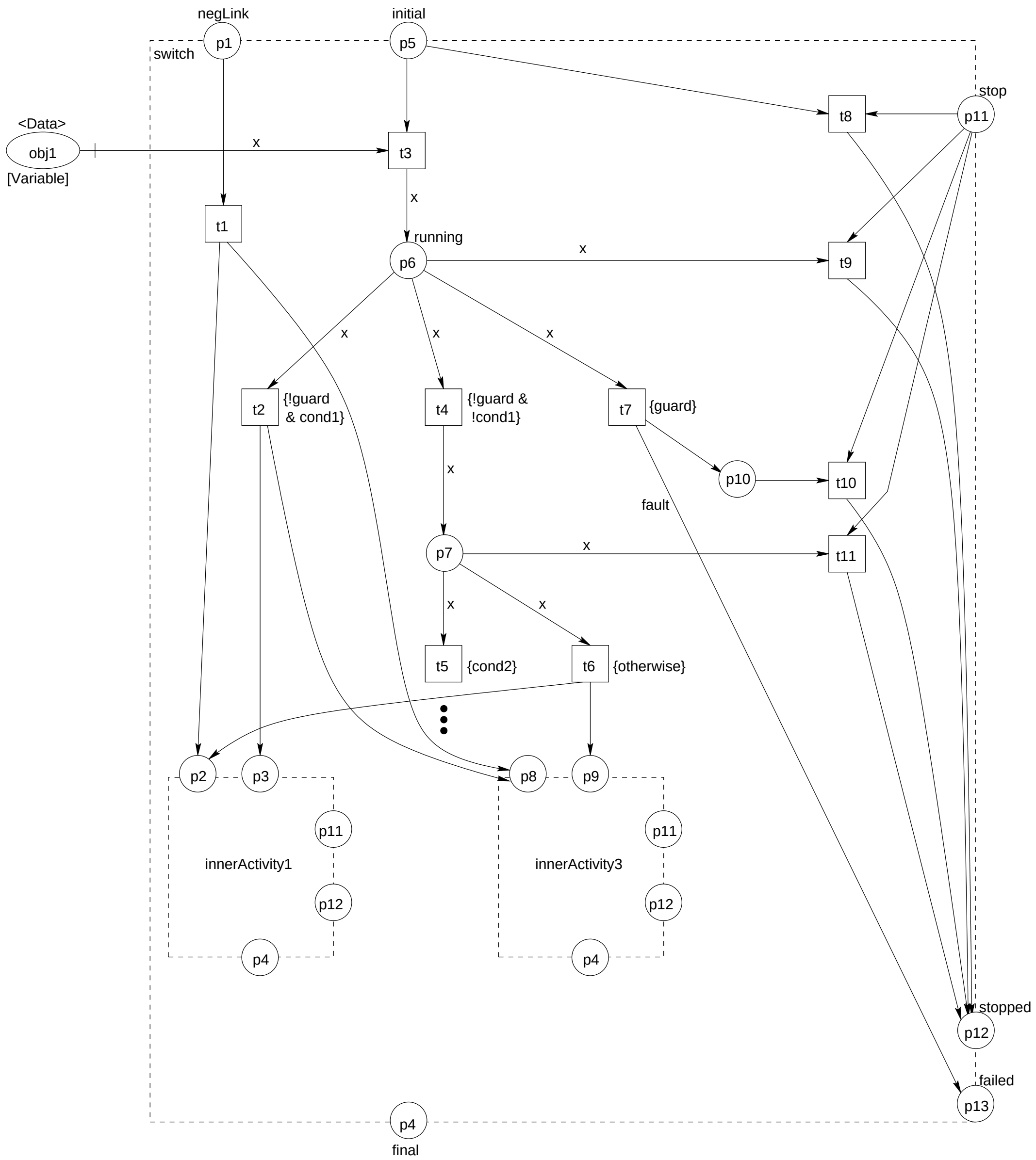


Figure 19: Pattern for BPEL's `switch` and two case branches.

Either a fault occurs (t7), e.g. because of a selection failure, or the condition of the first case branch is evaluated. If the condition holds (i.e. t2 fires), innerActivity1 is performed and the status of all source `links` embedded in the other inner activities that are not performed anymore are set to negative, e.g. a token on place p8. Otherwise (t4) the condition of the second case branch is evaluated. If none of the case branch conditions hold, the otherwise branch is performed (t6). Again, on every negLink place of all other inner activities a token is produced. If negLink is marked (scenario 2), the status of all source links embedded in the switch activity are set to negative (t1). The pattern's stop component is established by transitions t8 – t11.

## 5.5 Pick

A pick either waits on a message event or a timing event. This activity defines one or

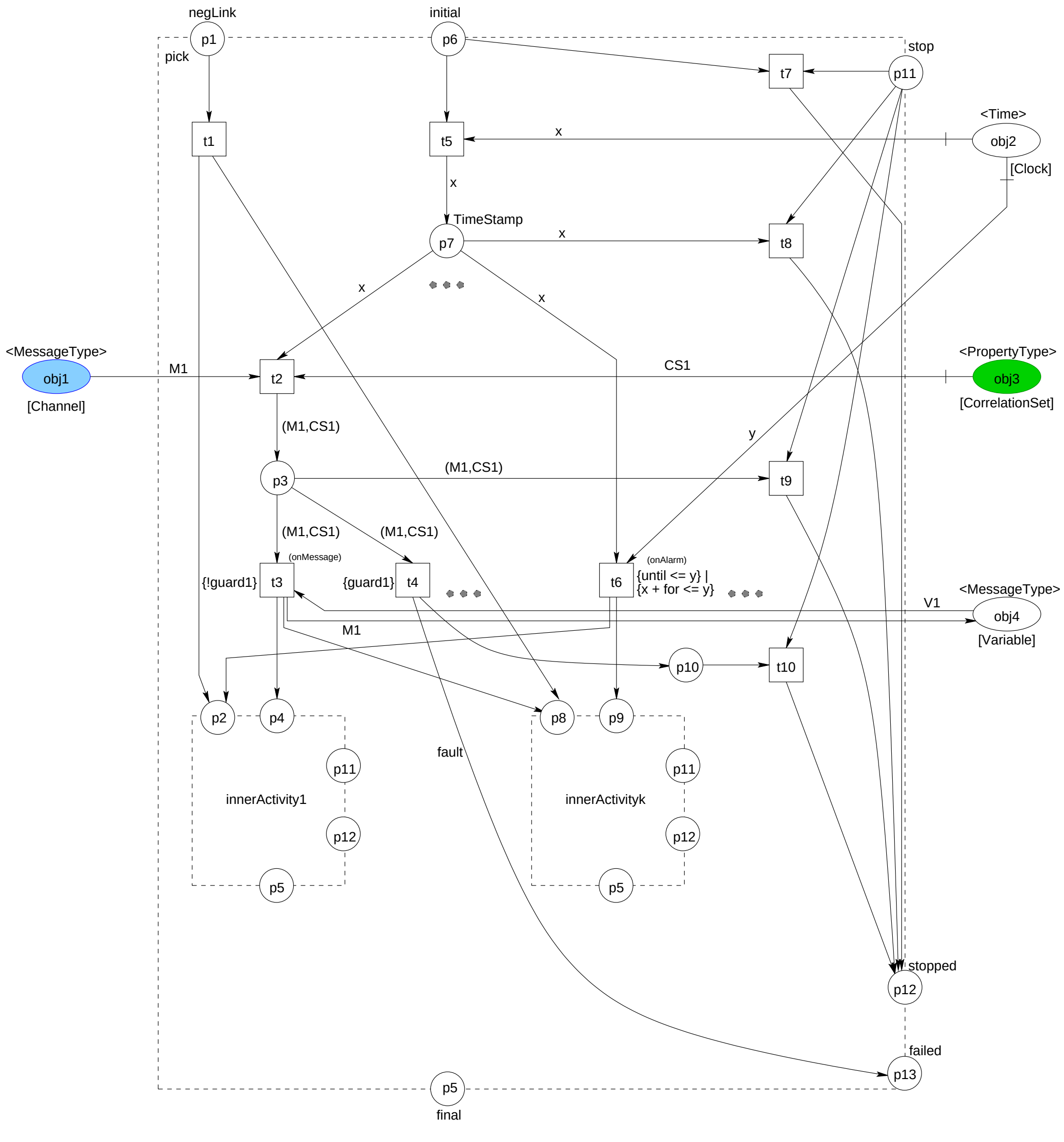


Figure 20: Pattern for BPEL's pick in case of initiate="no" that contains $k-1$ onMessage branches.

more so-called onMessage branches and any number of so-called onAlarm branches. As known from activity wait an alarm event occurs when either a delay for a certain period of time is reached or until a certain deadline is reached. Each onMessage branch also specifies an attribute initiate. So we have built two patterns. The general pattern of BPEL's `pick` defining $k-1$ onMessage branches and any number of onAlarm branches in case of initiate="no" is visualized in Fig. 20.

Depending on the initial marking (either initial or negLink are marked) two scenarios are possible. In the first scenario initial is marked and a time stamp is read (t5). Then either a message (t2) or an alarm event occurs (t6). If the transition guard, specifying the occurrence of a correlation violation or some other fault holds, the fault is thrown (t4). Otherwise the message is written in the `variable`, innerActivity1 is performed (t3) and the status of all source `links` embedded in all other inner activities not performed anymore is set to negative, e.g. a token on place p8. In the case that an alarm has occurred, innerActivityk is performed and the status of all source `links` embedded in all other inner activities not performed anymore is set to negative, e.g. a token on place p2. In the second scenario, negLink is marked, and the values of all source `links` embedded in the `pick` activity are set to negative (t1). The pattern's stop component is established by transitions t7 – t10.

The equivalent pattern in the case of (initiate="yes") is depicted in Fig. 21.

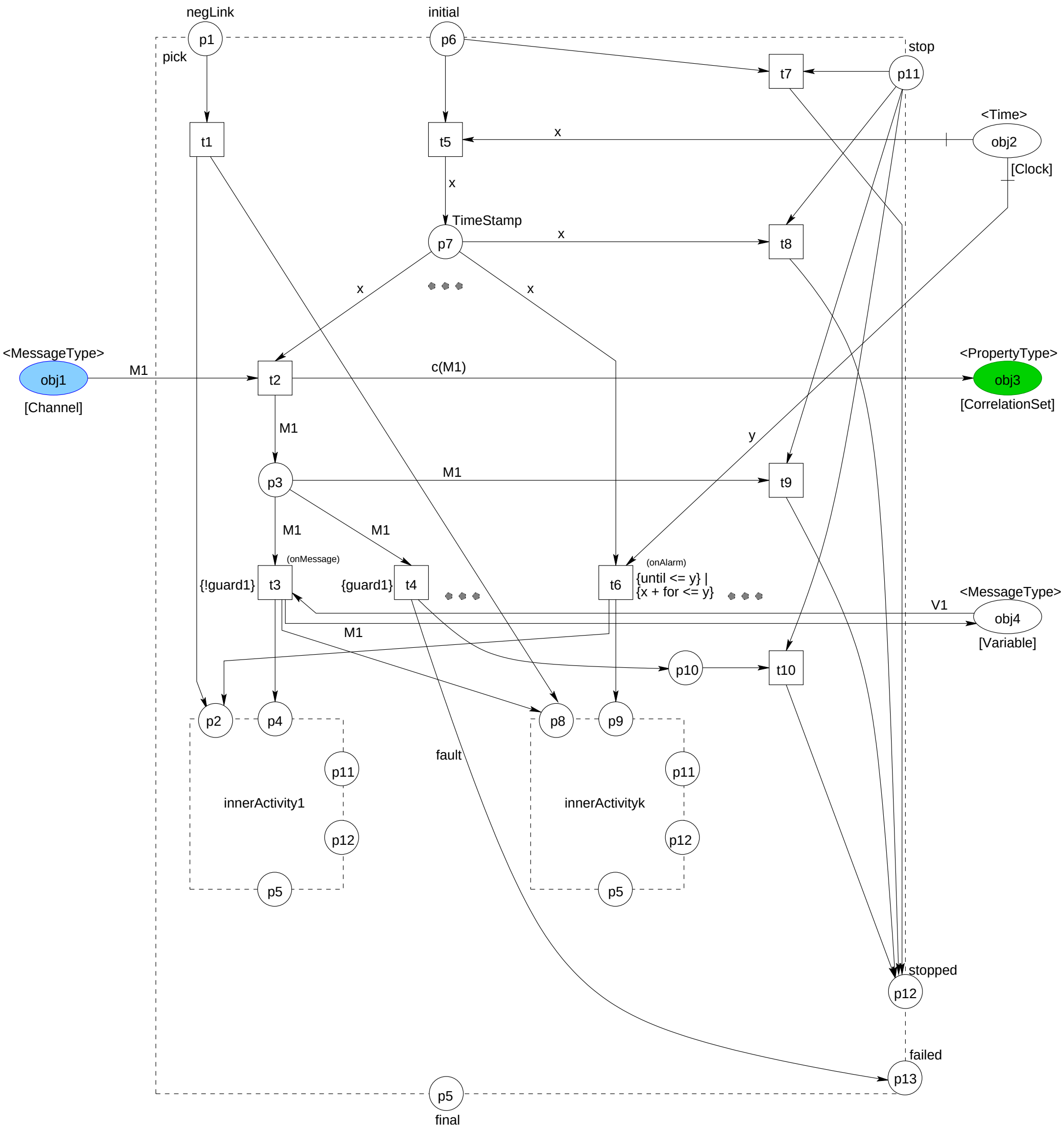


Figure 21: Pattern for BPEL's `pick` in case of initiate="yes" that contains $k-1$ onMessage branches.

# 6 Transformation of BPEL's Link Semantic

For synchronizing subtasks of a flow, BPEL provides a construct of the so-called links. Each link specifies two elements: a *source* and a *target* activity. Whereas a source activity may specify a *transition condition* a target activity may specify a *join condition*[6]. Furthermore, a link depends on the value of the attribute suppressJoinFailure which is either "yes" or "no". If the join condition does not hold, BPEL's standard fault joinFailure occurs. The fault is either suppressed, i.e. the target activity is not executed anymore and the values of all outgoing links (i.e. links for that the activity is the source) embedded in the target activity are set to negative. Otherwise, the fault is not suppressed, that means, it is thrown to the stop pattern of the surrounding scope.

A link is only set once and it must be possible to check whether it is set or not. Thus, we decided to model a link by two complementary places – !outLink1 and outLink1 in Fig. 22, for instance. !outLink1 is a low-level place marked at the beginning of the process pattern whereas outLink1 is of the sort boolean. Note, link places are outside the frame of the respective pattern, because they synchronize two activities. A link place is no object.

We again extend our graphical notation in order to simplify the Petri nets: A place depicted with a dotted line, e.g. p10 in Fig. 22 is a replication of a place depicted with a solid line and the same identifier. This notation is used to avoid crossing edges in the net.

## 6.1 Source Activity

In Fig. 22 the source link pattern is visualized. It does not depend on the value of the attribute suppressJoinFailure. We suppose that X is a pattern of a structured activity that embeds at least one activity that is source of a link.

There are two possible scenarios: Either initial or negLink is marked. In the case of the positive control flow, there is a token on initial; thus, the embedded activity X is activated. After X is executed faultlessly, place p4 is marked and either t2 fires or t3. To set the links means firing t2. Variables transCond1 and transCond2 get the value of the given transition condition that is produced on p10 and p11. Otherwise, a fault has occurred and the values of all links were set to negative by the surrounding scope. Then t3 fires.

If negLink is marked, X is an activity within a branch which is not executed anymore. This is the case of dead-path-elimination where the values of all source links have to be set to negative. By firing t1, outLink1 and outLink2 are set to "false". Furthermore a token is produced on p3, because the interface of activity X contains a place negLink.

[6]Both conditions are boolean expressions.

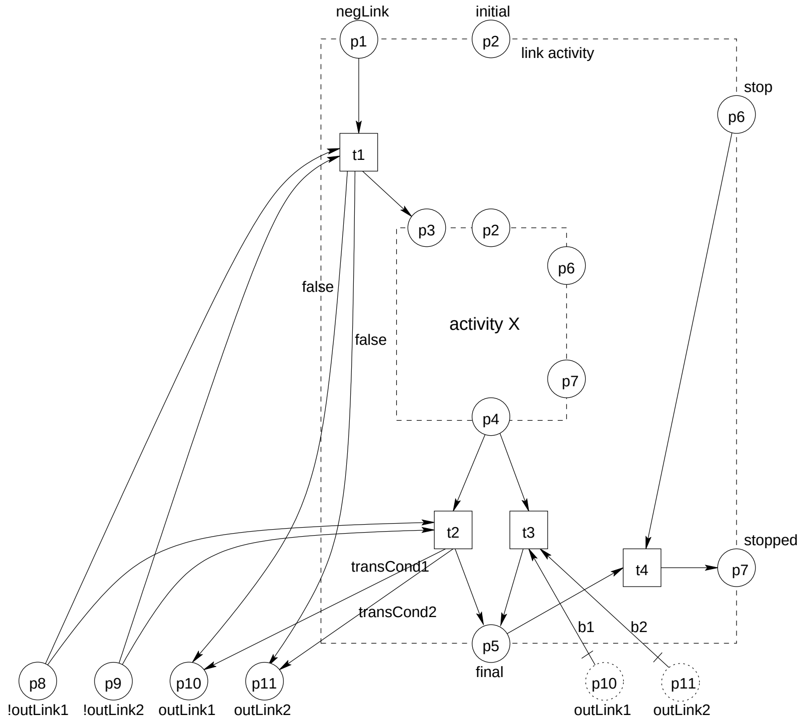


Figure 22: Pattern for an activity that is source of two `links`.

## 6.2 Target Activity

The pattern of the target activity which shows `suppressJoinFailure` = "no" is depicted in Fig. 23. We suppose that X is a pattern of a structured activity that embeds at least one activity that is source of a `link`.

Depending on whether initial or negLink is marked or not there are two possible scenarios. If initial is marked, there are two interleavings: The join condition – modelled as the guard joinCond – either holds and activity X is executed (t1) or joinCond does not hold (t2) and the fault `joinFailure` – saved in variable fault – is thrown. Otherwise, if a token is on negLink, X is not executed and the values of all source `links` being embedded in X are set to negative.

Each link place is joined with the respective link place of the source pattern in Fig. 22: !inLink1 with !outLink1, !inLink2 with !outLink2, inLink1 with outLink1 and inLink2 with outLink2.

In contrast, Fig. 24 depicts the same pattern in the case of `suppressJoinFailure` = "yes". If the join condition does not hold (t2), the target activity is not executed and the control flow is continued after this activity. Furthermore, if there are source links embedded in X (as shown in Fig. 24), their values are all set to negative by producing a token on p5.

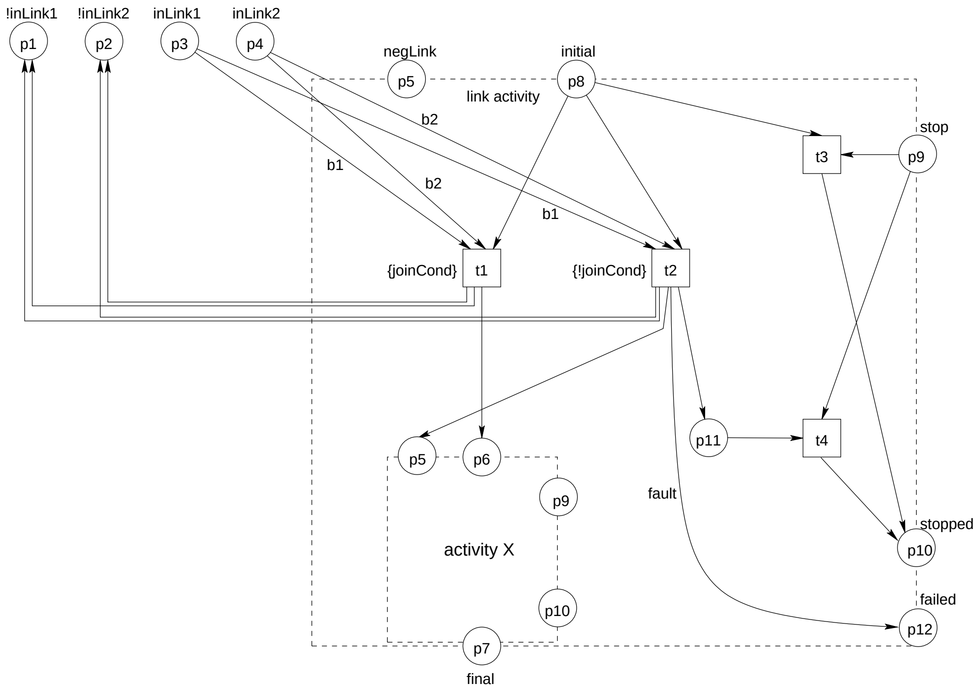


Figure 23: Pattern for an activity that is target of two `links` in case of `suppressJoinFailure` =“no”.

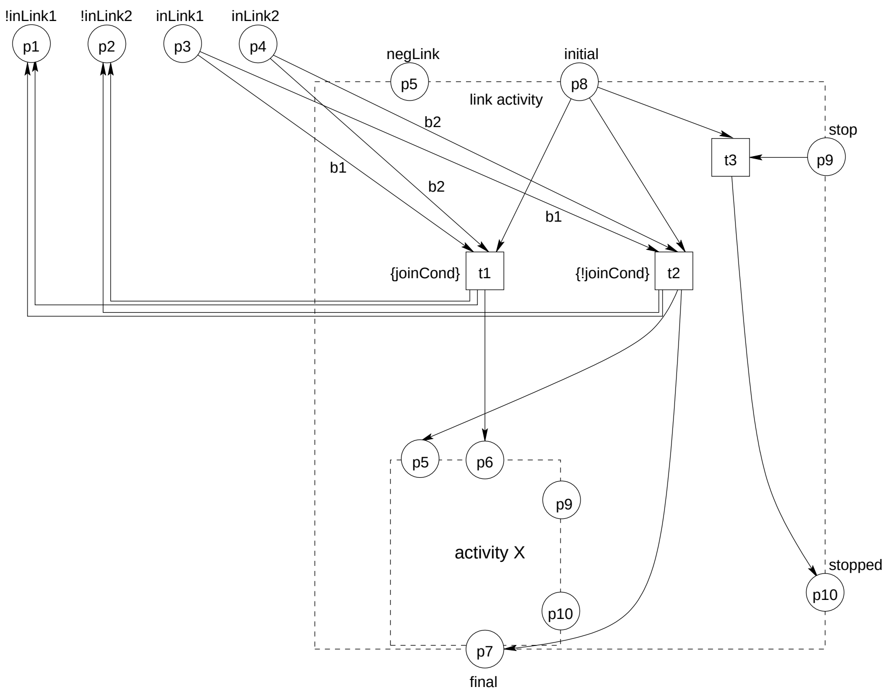


Figure 24: Pattern for an activity that is target of two `links` in case of `suppressJoinFailure` =“yes”.

### 6.3 Activity with Source and Target Element

Now we present the pattern for an activity that is source and target of two `links`. We again need two patterns: Fig. 25 shows `suppressJoinFailure` =“yes” and Fig. 26 shows `suppressJoinFailure` =“no”.

Figure 25 results from the patterns visualized in Figures 24 and 22. There are only four new arcs: p5t5, p6t5, t5p7, and t5p8. The two possible scenarios are similar to the patterns presented in the subsections before. Either there is a token on negLink or initial is marked.

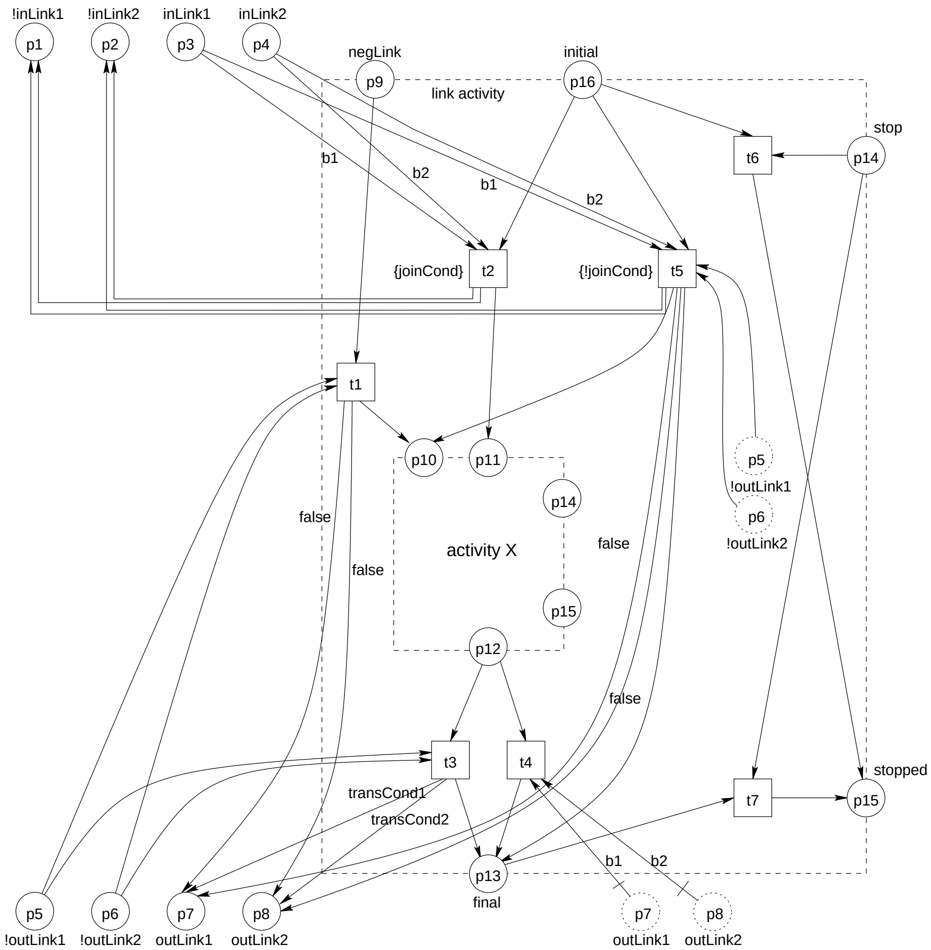


Figure 25: Pattern for an activity that is source and target of two `links` in case of `suppressJoinFailure` =“yes”.

The case of `suppressJoinFailure` =“no” depicted in Fig. 26 results from Figures 23 and 22 and is as expected.

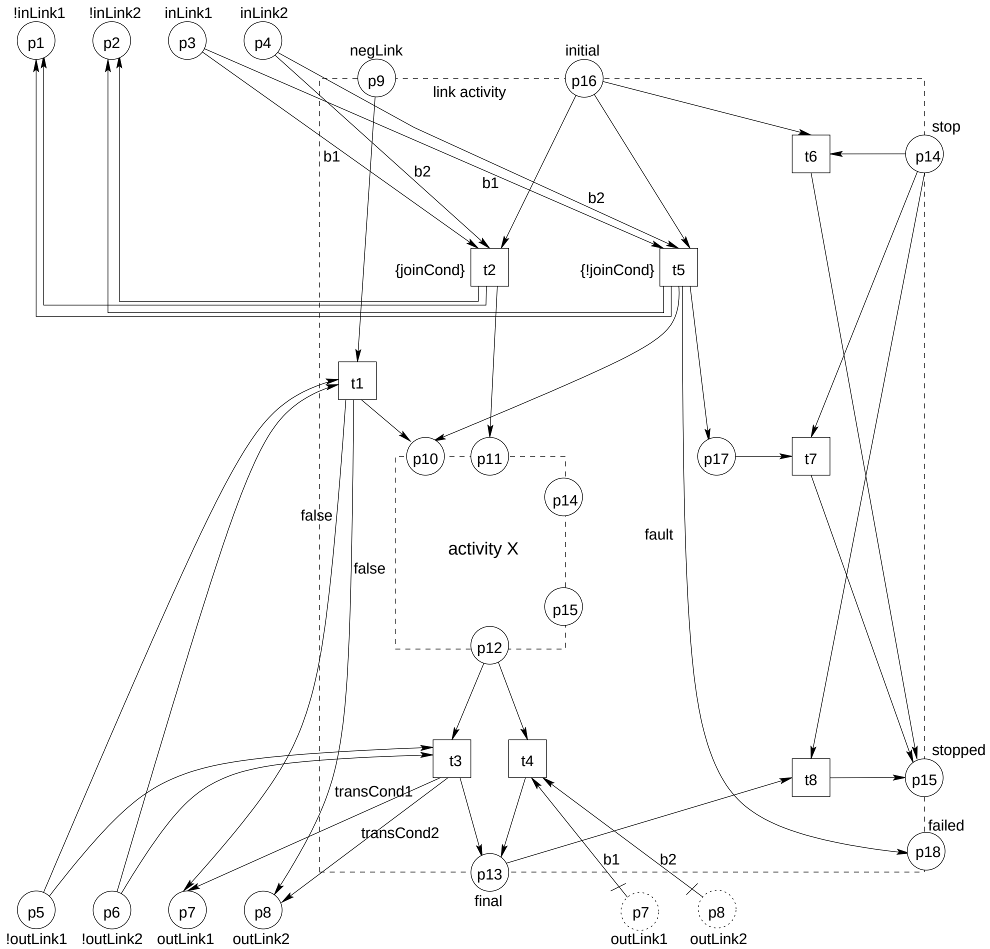


Figure 26: Pattern for an activity that is source and target of two `links` in case of `suppressJoinFailure` =“no”.

## 6.4 Summary Dead-Path-Elimination

Let us summarize the modelling of dead-path-elimination. In general, DPE is realized by the place negLink which is used in the patterns of BPEL's `scope`, source, target, and structured activities. In the following we explain what happens, if the negLink place of the respective pattern is marked:

In the **source activity** the values of all target `links` are set to negative. If the source activity embeds an activity that is source of a `link`, the status of that `link` is set to negative, too.

If the **target activity** embeds an activity that is source of a `link`, the status of that `link` is set to negative.

The negLink places of all patterns enclosed in a **sequence** or **flow** are marked, if the respective pattern is source of a `link`.

The patterns of **pick** and **switch** behave just like the `flow` and `sequence`. But in the case of the positive control flow, the negLink place of every pattern in a branch, which is not performed anymore, is marked.

In a **scope** pattern the negLink place of the embedded activity is marked, if it is source of an activity.

There is no negLink place in the pattern of BPEL's **process**, because a `process` is not enclosed by another activity.

The pattern of BPEL's **while** does not have a place negLink as well, because no `link` is allowed to cross this activity.

Last but not least the patterns of BPEL's **elementary activities** do not have a place negLink, because there is no activity which can be enclosed in an elementary activity.

# 7 Transformation of BPEL's Scope

In this section we want to transform BPEL's `scope` which is also a structured activity. A `scope` encloses one activity linked to the transaction management. It is a wrapper for an activity, a `fault handler` and a `compensation handler` whereas both handlers are defined at least by default. Optionally, a `scope` may enclose an `event handler`.

In order to transform the `scope` we have to link its elements. In addition, a stop pattern is used in our model: If a `scope` has to be stopped, the stop pattern controls this procedure. Furthermore, we use *state places* and a subnet which sets the status of all `links` in the `scope` after the `fault handler` has finished. The scope pattern also encloses `variables` and `correlation sets`.

## 7.1 State Places

Now we give an outline of the *states* a `scope` can be in. Most of the states are based on the Business Agreement Protocol's state diagram [CGK+03, pp. 116]. Modelling the transaction management it is important to know which event has taken place. For instance, whether a fault has occurred or a `scope` has been compensated. Such a state is modelled by a *state place*. In more detail, if a state place is marked, the `scope` is in that respective state. For every state there exists a complement state modelled by a complement place. Except for Terminate and !Terminate defined globally in the process pattern, every `scope` has the following state places:

**(!)Active:** *As long as positive control flows in a scope, this scope is in state Active.*

State Active is reached as soon as the `scope`'s inner activity is activated. After this inner activity and the `event handler` have finished, the `scope` changes into state !Active. The token on place Active is also consumed by the stop pattern, if the `scope`'s stop place is marked. That means, either an activity `terminate` is activated, a `forcedTermination` is signalled (i.e. a `scope` is forced to stop) or a fault has occurred.

**(!)Completed:** *A scope changes into state Completed, after its inner activity has been executed faultlessly and its event handlers have been finished.*

When its inner activity is activated, the `scope` reaches state !Completed. After the `scope` has completed faultlessly, it changes into state Completed. This state is queried inside the `scope`'s `compensation handler`, because a `scope` is compensated only when it has completed faultlessly. A `scope` in state Completed cannot leave this state.

**(!)Compensated:** *A scope changes into state Compensated, if its compensation handler is activated for the first time.*

By activating its inner activity the `scope` reaches state !Compensated which is left only when the `compensation handler` is executed. A `scope` can be compensated only once. So this state is also queried in the `compensation handler` of the `scope`. A `scope` cannot leave the state Compensated.

**(!)Ended:** *A scope changes into state Ended if it has finished faulty, i.e. it was stopped, and control does not flow after the scope, but in the hierarchy of its enclosing scope.*

With the activation of its inner activity, the `scope` changes into state !Ended. This state can only be left, if control flow is inside the stop pattern or inside the fault handler pattern. In the stop pattern state Ended is reached either if the activation of an activity `terminate` is signalled and this signal stops the `scope` or if a fault occurs while the `compensation handler` either is executed or was executed. In the `fault handler` pattern the state is changed into Ended when the `scope` rethrows a fault to the immediately enclosing `scope`. Note, if a user defined `fault handler` has caught a fault (i.e. the fault is handled), the `scope` stays in state !Ended. Furthermore state Ended is used for removing tokens in the fault handler pattern and in the stop pattern. Such a token symbolizes a fault that cannot be handled anymore. A `scope` cannot leave state Ended.

**(!)Faulted:** *If a scope was stopped by a signalled forcedTermination or a fault, it is in state Faulted.*

This state is needed to distinguish between the occurrence of a fault during the fault handling or during the positive control flow. With the activation of its inner activity the `scope` reaches state !Faulted. When a fault or `forcedTermination` is signalled, the stop pattern stops the `scope` and all signalled faults are removed. Then the `scope` changes into state Faulted. Afterwards the `fault handler` is activated. Therefore all subsequently signalled faults (i.e. all faults that occur, if the state is already in state Faulted) can only occur while the `fault handler` is being executed. A `scope` cannot leave state Faulted.

**(!)Terminated:** *The process and with it every scope changes into state Terminated, as soon as the first terminate activity is activated.*

As mentioned before, both places are defined globally in the process pattern and every `scope` has access to them. If the process pattern is activated, it reaches state !Terminated. It does not change this state until the first `terminate` activity is activated. Being in state Terminate, a process pattern cannot leave this state.

## 7.2 Scope

As explained before, a `scope` is a wrapper for an activity, a `fault handler`, a `compensation handler` and optionally an alarm `event handler` and a message `event handler`. Figure 27 depicts the pattern of a `scope` enclosing all these elements. It is also source of a `link` scopeLink and embeds an activity which is source of the `links` sourceLink1 and sourceLink2.

The idea of our pattern is as follows: Firstly, the `scope` activates its inner activity, the `event handlers` and sets its state places (t2). After the `scope`'s inner activity is finished, the `event handlers` (t3) and afterwards the `scope` itself is finished (t4). In addition, the `scope`'s name, A, (saved in variable scopeName) is saved in the `compensation handler` of the immediately enclosing `scope` (token on place push_A). This is possible, because starting the process pattern, on all places !push_scopeName a token is produced (see transition t2 in Fig. 28 in Sec. 7.3). More details to the concept of push places can be found in Sec. 10.1. This behaviour is called positive control flow of the `scope`. The stop pattern gets the control of the `scope` in case a fault occurs. The `fault handler` is started by the stop pattern whereas the `compensation handler` is invoked by activity

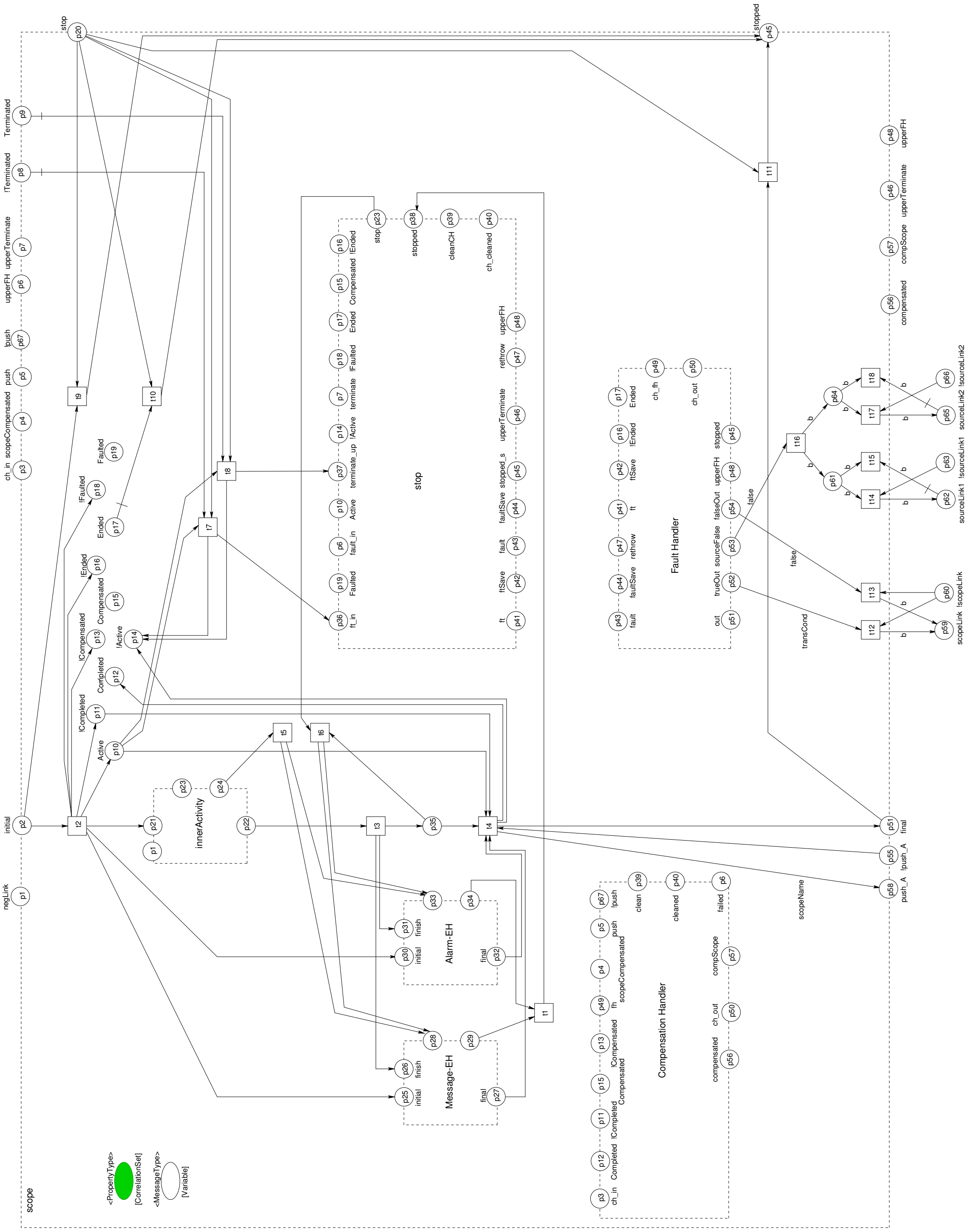


Figure 27: Pattern of BPEL's scope.

compensate which is either embedded in the fault handler or in the compensation handler of the immediately enclosing scope. Furthermore, the scope is influenced by the immediately enclosing scope as well as by scopes it encloses. Thus, it has an interface for sending and receiving signals.

Looking at the interface, the places initial, final, stop, stopped, and negLink of Fig. 27 are known. The places ch_in, scopeCompensated, and push are joined with the respective places in the compensation handler. The places upperFH and upperTerminate are joined with the places fault_in and terminate in the stop pattern. As mentioned before, the places Terminated and !Terminated are defined in the process pattern. The places completed and compScope as well as upperTerminate and upperFH are joined with the respective places in the compensation handler and the fault handler. The places push_A and !push_A are joined with the respective places in the compensation handler of the immediately enclosing scope. The variable and the correlation set on the left in Fig. 27 visualize objects defined in the scope. They are visible in the scope only.

There are two possible interleavings: Either the status of all source links embedded in the scope is set to negative, i.e. negLink is marked or positive control flows, i.e. initial is marked.

The pattern's stop component is established by transitions t7 – t11. Let stop be marked. If the scope pattern is already stopped, it is in state Ended. Then t10 fires. Otherwise, the scope pattern is in state Active. The scope is either in state !Terminated or Terminated. Thus, either t7 is enabled (i.e. the immediately enclosing scope signals a forcedTermination) or t8 is enabled (i.e. the immediately enclosing scope signals the activation of an activity terminate). In contrast, place upperFH is marked if a fault is signalled which has occurred in an enclosed scope. There is a token on place upperTerminate, if the activation of an activity terminate, occurred in an enclosed scope, is signalled.

If the stop pattern starts the stop procedure, place p23 is marked and two scenarios are possible depending on whether the innerActivity is still executed (scenario 1) or not (scenario 2): In scenario 1 the innerActivity is stopped, because of joining the stop places. After stopping the inner activity, place p24 is marked. Firing t5 stops the event handlers and afterwards the stop pattern continues its execution (t1). In scenario 2 the innerActivity is already finished and has signalled a finish to the event handlers (t3). If place stop is marked, the scope pattern is in state !Active; thus, t4 cannot be enabled anymore. Then, the scope pattern is stopped by firing t6 and t1.

Let us take a look on how the fault handler is linked to the scope's structure, especially how links are set. Place out is marked, if the fault handler has caught a fault and the processing continues after the scope. With the help of the places trueOut and falseOut, respectively the status of links, e.g. scopeLink, whose source activity is the scope, is set to the value of the transition condition (t12) or to negative (t13). Furthermore, the status of all links whose source activity is enclosed in the scope and that could not be executed because of a fault have to be set to negative. This happens, if place sourceFalse is marked in our model. Then the status of all links is checked. We explain the three possible scenarios by considering sourceLink1 as an example: Firstly, consider this link not to be set by its source. So !sourceLink1 is marked and the status is set to negative by firing t14. Secondly, consider this link to be already set and the join condition of the respective target activity is evaluated, too. Again, !sourceLink1 is marked and t14 fires. However, the token on sourceLink1 does not influence the control flow anymore. In the third scenario there is a token on sourceLink1, i.e. either the link was set by its source activity or the second scenario was executed in an enclosed scope. So the link is not set again, but t15 fires to consume the token.

## 7.3 Process

BPEL's process is a special case of the scope. In more detail, it is the outmost scope of the BPEL process. Figure 28 depicts the pattern of a process, which embeds the similar elements as the scope in Fig. 27.

In fact, both patterns look very similar, but the process pattern is less complex. On the one hand it has a smaller interface, because the process has no enclosing scope. On the other hand links whose source activity is embedded in the process and that could not be executed because of a fault, do not have to be set, because the process will be stopped.

The interface is a subset of the scope's interface. On the left there are variables and correlation sets defined and therefore visible inside the process. Furthermore, all channels and the clock are defined in the process. There is only one possible scenario, i.e. initial is marked. Firing t2 initializes all source links as well as state places, the inner activity, the event handlers, and for each scope embedded in the process its respective place !push (we consider scope A to be the only scope embedded in the process, so the !push place is !push_A).

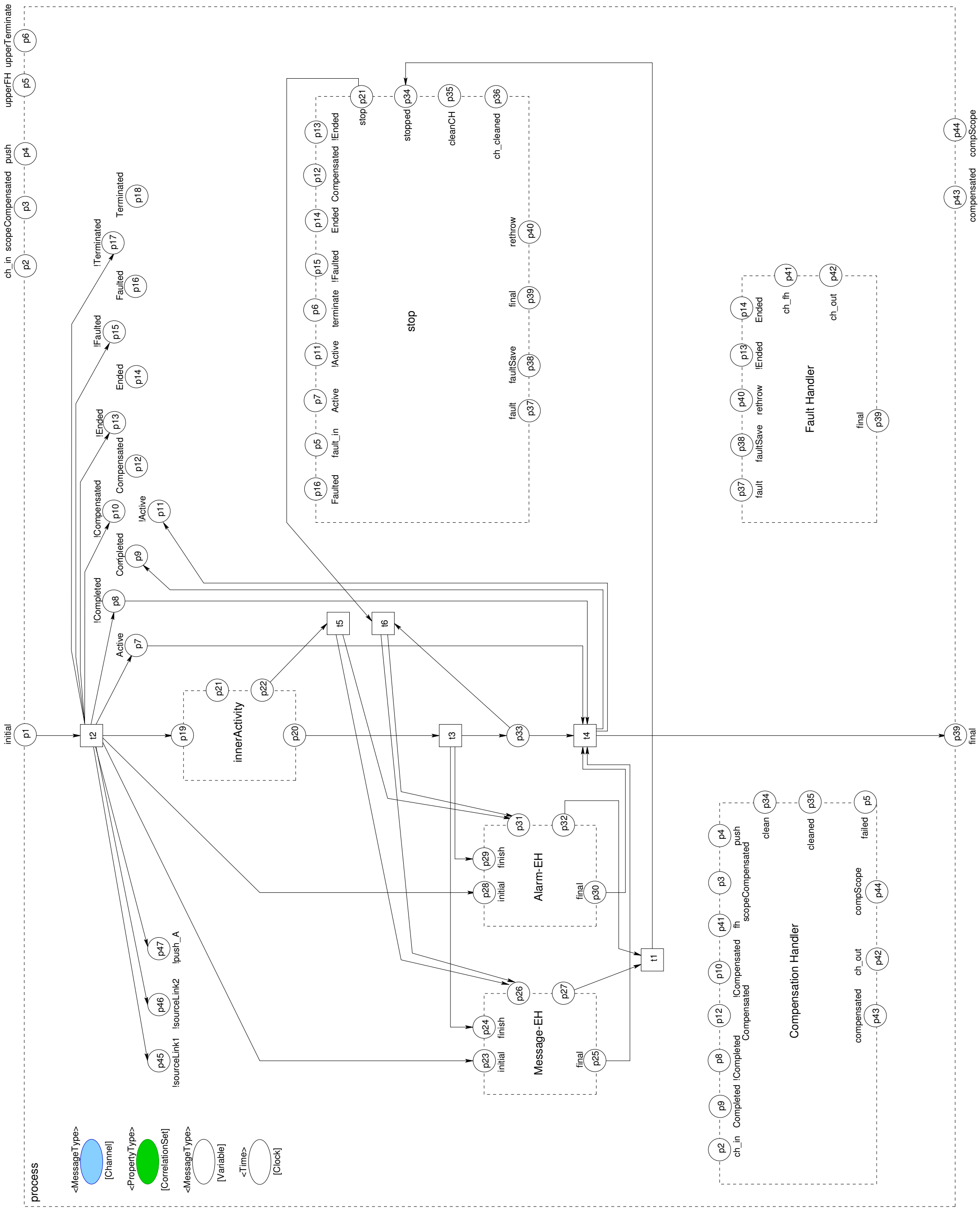


Figure 28: Pattern of BPEL's `process`.

# 8 Transformation of BPEL's Event Handler

In this section we present the transformation of BPEL's `event handlers`. "The whole process as well as each scope can be associated with a set of event handlers that are invoked concurrently if the corresponding event occurs" [CGK+03, p. 80]. BPEL distinguishes between alarm and message `event handlers`. Whereas the alarm `event handler` defines one or more onAlarm branches and is started when a timeout event occurs, the message `event handler` defines at least one onMessage branch and is executed when a message arrives. Both kinds of branches are already known from BPEL's `pick` activity (see Sec. 5.5). We present the respective patterns in Sections 8.1 and 8.2, respectively.

Note, an `event handler` is no activity. Thus, we have to change the interface by adding a place *finish*. If the `scope`'s inner activity is finished, the `event handler` receives the signal finish, asking the `event handler` to finish the execution of the inner activity and then to finish itself.

## 8.1 Alarm Event Handler

Figure 29 depicts the pattern of BPEL's alarm `event handler` which has two onAlarm branches. The general pattern with $n$ branches is as expected. The idea of this pattern is as follows: When it gets started, the alarm `event handler` reads a time stamp which is copied into each branch (t5). Such a branch is modelled as it is in the pick pattern (see Sec. 5.5). If an alarm is signalled, the respective branch and so its inner activity is executed. When receiving the signal finish, all branches running are executed until they have finished and afterwards they are synchronized (t7).

By firing t5 both branches are activated. If an alarm occurs, the respective branch is chosen, i.e. either t1 or t9 fires. Then, either innerActivity1 or innerActivity2 is executed. As long as control flows in the `scope`, running is marked. After the `scope`'s inner activity is finished, a token is produced on finish (i.e. transition t3 in Fig. 27 in Sec. 7.2 fires). During the finish procedure place finishing is marked. The specification demands to delay this procedure until all active branches have completed. Thus, the token in each branch is either removed before or after its inner activity is executed. By firing t7 all branches are synchronized and the `event handler` is finished. Due to the involved concurrency there could be a conflict between t3 and t4 (and also in the branch on the right) when p3 and p5 are marked and the transition guard is evaluated to true.

In order to stop this pattern we need to consider two scenarios: stop is marked either before or after the finish signal is received. It is impossible to receive stop before finish (see the process pattern in Fig. 28). Both scenarios can be distinguished by either a token on place running or finishing. When stop is marked before signal finish is received, the token on running is consumed by firing the transition normalStop. So t11 could not be activated anymore. Then each branch is stopped concurrently and at the end all branches are synchronized (t16). Otherwise, if the finish procedure is running, the token on finishing is consumed by firing transition stop+finish. Therefore t7 could not be activated anymore. Afterwards in every branch the finish token has to be removed first. Then the branches

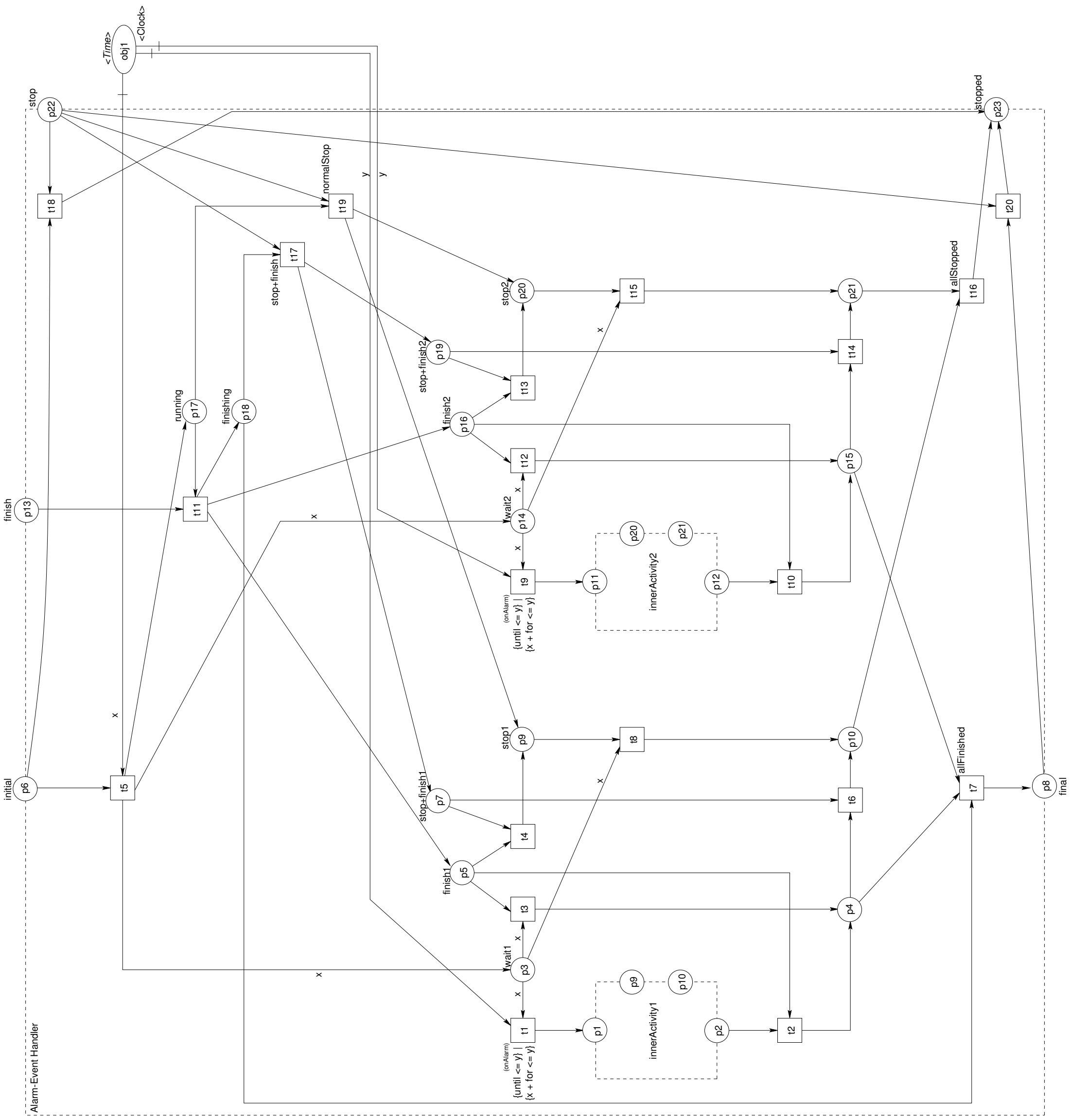


Figure 29: Pattern of BPEL's alarm `event handler` with two branches.

are stopped as in the former scenario. Note, when signals stop and finish are received concurrently it is possible that place finish is still marked at the end of the stopping procedure. In other words, for the `event handler` we cannot guarantee that all tokens will always be removed after the stopping procedure.

## 8.2 Message Event Handler

A message `event handler` defines one or more onMessage branches, where each of them specifies an attribute initiate. So we have built two patterns. The pattern of BPEL's

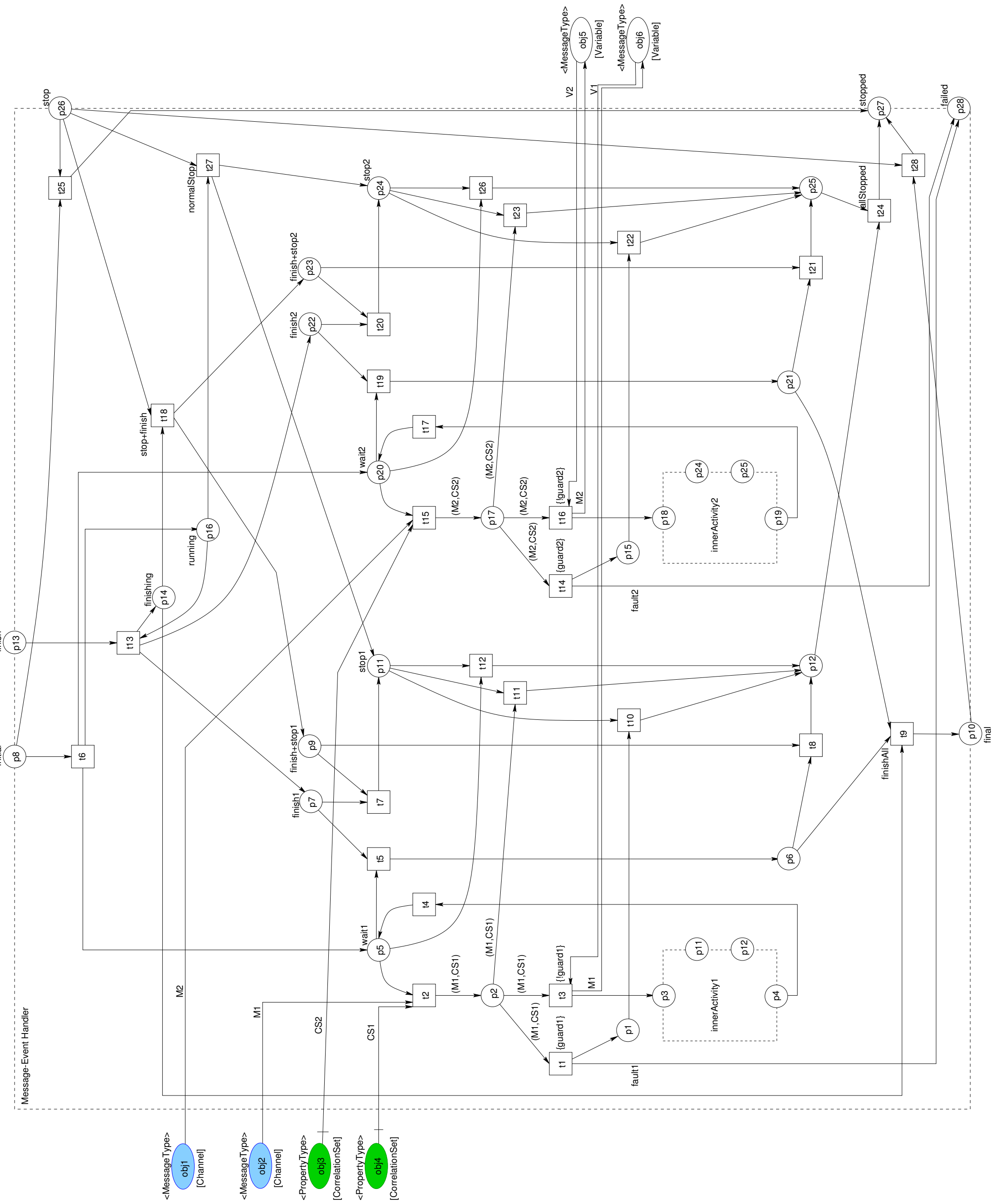


Figure 30: Pattern of BPEL's message **event handler** in case of initiate="no" that contains two onMessage branches.

message **event handler** defining two onMessage branches in case of initiate="no" is visualized in Fig. 30. Again, the general pattern with $n$ branches is as expected.

It can easily be seen that the pattern of BPEL's message **event handler** is quite similar to the alarm `event handler` in Fig. 29. Firstly, all branches are activated (t6). Afterwards, if a message is received, the inner activity of the respective branch is executed. When receiving the signal finish, all branches that are still running will be executed until they have finished and finally they get synchronized (t9). Such an onMessage branch is modelled as in the pattern of BPEL's `pick`. But in contrast to the `pick`, a branch can receive a message once more after it is executed faultlessly. The interleaving of the pattern is similar to Fig. 29.

In order to stop the message `event handler` we need to consider the same two scenarios as in the alarm `event handler`. The realization is similar to Fig. 29 and it is also possible that place finish is still marked at the end of the stopping procedure.

The equivalent pattern in case of initiate="yes" is depicted in Fig. 31. Only the onMessage branches differ from the one in Fig. 30. We add places p29, p30 and transitions t29 – t34. All other net elements have the same identifier as in Fig. 30.

Let us have a look at the left branch in Fig. 31 in order to explain the differences to Fig. 30: If the branch receives its first message, the `correlation set` (obj4) has to be initialized. Transition t2 fires and a token, an object of type (<MessageType>, $\emptyset$), is produced on place p2. Every further message is received by firing t29. That produces a token, an object of type (<MessageType>, <PropertyType>), on place p2. In this case, the `correlation set` being already initialized only has to be read. Note, transition guard guard1 must distinguish between receiving the first (scenario 1) and a following message (scenario 2). This can easily be done by evaluating the objects described above. In scenario 1 only the first set of the tupel needs to be evaluated, because a standard fault like `correlationViolation` (signalling that the <PropertyType> of the `correlation set` is not equal to the <PropertyType> defined in the received message) cannot occur, because the `correlation set` is just initialized by the message. In contrast in scenario 2, where the `correlation set` is already initialized, `correlationViolation` can occur. Thus, both – the message and the `correlation set` – have to be evaluated.

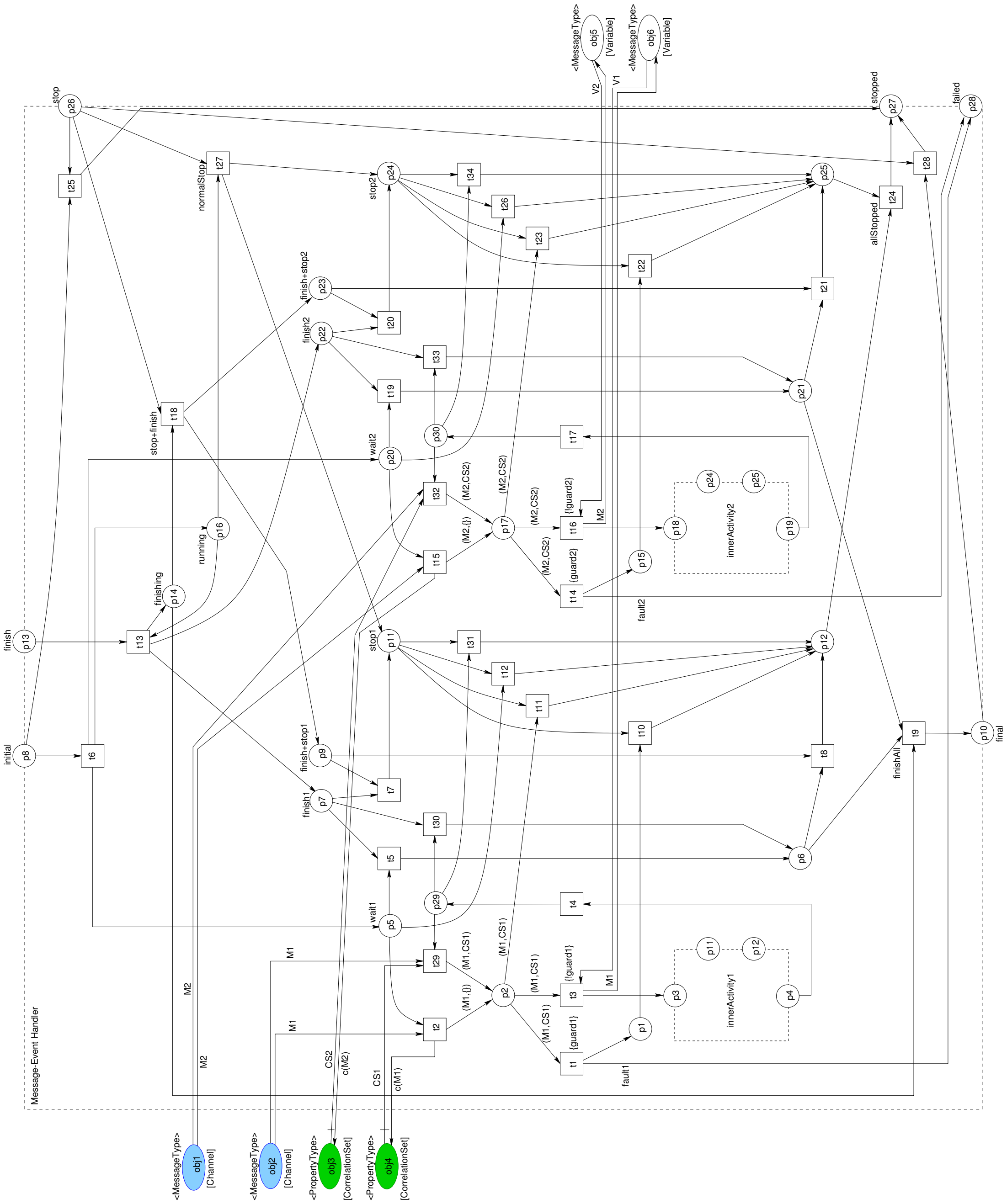


Figure 31: Pattern of BPEL's message **event handler** in case of initiate="yes" that contains two onMessage branches.

# 9 Transformation of BPEL's Fault Handler

This section shows, how to transform the `fault handler`, a further component of BPEL's `scope`. Its "sole aim is to undo the partial and unsuccessful work of a scope in which a fault has occurred" [CGK+03, p.75]. Firstly, the control flow of the corresponding `scope` has to be finished. Secondly, the `fault handler` tries to match the fault with one of its catch activities using the fault's name or the associated fault data. If the fault can be matched, it is handled. Otherwise, it is rethrown to the `fault handler` of the enclosing `scope`. In any case a "scope in which a fault occurred is considered to have ended abnormally, whether or not the fault was caught and handled without rethrow by a fault handler" [CGK+03, p.77].

In order to finish the control flow of a `scope`, a stop component is embedded in every activity's pattern. Furthermore, every `scope` contains a so-called stop pattern which triggers the stopping of the `scope` by signalling a stop signal to the inner activity of the `scope`. After the stopping of the `scope` the stop pattern signals the fault to the `fault handler`. The transformation of the stop pattern is described in Sec. 9.1.

We have to distinguish the implicit and the user defined `fault handler`. The transformation of both constructs is explained in Sec. 9.2 and 9.3, respectively.

Throughout this section we need to take a look at more than one pattern. We rather have to take a look at the interplay between a `scope`, its inner activity and its enclosed `scope`. For this purpose it is useful to make the following commitment: The pattern we have a look at is embedded in a `scope` B. B itself embeds a `scope` C called the *child scope* of B. Furthermore B is child scope of A or in other words: A is the *parent scope* of B.

## 9.1 The Stop Pattern

After an activity has thrown a fault, the `fault handler` of the enclosing `scope` has to finish the positive control flow inside the `scope` first. Afterwards it has to handle the fault. Every `fault handler`, implicit as well as user defined, behaves this way. We keep this division and extend every `scope` by a so-called stop pattern which has no equivalent construct in BPEL. When the stop pattern receives the fault, it finishes its enclosing `scope` and afterwards it signals the fault to the `scope`'s `fault handler`. Furthermore the stop pattern is also used to realize BPEL's `terminate` activity, i.e. to stop the entire process. We start with the transformation of the stop pattern of a `scope` (Fig. 32 in Sec. 9.1.1). After this we introduce the stop pattern embedded in a process pattern (Fig. 33 in Sec. 9.1.2).

### 9.1.1 The Stop Pattern Embedded in a Scope

Figure 32 depicts the stop pattern of a `scope`. First of all we have a look at the interface of Fig. 32 which differs from the former patterns. On top there are four important places: ft_in (marked if A wants B to be stopped), fault_in (a fault has occurred in an enclosing activity of B, i.e. either there is a token on a failed place or C's `fault handler`

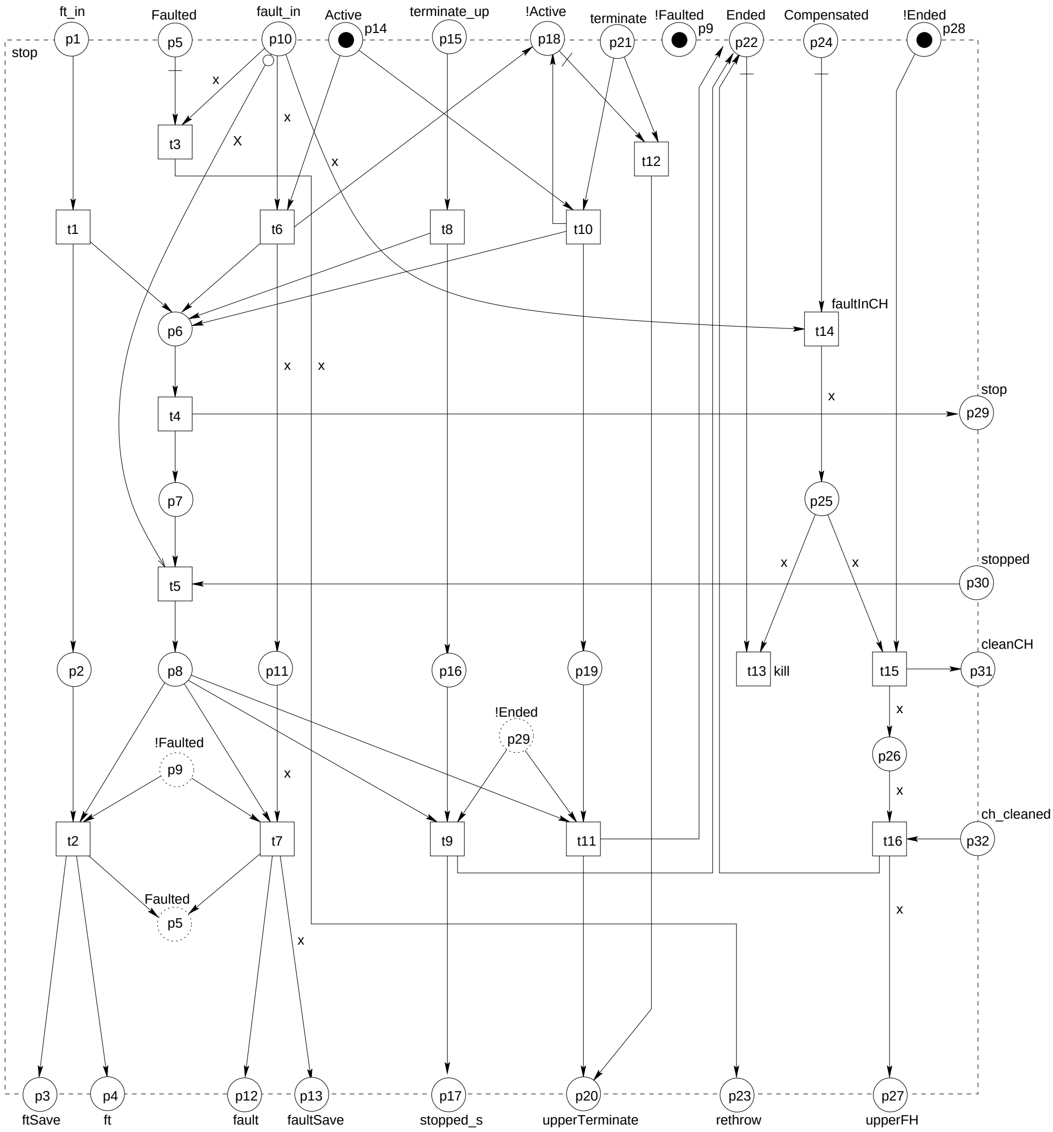


Figure 32: Stop pattern embedded in a `scope`.

rethrows a fault it cannot handle), terminate_up (a `terminate` activity embedded in A is activated) and terminate (a `terminate` activity either embedded in C or in B is activated). The place fault_in results from joining the failed places of all activities enclosed by B. All other interface places on top are state places of B. The places on the right are used to remove all tokens in B's `compensation handler` (cleanCH, ch_cleaned) and to stop the positive control flow of B (stop, stopped). On the bottom there are places to activate other patterns: ft and ft_fault (signalling that A wants to stop B), fault and faultSave (signalling the occurrence of a fault) and rethrow (signalling the occurrence of a fault during the execution of B's `fault handler`) activate the `fault handler` of B. In contrast, upperTerminate (signals `scope` A that it has to be terminated) and upperFH (rethrows a fault to A's `fault handler` that could not be handled by B's `fault handler`) activate the parent scope and the parent scope's `fault handler`, respectively. stopped_s is the stopped place of B.

Next we explain the most important scenarios of the stop pattern. At first we describe a special behaviour in BPEL and how this behaviour is modelled in the stop pattern. In Fig. 32 the subnet consisting of places p6 – p8 and transitions t4, t5 plays a key role. In this subnet a token on fault_in (i.e. a signalled fault) leads to the sending of the stop signal. In more detail, variable x holds the fault information. If there is more than one token on fault_in, one of them is nondeterministically chosen. Nondeterminism is a suitable modelling, because it is nearly impossible to realize which of two or more signals arrive first. As a result of firing t6 and t4, place stop is marked. This leads to a token on place stopped which means that the control flow of `scope` B (and therefore its child scope C as well) is finished successfully. For a proof see [Sta04]. Of course, at this point no fault can occur, because B is finished. The remaining tokens on place fault_in are removed by firing t5 where X is a tupel holding the fault information of all remaining faults. In the pattern this is realized by a special arc – a *reset arc* [DFS98] (the one between p10 and t5 having a little circle at its source). This arc consumes all tokens of p10 making no difference if there are 0, 1 or more tokens on this place. In other words, p10 is emptied.

**Scenario1:** In `scope` A a fault is thrown. A's `fault handler` has to finish the inner activity of A and of B as well. To stop its child scope, A signals `forcedTermination` to B.
**Realization:** B is in state !Terminated and Active. Place stop in B is marked; thus, t7 in the scope pattern fires. B changes into state !Active and a token is produced on ft_in in the stop pattern. By firing the transition sequence t1, t4, t5, t2 `scope` B is stopped and changes into state Faulted. At the end of this scenario the places ft and ftSave are marked.

**Scenario2:** `Scope` B is executed if one of its embedded activities signals a fault. Alternatively a fault, which occurred in `scope` C, is rethrown to `scope` B. In both scenarios `scope` B has to be finished first. Afterwards the `fault handler` of B is activated and tries to handle the fault.
**Realization:** B is in state Active. If a fault occurs in an inner activity of `scope` B, its place failed is marked and therefore place fault_in is marked as well. In contrast, if C rethrows a fault, the respective token is produced on place upperFH in B's `scope` joined with fault_in. That means, in both scenarios place fault_in is marked. Then the transition sequence t6, t4, t5, t7 is fired. As a result B is finished. It has changed into state !Active and Faulted. Furthermore fault and faultSave are marked. If more than one fault occurs, one fault (i.e. one token) is chosen nondeterministically.

**Scenario3:** B or C embed an activity `terminate` that becomes active. That means, the entire BPEL process has to be finished without calling the `fault handler` nor the `compensation handler`.
**Realization:** Let B be in state Active. If activity `terminate` is embedded in C, a

token is produced on place upperTerminate of B's scope joined with terminate of B's stop pattern. Note, it is possible that either C still stops its inner activity or C's fault handler is still executed. If activity terminate is embedded in scope B, a token is also produced on place terminate, because it is joined with the terminate place of the terminate pattern. In other words, in both scenarios place terminate is marked. The implementation of the activity terminate is realized with the stop components. By firing the transition sequence t10, t4, t5, t11 B changes into state !Active and B stops. In contrast to the former scenarios, scope B changes into state Ended. Furthermore upperTerminate is marked, propagating the terminate signal to A.

**Scenario4:** Analogue to Scenario3. As the only difference, B's fault handler is activated.
**Realization:** Again, place terminate is marked. Scope B is in state !Active, because its fault handler is activated. t12 fires and a token is produced on upperTerminate that signals the terminate signal to the parent scope A. Scope B can be "left", because the active fault handler has already finished B or it will do so. Therefore the termination of A can be started.

**Scenario5:** Analogue to Scenario3, with the terminate activity being embedded in the parent scope A.
**Realization:** A has to finish its inner activity and therefore B as well. This is done by sending the stop signal to B. There is a token on B's stop place and B is in state Active (up to now no fault has occurred) and Terminate. Therefore in scope B t8 becomes active and produces a token on place terminate_up in B's stop pattern. B also changes into state !Active. Then the transition sequence t8, t4, t5, t9 is fired. Afterwards B is finished. It has changed into state Ended and there is a token on place stopped_s which is joined with B's stopped place signalling A that B was finished.

**Scenario6:** Analogue to Scenario5. As the only difference B's fault handler is activated.
**Realization:** B is in state !Active, because its fault handler is still executed. Furthermore B's stop place is marked, but no transition of its post-set is enabled. Thus, the propagation of the stop signal has to wait as long as the fault handler is running. If the fault handler can handle the fault, the control flow will go on immediately after scope B. However, the control flow can be stopped by the token on stop. In contrast, if the fault handler cannot handle the fault (i.e. the fault is rethrown to A), B is in state Ended. Then, t10 in scope B is enabled and by firing this transition, place stopped is marked signalling that B was finished.

**Scenario7:** Let scope B contain a user defined fault handler. Let f1 be a fault causing B to finish. f1 is handled by B's fault handler. During the execution of this fault handler another fault, f2, is signalled which cannot be handled by the fault handler itself (because its inner activity which throws f2 has no enclosing scope). Thus, B's fault handler is finished and f2 is rethrown to A.
**Realization:** Fault f1 initiates Scenario2. Afterwards B is in state Faulted and !Ended and B's fault handler is still active. If f2 occurs, a token is produced on fault_in. At the end of Scenario2 all faults are removed. Thus, f2 is the only token on fault_in. If more

than one fault has occurred by handling fault f1, the respective number of tokens is on fault_in. Transition t3 is fired and as a result, a token is produced on place rethrow. B is still in state Faulted. Therefore for every fault being thrown by executing the `fault handler` a token is produced on rethrow.

**Scenario8:** Let `scope` B contain a user defined `compensation handler` whose inner activity is executed. A fault occurs within the `compensation handler`. This fault cannot be caught by the `compensation handler` of B, because the activity throwing the fault is not enclosed by a `scope`. Thus, the fault is rethrown to the `fault handler` of A.
**Realization:** Starting the execution of B's `compensation handler`, B changes into state Compensated. If a fault occurs, a token is on place failed (in the respective activity inside the `compensation handler`). This place is also joined with fault_in in the stop pattern of B. Thus, fault_in is marked. As already mentioned in Scenario2, more than one token on place fault_in is possible, too. t14 is enabled and variable x holds the information of the fault which is nondeterministically chosen. Firing t15 and t16, B changes into state Ended and the tokens in B's `compensation handler` are removed (token on place cleanCH). Finally, the fault is rethrown to the parent scope's `fault handler` (place upperFH). All other faults (i.e. tokens on place fault_in) are removed by firing t14 and t13. Thus, at the end the stop pattern is cleared.

### 9.1.2 The Stop Pattern Embedded in a Process

Now we take a look at the stop pattern embedded in the process pattern (see Fig. 28 in Sec. 7.3). It is depicted in Fig. 33. Comparing this Figure with Fig. 32 only a few differences can be established:

Both patterns have the same interfaces except places ft_in and terminate_up which are missing in Fig. 33, because a `process` has no parent scope. Furthermore, Fig. 33 has a new place final, the final place of the process pattern.

Let us have a look at the possible scenarios of Fig. 33 with respect to changes compared to Fig. 32. We keep the same numbering of the scenarios as in Sec. 9.1.1 for Fig. 32. Scenarios not listed in the following do not differ to the respective scenario presented in Sec. 9.1.1. In the following B is the `process`.

**Scenario1:** It cannot occur, because a `process` has no parent scope.

**Scenario3:** For description see Scenario3 in Sec. 9.1.1.
**Realization:** We continue our explanation where place terminate is marked. By firing the transition sequence t6, t2, t3, t7 the control flow of B is finished. At the end there is a token on final, because the whole `process` is finished.

**Scenario4:** For description see Scenario4 in Sec. 9.1.1.
**Realization:** Place terminate is marked and B's `fault handler` is active. After the execution of the `fault handler` the `process` is finished. Thus the terminate signal is not needed. So the token on terminate is removed by firing t8. In this scenario the `process` can deadlock (see Scenario4 in Sec. 9.2.1). In our opinion, this is a bug in BPEL and it is not caused by our Petri net model.

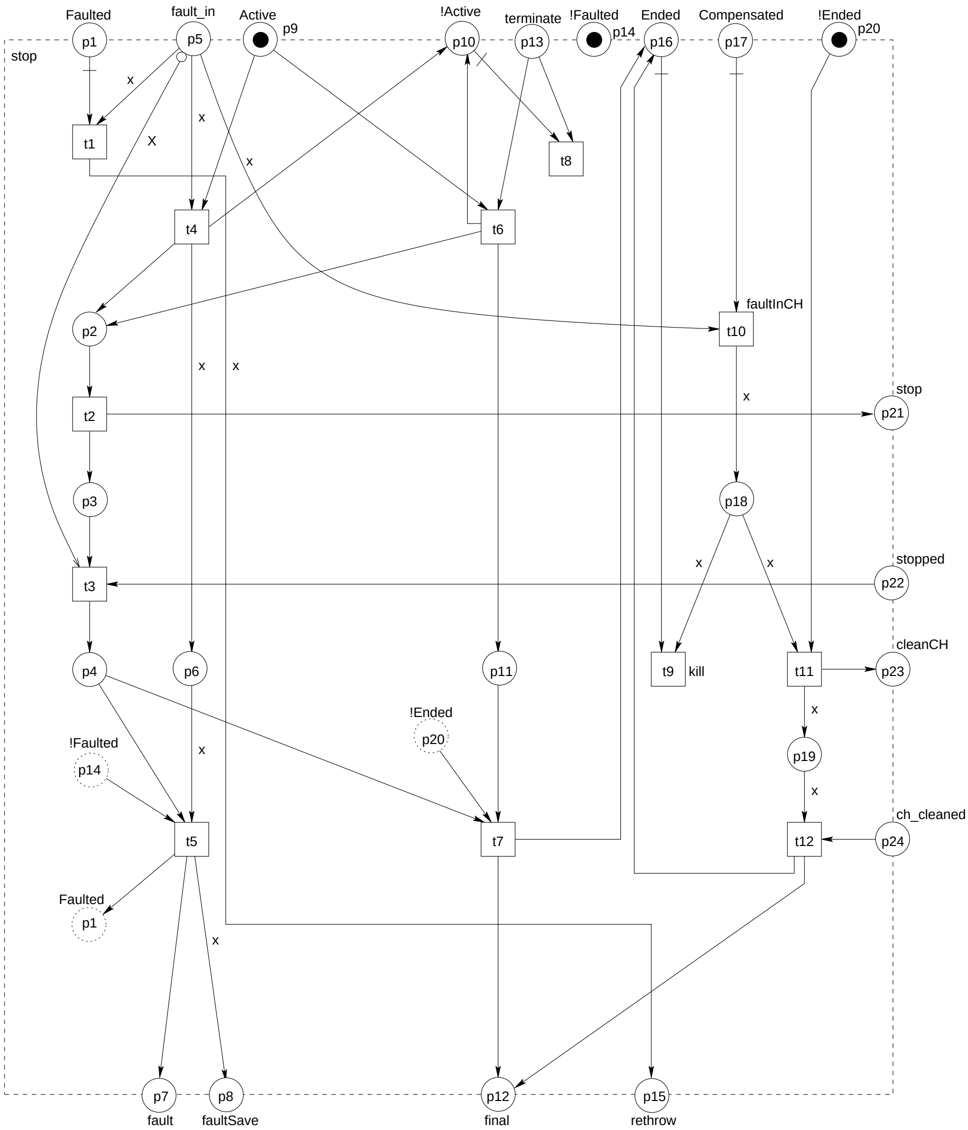


Figure 33: Stop pattern embedded in a `process`.

**Scenario5 & 6:** Both scenarios cannot occur, because a `process` has no parent scope.

**Scenario7:** See Scenario7 in Sec. 9.1.1. B is the `process`, i.e. fault f2 cannot be rethrown (because there is no parent scope). The `process` is finished instead.
**Realization:** For description see Scenario7 in Sec. 9.1.1.

**Scenario8:** BPEL allows the compensation of an executed process instance when the attribute `enableInstanceCompensation` is set to "yes". Let the `compensation handler` of `process` B be active. Let further during the execution of the `compensation handler` a fault occur that cannot be handled. This fault cannot be rethrown, because B is the `process`. Furthermore, the `fault handler` of B cannot be activated, too. Therefore B

is finished faulty.
**Realization:** For description see Scenario7 in Sec. 9.1.1. If t12 is fired, the fault is not rethrown to the fault handler of the parent scope. A token is produced on place final, i.e. the process is finished.

## 9.2 Implicit Fault Handler

Let B be a scope that embeds an implicit fault handler. Before the fault handler can be activated, scope B has to be finished. Then the fault handler works as follows (see [CGK$^+$03, p.78]): Firstly, it runs all available compensation handlers for immediately enclosed scopes in the reverse order of completion of the corresponding scopes. More detailed, activity <compensate/> is executed. Secondly, the fault is rethrown to the parent scope. If the fault was executed faultlessly, all outgoing links, for which scope B functions as a source, have to be set to true. Otherwise, these links have to be set to false. Furthermore, all outgoing links whose source activity is embedded in scope B and that are not executed anymore because of the fault have to be set to false, too[7].

To finish the scope in the model, the stop pattern is used. Therefore the fault handler's work is reduced to three tasks: Firstly, the child scopes are compensated, whereby the fault handler only initiates the compensation process with the help of the activity <compensate/>. Secondly, the fault is rethrown. Thirdly, the links are set. As the implicit fault handler has the same behaviour as the fault handler for the standard fault forcedTermination, both patterns are merged to one pattern illustrated in Fig. 34 in Sec. 9.2.1. The special case, the implicit fault handler embedded in the process pattern, is shown in Fig. 35 in Sec. 9.2.2.

### 9.2.1 Implicit Fault Handler Embedded in a Scope

In this section we present the implicit fault handler embedded in a scope. The pattern is depicted in Fig. 34.

Let us have a look at the interface of Fig. 34. Places ft, fault, ftSave, faultSave, and rethrow are the same as the ones depicted in the interface on the bottom of the stop pattern (see Fig. 32). Whereas ft and fault are marked in order to activate activity <compensate/>, places faultSave and rethrow hold the information of the fault. In contrast, place ftSave only holds black tokens. The places ft and faultSave are marked, if the fault forcedTermination shall be handled. The places fault and faultSave activate the implicit fault handler. If a fault is thrown during the execution of the fault handler, place rethrow is marked. Ended and !Ended are (already known) state places of the scope. On the right, the places ch_fh and ch_out are used to invoke B's compensation handler and receive the signal that the compensation handler has finished. Both places are joined with fh and ch_out in the compensation handler pattern (see Fig. 42 in Sec. 10.3). The

[7] In our model all outgoing links are set to false. That means, those links whose source activity is not executed anymore as well as those links whose source activity was already executed. This modelling decision is described in Sec. 7.2

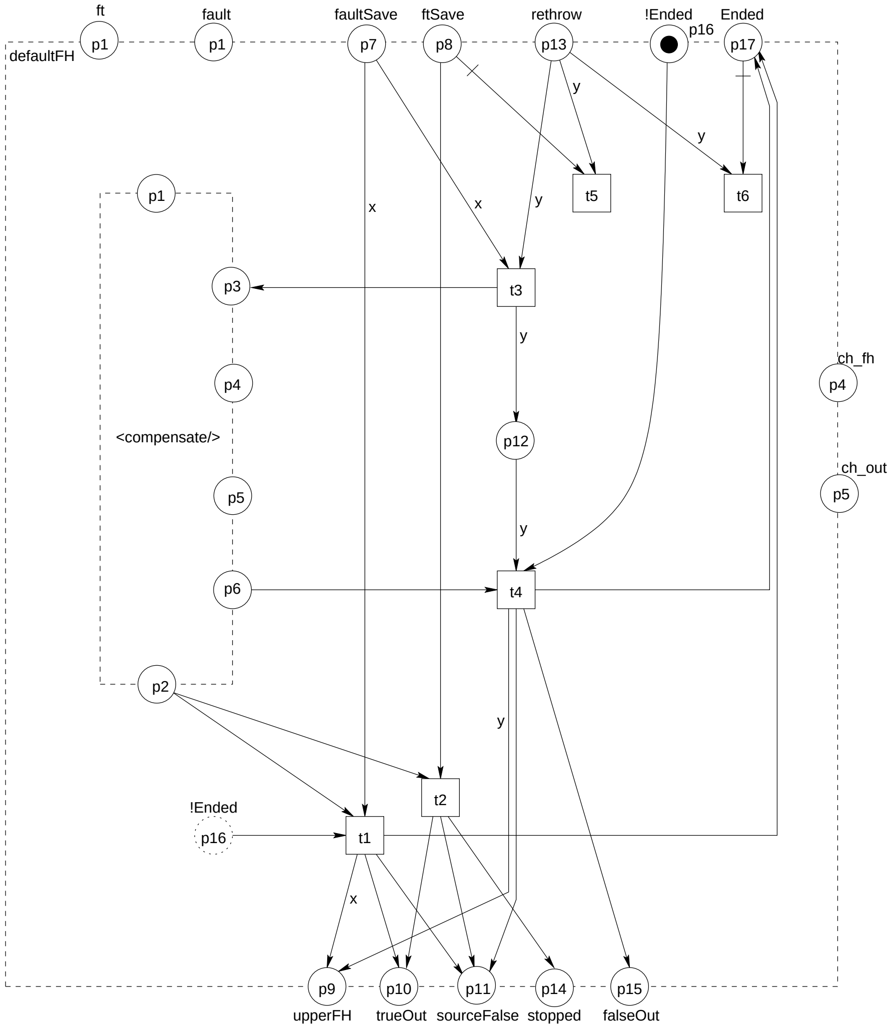


Figure 34: Pattern of the implicit `fault handler` embedded in a scope pattern.

places on the bottom have to forward the results of the fault handling: Place upperFH is the same place as the one with the same name in the pattern of the parent scope. Place stopped is the stopped place of `scope` B. A token on place trueOut (falseOut) sets all `links` to true (false) whose source activity is `scope` B. A token on place sourceFalse sets all `links` to false whose source activity is embedded in `scope` B. The exact realization of these three places was explained in Sec. 7.2.

We will explain the most important scenarios of the implicit `fault handler` as we have done it for the two stop patterns in Sec. 9.1. Again, we describe a special behaviour in BPEL and how this behaviour is modelled in Fig. 34.

**Scenario1:** This scenario completes Scenario2 described in Sec. 9.1.1. After B is finished, the implicit `fault handler` is activated and invokes all the `compensation handlers` of the child scopes (here C) in the reverse order of completion of the corre-

sponding scopes. Then the fault is rethrown to scope A.
**Realization:** After the completion of Scenario2 in the stop pattern (see Sec. 9.1.1) the places fault and faultSave are marked. Thus, <compensate/> is activated. If this activity is executed, B's compensation handler is invoked (token on place ch_fh). The compensation handler of B itself invokes C's compensation handler. After the compensation is finished faultlessly, a token is produced on place ch_out. Reasons for this unusual order of invocations are presented in Sec. 10.1. After the execution of activity <compensate/>, place p2 is marked. By firing t1, scope B changes into state Ended. The fault handler was executed faultlessly, thus links whose source activity is scope B are set to true (token on trueOut) and all other outgoing links whose source activity is embedded in scope B are set to false (token on sourceFalse). Furthermore the fault is rethrown to scope A (token on place upperFH).

**Scenario2:** Analogue to Scenario1, but during the execution of C's compensation handler a fault f occurs that is rethrown to scope B. Consequently, B's fault handler is finished and f is rethrown to scope A.
**Realization:** Fault f is rethrown to the stop pattern of B. Scope B behaves as described in Scenario7 in Sec. 9.1.1. As a result, place rethrow is marked and we continue at this point. Apart from place rethrow there is also a token on place faultSave (caused by the first fault). Activity <compensate/> is deadlocked, because it is still waiting for the answer of B's compensation handler which cannot be send anymore. By firing t3 and t4 activity <compensate/> and the fault handler are finished. Furthermore, B changes into state Ended. Variables x and y hold the information of the first fault and fault f. The latter is rethrown to A. The execution of the fault handler failed, so all links whose source activity is B are set to false (token on place falseOut) as well as the ones whose source activity is embedded in B.

If more than one fault occurs during the execution of B's fault handler, there can be more than one token on place rethrow. Only one fault that is chosen nondeterministically, is rethrown to A. All others are removed by firing t6 after B is in state Ended.

**Scenario3:** A's fault handler has to finish B. Thus, it signals the standard fault, forcedTermination. The control flow of B is finshed and B's fault handler invokes all the compensation handlers of the child scopes (here C) in the reverse order of completion of the corresponding scopes. Then the finishing of scope B is signalled to A.
**Realization:** The way how B is finished is described in Scenario1 in Sec. 9.1.1. As a result, ft and faultSave are marked. At this point we continue. Activity <compensate/> is activated and can be executed as described above in Scenario1. Transition t2 fires setting the links as in Scenario1. But instead of rethrowing a fault to A, place stopped is marked.

**Scenario4:** Analogue to Scenario3, but during the execution of C's compensation handler a fault f occurs that is rethrown to B. In B the forcedTermination fault handler is active. So the fault can neither be handled by B (because the fault handler of B is already activated) nor can it be rethrown to A (because of the forcedTermination sent by A). Consequently, B and so the whole process runs into a deadlock.

**Realization:** Fault f leads to a token on place rethrow (Scenario7 in Sec. 9.1.1) and place ftSave is marked, too. Activity <compensate/> is still waiting for a signal of B's `compensation handler` that will never be signalled because of the fault. By firing t5, the fault token is removed. Now neither the deadlock situation in which activity <compensate/> ran into is abolished nor the fault is rethrown to A. Consequently, the model is in a deadlock.

### 9.2.2 Implicit Fault Handler Embedded in the Process

Now we present the implicit `fault handler` embedded in a process pattern. This pattern is illustrated in Fig. 35. In fact, there are only few differences between this pattern and Fig. 34 described in Sec. 9.2.1. Thus, we want to restrict ourself to explain these differences. The `fault handler` embedded in a `process` has to compensate its child scopes, but there is no parent scope the fault can be rethrown to. Instead of rethrowing the fault, the `process` is finished. The `fault handler` also does not have to react to the standard fault `forcedTermination`, because it cannot occur for the same reason. Furthermore, no outgoing `links` have to be set, because on the one hand the `process` cannot be source of a `link` and on the other hand `links` whose source activity is embedded in the `process` do not have to be set to false, because the whole `process` is finished.

Looking at the interface of Fig. 35, all places are already known except for the place final which functions as the final place of the process pattern. Instead of rethrowing the fault, the `process` is finished after the execution of its `fault handler`. Thus, a token is produced on place final.

Let us now explain the possible scenarios of the implicit `fault handler` embedded in a `process`. We will only list the differences to Fig. 34. We keep the same numbering as in the last section.

**Scenario1:** See Scenario1 in Sec. 9.2.1. But instead of rethrowing the fault, the `process` is finished.

**Realization:** The places fault and faultSave are marked. Activity <compensate/> is executed. Then t1 fires and place final is marked. Thus, the `process` is finished.

**Scenario2:** See Scenario2 in Sec. 9.2.1. But instead of rethrowing the fault, the `process` is finished.

**Realization:** The places rethrow and faultSave are marked. By firing t2 and t3 activity <compensate/> is stopped and place final is marked. Thus, the `process` is finished.

**Scenario3 & 4:** Both scenarios cannot occur, because a `process` has no parent scope.

## 9.3 User Defined Fault Handler

In this section we present the transformation of BPEL's user defined `fault handler` which contains a finite number of catch branches and an optional catchAll branch. Every catch branch has an inner activity which is executed when the respective branch is chosen. The fault information of a signalled fault is compared to every catch branch. If there is a matching, i.e. the catch branch is able to handle the fault, the respective

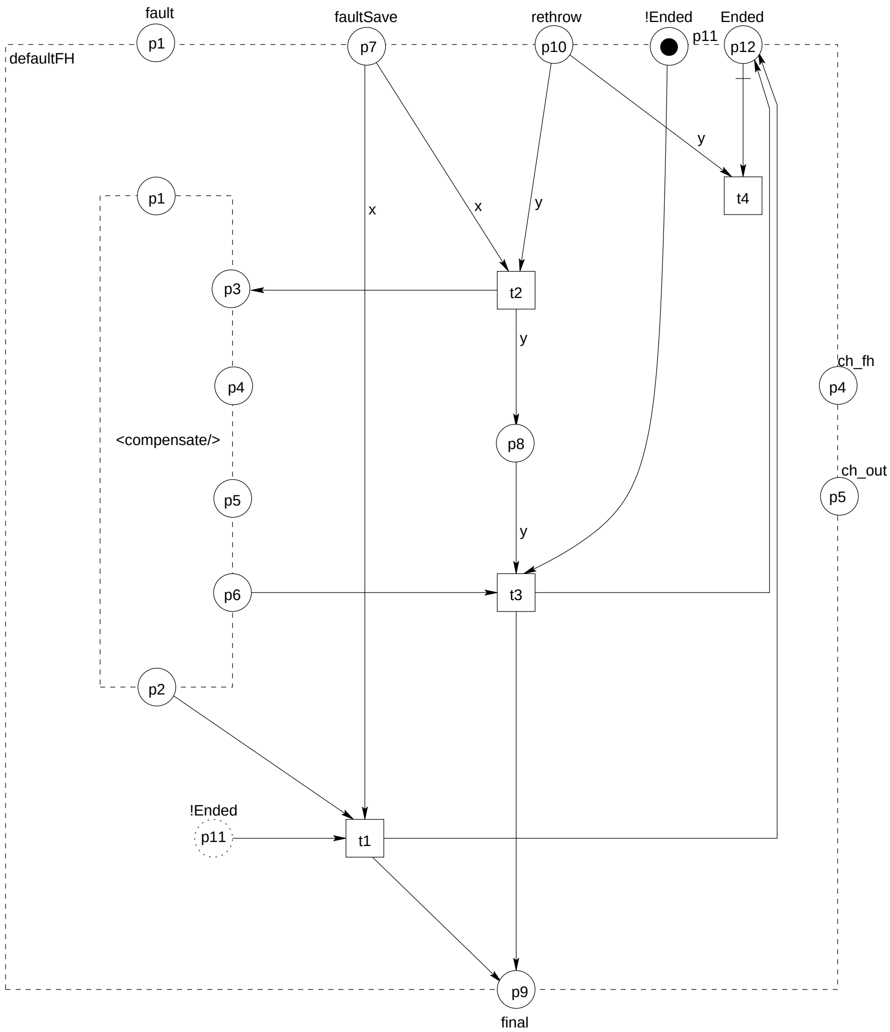


Figure 35: Pattern of the implicit `fault handler` embedded in a process pattern.

branch is chosen and its inner activity is executed. Otherwise, catchAll is executed. In case no catchAll branch is defined and none of the catch branches is chosen, the user defined `fault handler` behaves like the implicit `fault handler`: It runs all available `compensation handlers` for the immediately enclosed `scopes` in the reverse order of completion of the corresponding `scopes`. Afterwards the fault is rethrown to the parent scope. The user defined `fault handler` also embeds a special branch that catches the fault `forcedTermination`. This branch is equal to the one in the implicit `fault handler`.

The pattern of the user defined `fault handler` embedded in a `scope` is illustrated in Fig. 36 in Sec. 9.3.1. The corresponding pattern embedded in the `process` is presented in Fig. 37 in Sec. 9.3.2.

### 9.3.1 User Defined Fault Handler Embedded in a Scope

Figure 36 depicts the general pattern of a user defined `fault handler` embedded in a scope. It consists of $n - 1$ catch branches and a catchAll branch. Furthermore, there is a branch that catches the standard fault `forcedTermination`. This branch is only embedded if no catchAll is defined. For reasons of simplicity, it is illustrated in the pattern, too. A branch is chosen, if one of the transition guards {catch1}, {catchAll}, etc. holds. The evaluation depends firstly on the fault name and the fault information which is stored in variable x and secondly on the information of the branch saved in the respective transition guard.

The interface of Fig. 36 on top and on the right is equal to the interface of the pattern of the implicit `fault handler` (see Fig. 34 in Sec. 9.2.1). Only on the bottom there is a new place – out. If a catch branch is executed faultlessly, the `scope` is finished. The control flow, however, goes on as if the `scope` was executed faultlessly. This is realized by a token on out, because this place and the `scope`'s final place are identical. Comparing this pattern with the pattern of the implicit `fault handler`, both patterns only differ in the catch branches.

Again, we consider the possible scenarios that can happen in BPEL and how this behaviour is modelled in the pattern.

**Scenario1:** The control flow of A has to be finished. Therefore B has to be finished, too. That means, the fault `forcedTermination` is signalled to B. This scenario is analogue to Scenario3 and Scenario4 described in Sec. 9.2.1.

**Scenario2:** Either a fault occurs during the execution of B or C rethrows a fault to B. The fault matches a catch branch. So the inner activity of this branch (e.g. innerActivity1) is executed faultlessly. After the execution of this activity the fault handler and `scope` B are finished and the control flow continues as if B was executed faultlessly.

**Realization:** We continue where Scenario1 in Sec. 9.1.1 ends. There, the places fault and faultSave are marked. Variable x holds the fault information saved in the token on place faultSave. This data is used to evaluate the transition guards. Let guard {catch1} hold. Thus, t1 fires and innerActivity1 is executed. After its execution, place p2 is marked and t2 fires. All `links` whose source activity is B are set to true (token on place trueOut) and all other `links` whose source activity is embedded in B are set to false (token on place sourceFalse). Furthermore, B is finished positively (token on place out).

**Scenario3:** Analogue to Scenario2, but as the only difference a fault f occurs during the execution of innerActivity1. So B's `fault handler` and B itself are finished and furthermore f is rethrown to A.

**Realization:** If fault f occurs during the execution of innerActivity1, there is a token on place faultSave. B's stop pattern behaves as described in Scenario7 in Sec. 9.1.1. As a result, place rethrow is marked. Variables x and y hold the fault information of the first fault and fault f. By firing t7 and t8 innerActivity1 is stopped, i.e. all tokens inside the pattern are removed. Furthermore `scope` B changes into state Ended. All `links` (it makes no difference whether their source activity is the `scope` or whether they are embedded) are set to false (the places falseOut and sourceFalse are marked), and fault f is rethrown to A (token on place upperFH).

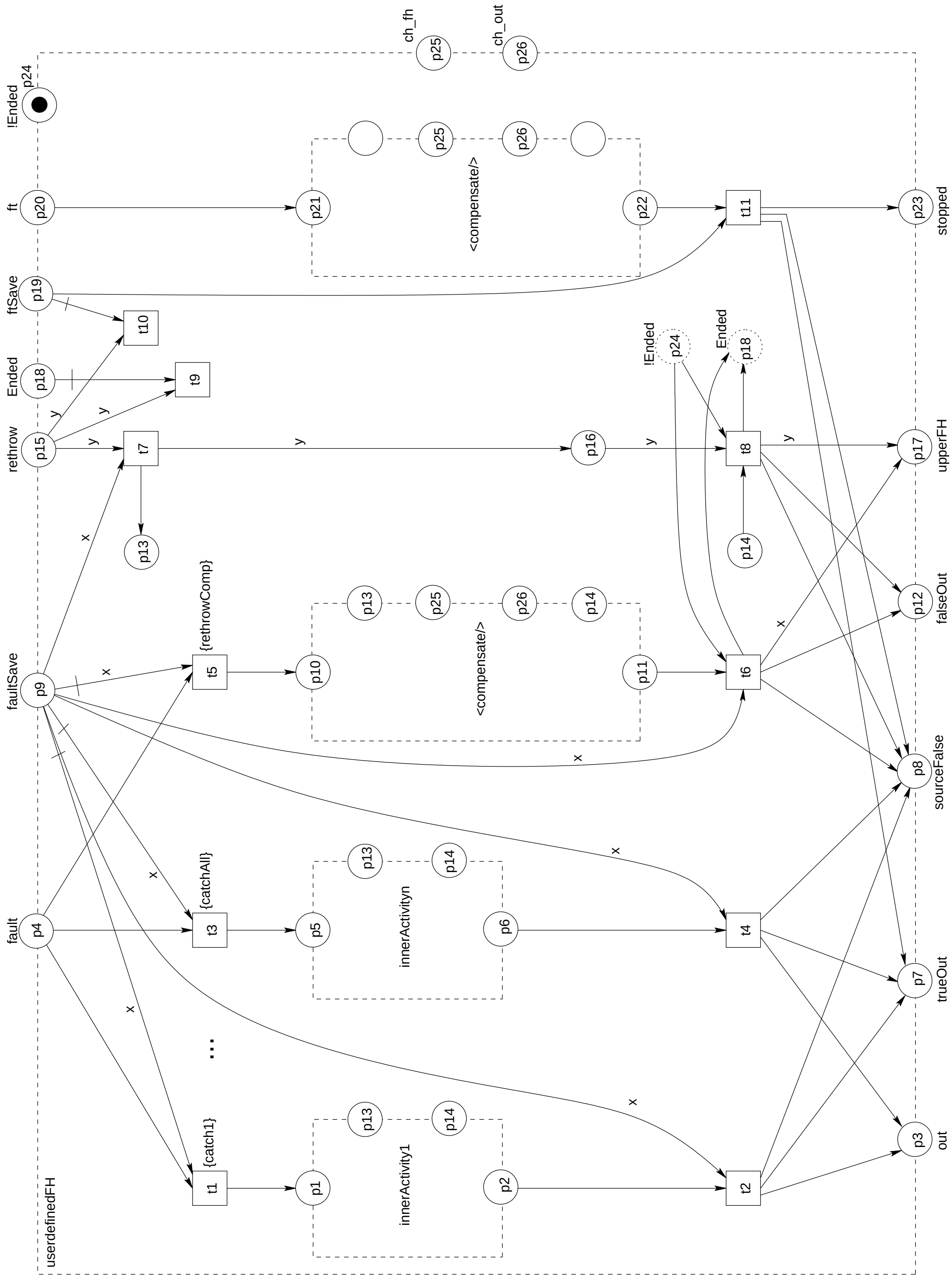


Figure 36: Pattern of the user defined `fault handler` with $n-1$ catch branches embedded in a `scope`.

If more than one fault occurred during the execution of innerActivity1, then there is more than one token on place rethrow. Variable y holds the fault information of a fault that is chosen nondeterministically. By firing t8, Ended is marked and so the remaining tokens (i.e. the remaining faults) can be removed one after another by firing t9.

**Scenario4:** Either a fault occurs during the execution of `scope` B or `scope` C rethrows a fault to B. In contrast to Scenario2, the fault does not match any catch branch and no catchAll branch is defined (in contrast to Fig. 36) as well. The `fault handler` behaves like the implicit `fault handler` described in Sec. 9.2.1.
**Realization:** As in Scenario2, the places fault and faultSave are marked. Variable x contains the fault information and the transition guard {rethrowComp} holds. t5 fires and the remaining realization is analogue to Scenario1 described in Sec. 9.2.1. In case a fault occurs during the execution of the `fault handler`, it is analogue to Scenario2 described in Sec. 9.2.1.

### 9.3.2 User Defined Fault Handler Embedded in a Process

In the following we have a look at the user defined `fault handler` embedded in a process pattern. This pattern is depicted in Fig. 37. Again, this pattern has only a few differences to the one shown in Fig. 36. We will bring out these differences. They are caused by the fact that a `process` has no parent scope and thus, it cannot rethrow any fault. Instead of rethrowing the fault, the `process` is finished. The `fault handler` also does not have to react to the standard fault `forcedTermination`, because it cannot occur for the same reason. Furthermore, no outgoing `links` have to be set, because on the one hand the `process` cannot be source of a `link` and on the other hand `links` whose source activity is embedded in the `process` do not have to be set to false, because the whole `process` is finished.

The interface is the same as in Fig. 36. So we only restrict ourself in bringing out the differences to the scenarios described in Sec. 9.3.1. We keep the same numbering of the scenarios.

**Scenario1:** This scenario cannot occur, because the `process` has no parent scope.

**Scenario2:** See Scenario2 in Sec. 9.3.1. B is the `process`. Thus, the control flow cannot be continued. The `process` is finished.
**Realization:** The same as described in Sec. 9.3.1. As the only difference, by firing t2 a token is produced on place final, i.e. the `process` is finished.

**Scenario3:** See Scenario3 in Sec. 9.3.1. B is the `process`. Thus, the control flow cannot be continued. The `process` is finished.
**Realization:** The same as described in Sec. 9.3.1. As the only difference, by firing t8 a token is produced on place final, i.e. the `process` is finished.

**Scenario4:** See Scenario4 in Sec. 9.3.1. B is the `process`. Thus, the control flow cannot be continued. The `process` is finished.
**Realization:** The same as described in Sec. 9.3.1. As the only difference by firing t6 a token is produced on place final, i.e. the `process` is finished.

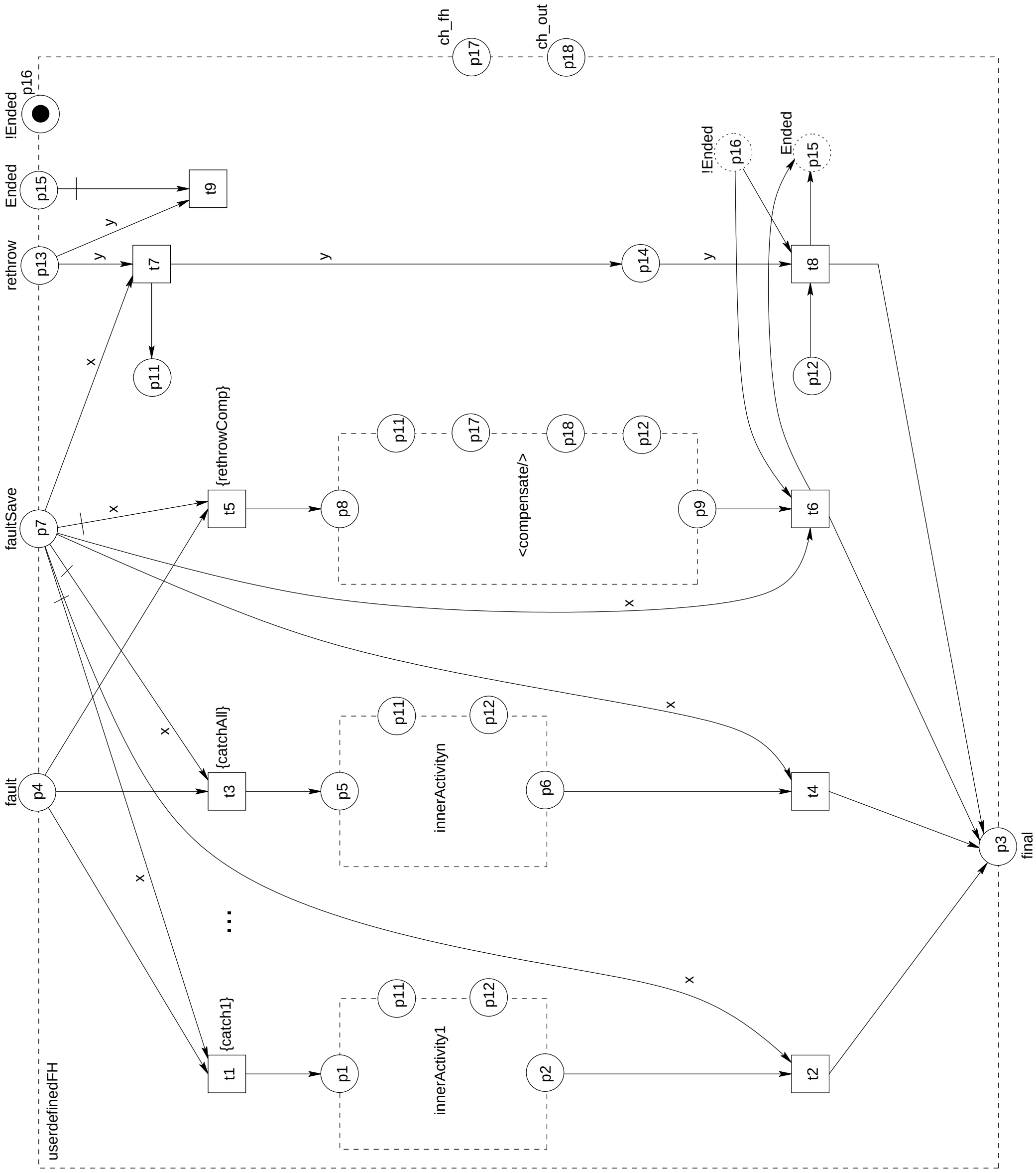


Figure 37: Pattern of the user defined fault handler with (n - 1) catch branches embedded in the `process`.

# 10 Transformation of BPEL's Compensation Handler

In this section we transform the `compensation handler`. In case a fault occurs, it is used to reverse some effects caused by the execution of the `scope`'s inner activities.

In general, the "compensation handler can be invoked by using the compensate activity, which names the scope for which the compensation is to be performed, that is, the scope whose compensation handler is to be invoked. A compensation handler for a scope is available for invocation only when the scope completes normally" [CGK$^+$03, p.74], i.e. faultlessly. "Invoking a compensation handler that has not been installed is equivalent to the empty activity" [CGK$^+$03, p.74]. Furthermore a `compensation handler` is never invoked for a `scope` in which a fault occurred.

In order to invoke a `compensation handler` the elementary activity `compensate` is needed. Section 10.2 explains the transformation of this activity.

As in the `fault handler`, BPEL distinguishes between implicit and user defined `compensation handler`. The implicit `compensation handler` runs all available `compensation handlers` for immediately enclosed `scopes` in the reverse order of completion of the corresponding `scopes`. In contrast, the user defined `compensation handler` is a wrapper for an activity that is executed, if the handler is activated. We transform both components in Sections 10.3 and 10.4.

First of all, we will explain the idea of the transformation. We will clarify the interplay of the patterns and introduce some design decisions.

For purposes of simplification we make the following commitment: The respective pattern introduced is enclosed by a `scope` B. As in Sec. 9, B is the child scope of A and the parent scope of C.

## 10.1 The Idea of Transformation

Before we go into detail and explain the patterns we will first of all introduce the idea of transformation. A `compensation handler` is invoked by using the activity `compensate` which is either embedded in the parent scope's `compensation handler` or `fault handler`. On the one hand the `compensate` can invoke a `compensation handler` explicitly. On the other hand activity `compensate` can invoke the `compensation handlers` of all child scopes in the reverse order of their execution. In the latter case, the activity `compensate` needs to know in which order the child scopes were executed.

In [Sta04] we suggested to embed a stack into every `compensation handler`. Every stack stores the names of all child scopes of its enclosing `scope` that are executed faultlessly. If `scope` B finishes its execution, it puts a token into the `compensation handler`'s stack of its parent scope. This token is an object that consists of the scope's name, B. If two `scopes` finish at nearly the same time, the two tokens would be ordered nondeterministically inside the stack. Finally, executing the implicit `compensation handler` means, to clear the stack element by element. As every token in the stack saves the name of a `scope` to be compensated, every child scope's `compensation handler` that should be invoked can be identified. This model has one disadvantage: it drastically increases the state space. To avoid this, we now suggest another approach.

Instead of using a stack, a `compensation handler` contains just two complementary places *push_scopeName* and *!push_scopeName* for each child scope. That means, if `scope` B is finished, it produces in addition to the token on place final a token on push_B and it consumes the token on !pushed_B (see transition t4 in Fig. 27 in Sec. 7.2). If the `compensation handler` of B's parent scope is activated, it invokes all child scopes that are executed faultlessly, i.e. such `scopes` that have produced a token on their respective push_scopeName place inside the `compensation handler`. The order in which the `compensation handler` invokes the `compensation handlers` of the child scopes is chosen nondeterministically. This is a reasonable modelling, because every possible order can be chosen and in a concurrent system it is very difficult to decide which one of two signals arrives first.

To embed the information about which child scope can be compensated and which one not inside the `compensation handler` causes the following problem: If activity `compensate` is embedded in a `compensation handler`, it has access to the places push_scopeName. Otherwise, if activity `compensate` is embedded in a `fault handler`, it has no access to the respective places. In order to solve this problem, activity `compensate` first invokes the `compensation handler` of the enclosing `scope`. That way it gets access to the scope names to be invoked. Looking at the BPEL specification, this invocation is not allowed. So, if a `fault handler` is activated, the `compensation handler` cannot be activated anymore. But in our model it is only a modelling decision to use these special structures to invoke the `compensation handler` of the child scopes. In order to show that our semantic preserves the properties of BPEL, we proved in [Sta04] that this invocation does not influence the `compensation handler`. We will not repeat this proof in this document, so the interested reader is referred to [Sta04]. Note, this proof is done for a `compensation handler` that embeds a stack, but the structure of the subnet that contains either the stack or the push places is the same. We must further ensure that the `scope` remains in the correct state. If a `compensation handler` is invoked by an activity `compensate`, the `scope` changes into state Compensated. Otherwise, if the `compensation handler` is only invoked in order to get access to the push_scopeName places, the `scope` does not change into state Compensated.

## 10.2 Activity Compensate

Next we present the transformation of BPEL's `compensate`. The BPEL specification distinguishes between explicit invocation of the `compensation handler` in child scope C (i.e. <compensate scope="C">) and the sequential invocation of all child scopes (i.e. <compensate/>). Furthermore, the activity `compensate` is either embedded in a `compensation handler` or in a `fault handler`. So there are 4 different components. Every component is transformed in a pattern presented in Sections 10.2.1 – 10.2.4.

### 10.2.1 <compensate/> Embedded in the Compensation Handler

<compensate/> is embedded in B's `compensation handler` and should sequentially invoke the `compensation handler` of all child scopes. The pattern's idea is to activate

the subnet. This subnet then invokes all child scopes and waits for the signal that the last child scope has been compensated. The pattern is visualized in Fig. 38.

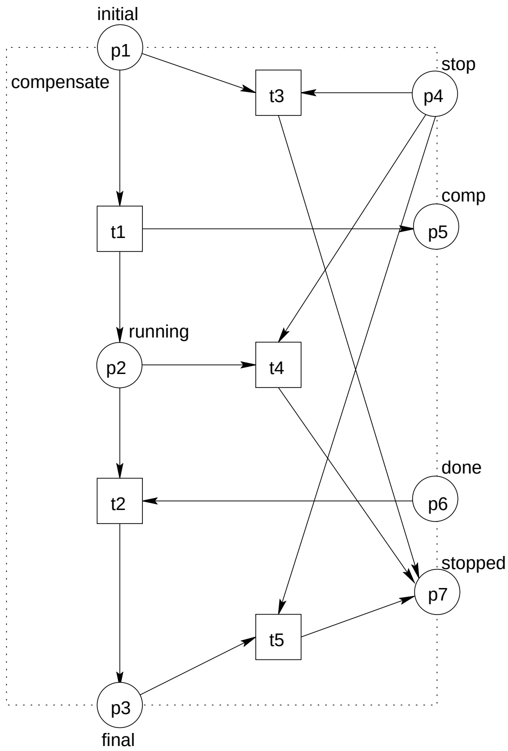


Figure 38: <compensate/> pattern embedded in a `compensation handler`.

The interface is depicted by places initial, final, stop, stopped, comp, and done. The first four places are already known from the other activity patterns. To understand the other two places, it is helpful to have a look at Fig. 43 in Sec. 10.4.1. The inner activity of this compensation handler pattern is the pattern depicted in Fig. 38. Places comp and done are the same as places p11 and p12 in Fig. 43. Furthermore, place comp and the place of the same name in the `compensation handler` in Fig. 43 are identical, too.

If place initial is marked and t1 fires, the subnet for invoking the child scopes is activated (token on place comp). After the compensation of all child scopes a token on place done is produced and t2 fires.

As expected, transitions t3 – t5 establish the pattern's stop component. Place stop is marked by the `compensation handler` (see Sec. 10.3).

### 10.2.2 <compensate/> Embedded in the Fault Handler

Figure 39 presents the pattern of activity <compensate/> embedded in a `fault handler`. This pattern invokes the `compensation handler` of the enclosing `scope` in order to get access to all names of child scopes that should be compensated. If no fault occurs during the compensation, activity <compensate/> receives the signal that all child scopes are compensated. This results in a token on place ch_out.

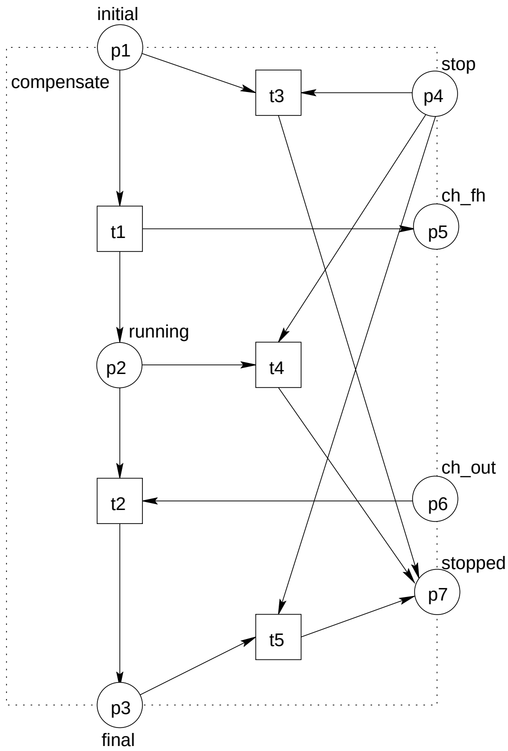


Figure 39: <compensate/> pattern embedded in the `fault handler`.

Comparing this pattern with the pattern in Fig. 38, both patterns only differ in two labels: Places p5 and p6 are labelled with ch_fh and ch_out, because they are identical to places fh and ch_out in the `compensation handler` (depicted in Fig. 42 in Sec. 10.3). Actually, the interface and the structure of both patterns are the same. For more details see Sec. 10.2.1.

### 10.2.3 <compensate scope="C"> Embedded in the Compensation Handler

Figure 40 presents the pattern of activity <compensate scope="C"> embedded in an inner activity of B's user defined `compensation handler` (see Fig. 44 in Sec. 10.4.2). The model directly invokes C's `compensation handler` by producing a token, an object that consists of the `scope`'s name (here C) on place compScope. The `compensation handlers` of B's child scopes compare that token with the names of their enclosing `scopes`. So we can guarantee that such an invocation is always unique. After C is compensated, its `compensation handler` signals its execution to the compensate pattern.

Comparing this pattern with the pattern in Fig. 38 in Sec. 10.2.1, both patterns only differ in two labels. Place p5 is labelled with compScope and place p6 is labelled with scopeCompensated. Place compScope and place p19 in the user defined `compensation`

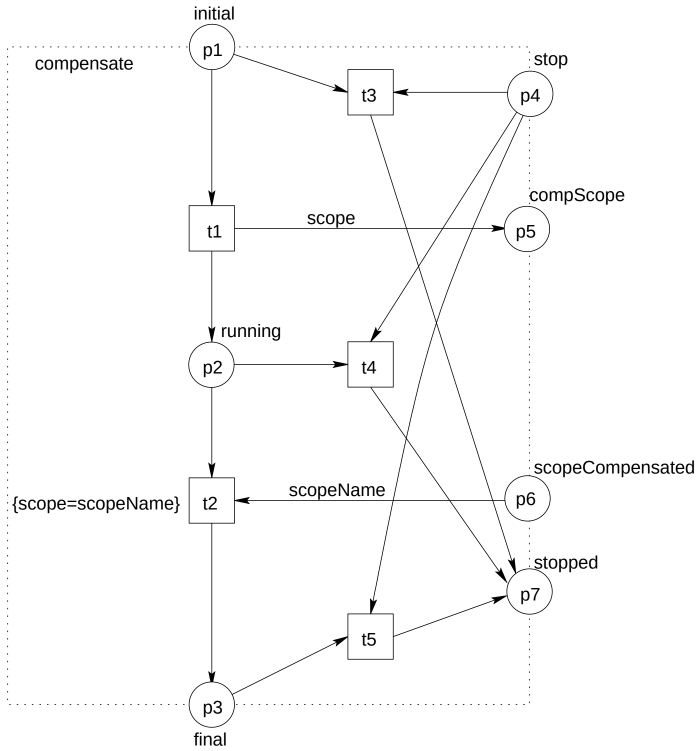


Figure 40: <compensate scope="C"> pattern embedded in the `compensation handler`

`handler` (see Fig. 44 in Sec. 10.4.2) are identical. Furthermore place scopeCompensated and place p12 in the user defined `compensation handler` are identical, too.

Let us take a look at Fig. 40. By firing `t1` variable scope saves "C", the name of the child scope to be compensated. C's `compensation handler` identifies this signal and therefore it is executed. After `scope` C is compensated, its `compensation handler` produces a token, an object that consists of the scope's name C, on place compensated. Place compensated in C's `compensation handler` and place scopeCompensated in Fig. 40 are identical. Variable scopeName holds C and variable scope holds the name of the child scope to be compensated (here C). Thus, the boolean expression holds and t2 is enabled and can fire.

With the help of the transition guard of t2, we achieve that the compensate pattern is only finished, if the token of the invoked `scope` is received. Otherwise, two activities <compensate scope="C1"> and <compensate scope="C2"> can not be executed concurrently.

### 10.2.4 <compensate scope="C"> Embedded in the Fault Handler

In this section we transform activity <compensate scope="C"> embedded in the user defined `fault handler` (see Sec. 9.3). The pattern is presented in Fig. 41.

This pattern only differs from Fig. 40 in Sec. 10.2.3 in two labels ch_in and compensated.

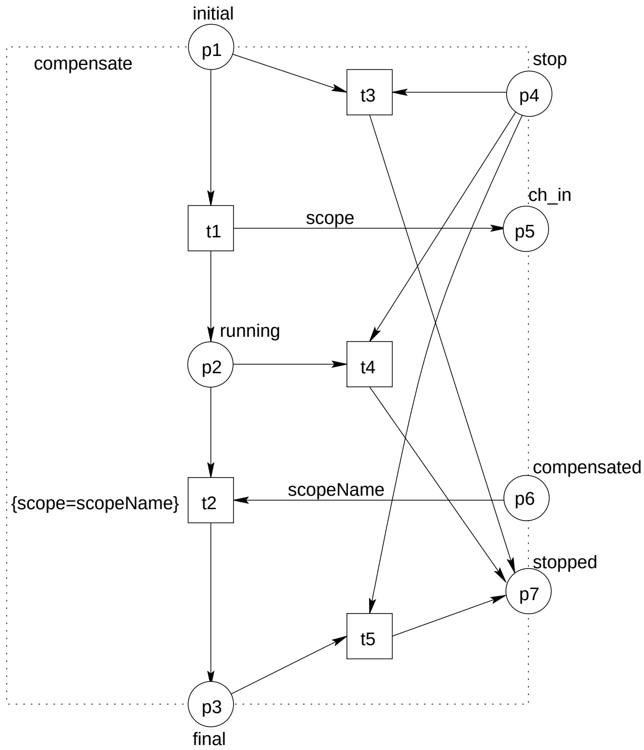


Figure 41: <compensate scope="C"> pattern embedded in the `fault handler`.

Each of the two places is the same as the respective place of the same name in C's `compensation handler`. Everything else is identically.

## 10.3 Implicit Compensation Handler

Next we show the transformation of BPEL's implicit `compensation handler`. The pattern is presented in Fig. 42. We start with an outline of the idea. After the `compensation handler` is invoked, it first tests the state of its enclosed `scope`. If the `scope` has not been completed faultlessly, the `compensation handler` is finished. If the `scope` has already been compensated, a fault is thrown. Otherwise, if the `scope` has been completed faultlessly and if it has not been compensated, the handler is executed. For all child scopes that have been completed faultlessly, the `compensation handler` is invoked. The order of invocation is chosen nondeterministically. After all child scopes have been compensated, the `compensation handler` is finished. This is signalled to the pattern that has invoked the `compensation handler` in the first place.

Now let us have a look at the interface. On top of the frame the places ch_in, fh, scopeCompensated, and p20 – p23 are important – the other places are state places (see Sec. 7.1). Usually, the `compensation handler` is invoked by a token on place ch_in. Only when the `fault handler` of the enclosed `scope` embeds the activity <compensate/> (and so needs to know all child scopes that have already been executed) a token is

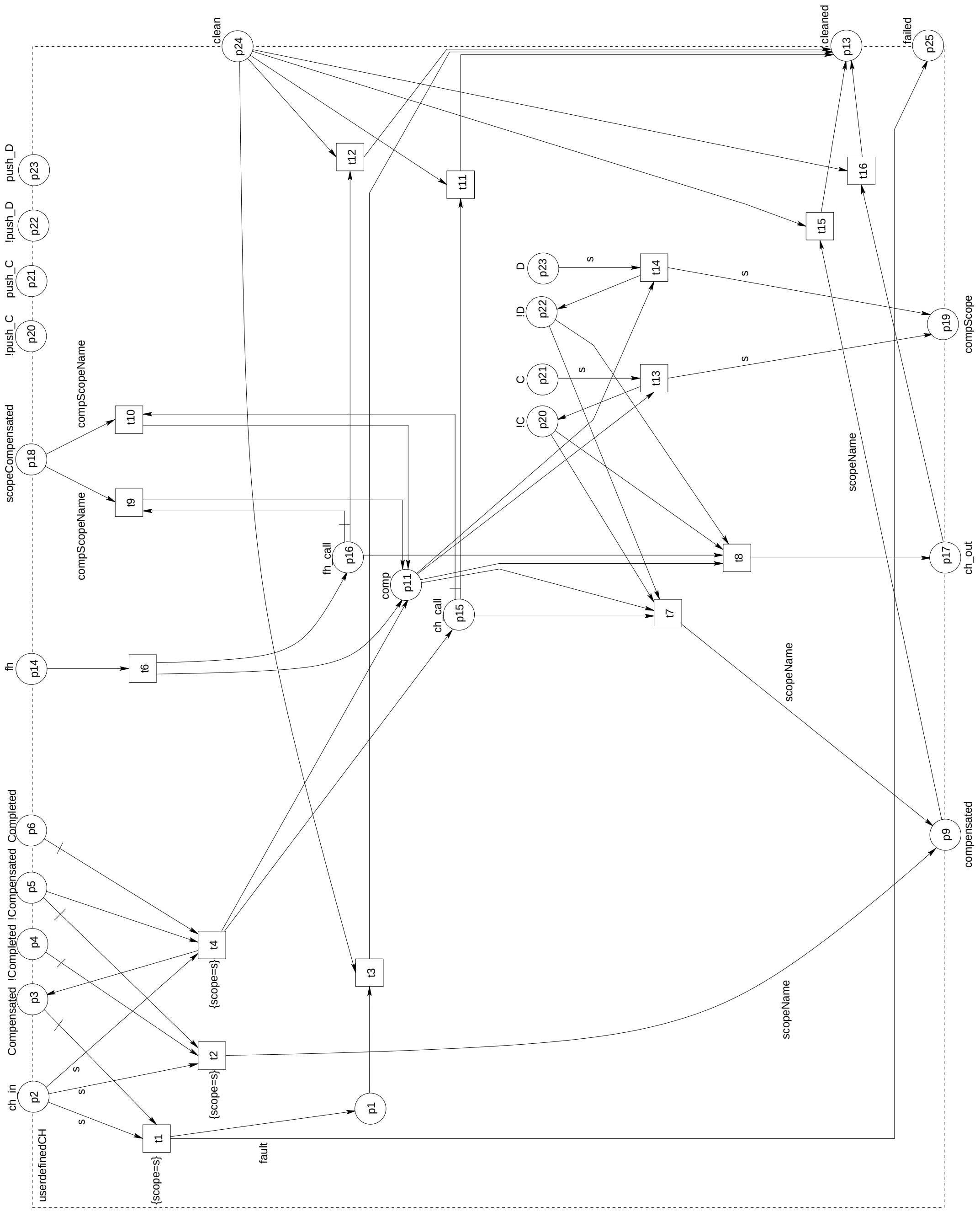


Figure 42: Pattern of the implicit `compensation handler`.

placed on place fh. Place scopeCompensated is marked when a child scope signals its successful compensation. For each child scope (here C and D) there exist two places: push_scopeName and !push_scopeName. For example, there is a token on push_C, if `scope` C is completed faultlessly. Otherwise, !push_C is marked. If there is neither a token on push_C nor on !push_C, scope C has been compensated. On the right of the frame, the places clean and cleaned are analogue to the places stop and stopped. They are used to remove the tokens in the pattern, if a fault occurs during the execution of the compensation handler. A token on place failed signals a fault. This place is the same as place fault_in in the stop pattern of the enclosed scope. The places on the bottom are used either to invoke the compensation handler of a child scope (compScope – the same place as ch_in in the child scope) or to signal that the compensation handler is finished. So, if the compensation handler is invoked by a token on place ch_in (fh) a token is produced on place compensated (ch_out).

Before we explain the possible scenarios, we have a more detailed look at the invocation of the child scopes. This is done with help of transitions t13 and t14. If the child scopes should be compensated, place p11 is marked. For example, to invoke the child scope C (i.e. firing t13) means to invoke the compensation handler of C (token on place compScope) first and then to save the information that the scope has been invoked (token on place !push_C). Place compScope and places ch_in in all child scopes are joined. So we have to ensure that the signal to invoke scope C is assigned to C. For that purpose, the token and so the variable s holds the name of the scope, i.e. C. Now, let us have a look at what happens in the child scope C. It is sufficient to explain this behaviour by means of Fig. 42. If place compScope is marked, place ch_in in C's compensation handler is marked, too (because both places are identical). Looking at the post-set of place ch_in, it can be seen that every transition can only fire when its guard (scope=s) holds. Here the variable scope holds the name of the enclosed scope, i.e. C and s holds the name of the scope to be compensated. If C is compensated, a token holding its name is produced on place compensated which is the same as place scopeCompensated in the compensation handler of the parent scope. Now we can go back to the parent scope of C. Place scopeCompensated is marked. The token is an object that contains the name of the compensated child scope, i.e. C. So variable compScopeName holds the value "C". Depending on if the compensation handler was invoked by a token on ch_in or fh, either t10 or t9 fires and the next child scope, i.e. D can be invoked until all child scopes are compensated.

There are five possibilities to invoke B's compensation handler:

1. Scope A encloses a user defined compensation handler H. H's inner activity embeds activity <compensate/>.
2. Scope A encloses a user defined compensation handler H. H's inner activity embeds activity <compensate scope="B">.
3. Scope A encloses an implicit compensation handler.
4. Scope A encloses a user defined fault handler F. F's catch branch embeds activity <compensate scope="B">.
5. Scope B encloses a user defined fault handler F. F's catch branch embeds activity <compensate/> which uses the compensation handler to invoke the compensation handlers of all child scopes.

For each of the five possibilities we will now present a scenario in BPEL and its realization in our pattern.

**Scenario1:** Looking at the first possibility, three cases have to be distinguished: If B is completed faultlessly the `compensation handlers` of B's child scopes have to be invoked in the reverse order of completion of the corresponding `scopes` (1a). Otherwise, activity `empty` is executed (1b). When B has already been compensated, BPEL's standard fault `repeatedCompensation` is thrown (1c).
**Realization1a:** Place ch_in is marked and evaluating the transition guard scope=s guarantees that only the `compensation handler` of one child scope can be activated at the same time (see the above for an explanation). As we assumed above, B is completed and it has not been compensated; thus it is in states !Compensated and Completed. The transition guard holds and t3 fires (in our case both variables, s and scope, hold the scope name B). Now, the `compensation handler` is activated and therefore B changes into state Compensated. The `compensation handlers` of both scopes, C and D, are invoked by firing t13 and t14 in any order (for detailed description see above). Afterwards t7 is enabled, variable scopeName holds the name of `scope` B and the `compensation handler` is finished by firing t7. This is signalled to `scope` A by a token on compensated. This token is an object that consists of the the name of `scope` B. That place compensated and the place of the same name in A's activity <compensate/> are identical.
**Realization1b:** The Scenario1b has not been taken into account, because A's activity <compensate/> takes the name of the `scope` to be compensated from the respective push place. This place is only marked, if the `scope` is completed faultlessly.
**Realization1c:** This scenario only happens, if the corresponding BPEL process is designed incorrectly. The designer has to ensure that every `scope` is compensated only once. After the first compensation, `scope` B is in state Completed and Compensated. If the handler is invoked again, t1 fires and a fault is thrown. With it variable fault holds the name of the fault – `repeatedCompensation`. Furthermore the occurrence of this fault results in a token on p1.

**Scenario2:** This scenario is analogue to Scenario1.
**Realization2b:** B's `compensation handler` is invoked directly, i.e. the name of B is not taken from place push_B in A's `compensation handler`. Therefore it is possible that B is not completed, i.e. it is in state !Completed and !Compensated. So t2 fires. In our model we do not execute activity `empty`. Instead, variable scopeName holds the name of the `scope` B which is produced on place compensated.

**Szenario3:** This scenario is analogue to Scenario1.

**Szenario4:** This scenario is analogue to Scenario1.

**Scenario5:** B's `fault handler` invokes all child scopes in the reverse order of completion of the corresponding `scopes`. In the `compensation handler` of the corresponding `scopes` we have to distinguish the same cases as described in Scenario1.
**Realization5:** In order to get access of the child scopes that are to be compensated, activity <compensate/> invokes B's `compensation handler`. This results in a token on place fh. After firing t5, the push places can be accessed. The `compensation handlers` of the child scopes are invoked as described in Scenario1. There are only two differences:

t9 fires (instead of t10) if a child scope signals its compensation. Furthermore, when all child scopes are compensated, t8 is fired (instead of t7) placing a token on place ch_out. Place ch_out and the place of the same name in the <compensate/> activity inside B's fault handler are identical.

Similar to the stop component in the patterns of the activities, the pattern of BPEL's implicit compensation handler embeds a subnet to remove the tokens. These places are clean and cleaned. They are not labelled by stop and stopped, because there are scenarios where it is not possible to remove all tokens (see Sec. 10.4.2 for more details). The control flow, however, can be finished in any case. This can be ensured by removing the tokens on places fh_call and ch_call, respectively. If these tokens are removed, neither t7, t8, t13 nor t14 can be activated anymore.

## 10.4 User Defined Compensation Handler

Looking at the user defined compensation handler we have to distinguish three different kinds of handlers: the compensation handler that embeds activity <compensate/> (Sec. 10.4.1), the compensation handler that embeds at least one activity <compensate scope="name"> (Sec. 10.4.2), and the compensation handler that embeds any activity except activity compensate (Sec. 10.4.3). The BPEL specification does not allow to embed both kinds of activity compensate, because <compensate/> invokes all child scopes and therefore activity <compensate scope="name"> would invoke one child scope twice.

### 10.4.1 User Defined Compensation Handler Embeds <compensate/>

Figure 43 depicts the pattern of the user defined compensation handler that embeds activity <compensate/>. For purposes of simplification, compensate is the only inner activity of the handler and it is not embedded in another activity. For the reader it is only important to see how the handler and activity compensate are connected. The interface and most of the pattern's structure is the same as in the pattern of the implicit compensation handler in Fig. 42. If t4 fires, a token is produced on the initial place of activity compensate instead of place comp. To bring out the differences to Fig. 42, we take a look at the possible scenarios of Fig. 43. We distinguish the same scenarios as described in Sec. 10.3. Only Scenario1 differs.

**Scenario1a:** Scope B is completed when its compensation handler is invoked. So the handler activates its inner activity – <compensate/>. This activity runs all available compensation handlers for the immediately enclosed scopes in the reverse order of completion of the corresponding scopes. The effect is the same as described in Scenario1a in Sec. 10.3.

**Realization1a:** Place ch_in is marked and firing t4 starts the handler. The token on place ch_call identifies the handler to be invoked via the place ch_in. Next, activity <compensate/> is executed. Places p11 and comp are identical. So the pattern reaches the same marking (ch_call and comp) as in Fig. 42. Therefore the invocation of the child scope's compensation handlers is analogue to the one depicted for the implicit

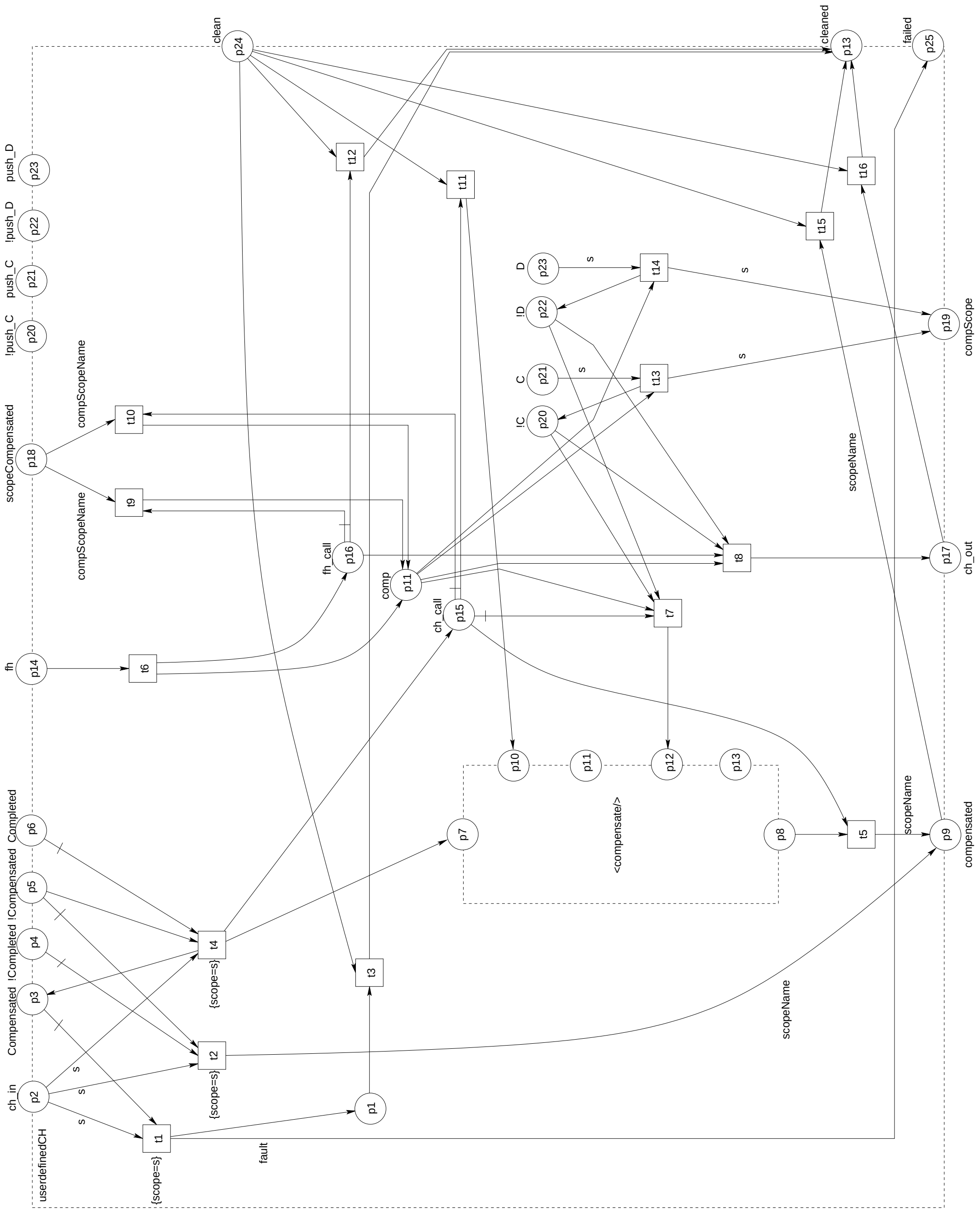


Figure 43: Pattern of the `compensation handler` with <compensate/>.

`compensation handler` in Fig. 42. If all child scopes were invoked, t7 can be fired and it produces a token on place p12. The token on place ch_call is read only and is not removed until the inner activity is completely executed. By firing t5 the name of the enclosing `scope` (here B) is saved in variable scopeName. The value of this variable is produced on place compensated.

In addition to the concept of removing the tokens by use of clean/cleaned presented in Fig. 42, we have to remove the tokens of the inner activity, too. Hence, firing t11 produces a token on place p10, the inner activity's stop place. The inner activity's stop procedure is started and results in a token on the stopped place of the inner activity which is the same place as cleaned.

### 10.4.2 User Defined Compensation Handler Embeds <compensate scope="C">

The pattern in Fig. 44 presents the user defined `compensation handler` that embeds the `compensate` activity <compensate scope="C">. For simplification, we decided to that `compensate` be the only activity. The interface and the main structure of the pattern are already known from the former section. There are only two differences to Fig. 43: Firstly, if the places ch_call and scopeCompensated are marked, i.e. child scope C signals that it is compensated successfully, there is no token produced on place comp. Secondly, places scopeCompensated and compScope, respectively and its corresponding places in activity <compensate scope="C"> are identical. These changes are caused by the fact that activity <compensate scope="C"> needs no access to the places p20 – p23. But these four places can be used by B's `fault handler`, if it embeds activity <compensate/>.

The scenarios are the same as described in Sec. 10.4.1. Only Realization1a of Scenario1 differs.

**Realization1a:** After t4 has fired, the token on place ch_call identifies the handler to be invoked via the place ch_in. Next, activity <compensate scope="C"> is executed. If there is a token on p19, C's `compensation handler` is invoked directly. Furthermore, the response of C is signalled directly to activity <compensate scope="C"> (via place p12).

In the following we draw a scenario where it is not always possible to remove all tokens in the pattern. Consider activities <compensate scope="C"> and <compensate scope="D"> embedded in a `flow`, invoking the `compensation handlers` of `scopes` C and D concurrently. Furthermore, let the `compensation handler` of C throw a fault that is rethrown to B's `fault handler` while D's `compensation handler` is still running. Via the stop pattern of `scope` B a token is produced on place clean in B's `compensation handler`. As a result, both `compensate` activities are stopped, but D's `compensation handler` is still running. When D's `compensation handler` is completed, a token is produced on scopeCompensated in B's `compensation handler`. This token cannot be removed anymore. In fact, we could extend our `compensation handler` pattern to avoid this scenario, but in the current version it is not.

### 10.4.3 User Defined Compensation Handler Embeds no Compensate

If the `compensation handler` only embeds an activity that is no `compensate`, we only need a simple modification of the implicit compensation handler pattern presented in

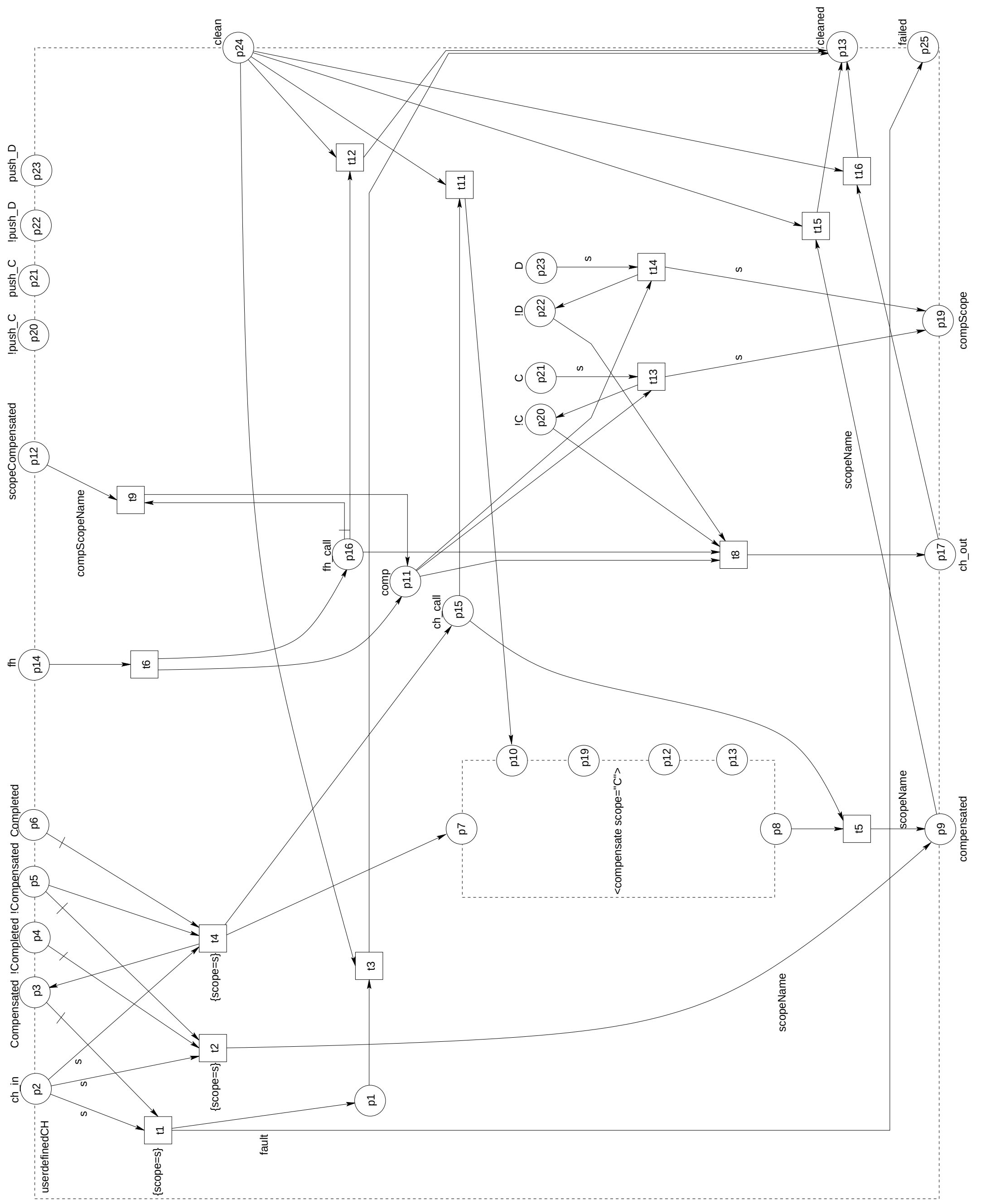


Figure 44: Pattern of the `compensation handler` with <compensate scope="C">.

Fig. 42 in Sec. 10.3. The resulting pattern is depicted in Fig. 45. In contrast to the pattern of the implicit `compensation handler`, the post-set of `t4` only contains the initial place of the inner activity. So the place ch_call and its post-set (t7, t10, t11) are not needed in the pattern. To remove the tokens of the <innerActivity> in the case of a fault, the places clean and stop and cleaned, and stopped are identical. Looking at the possible scenarios only Scenario1a differs:

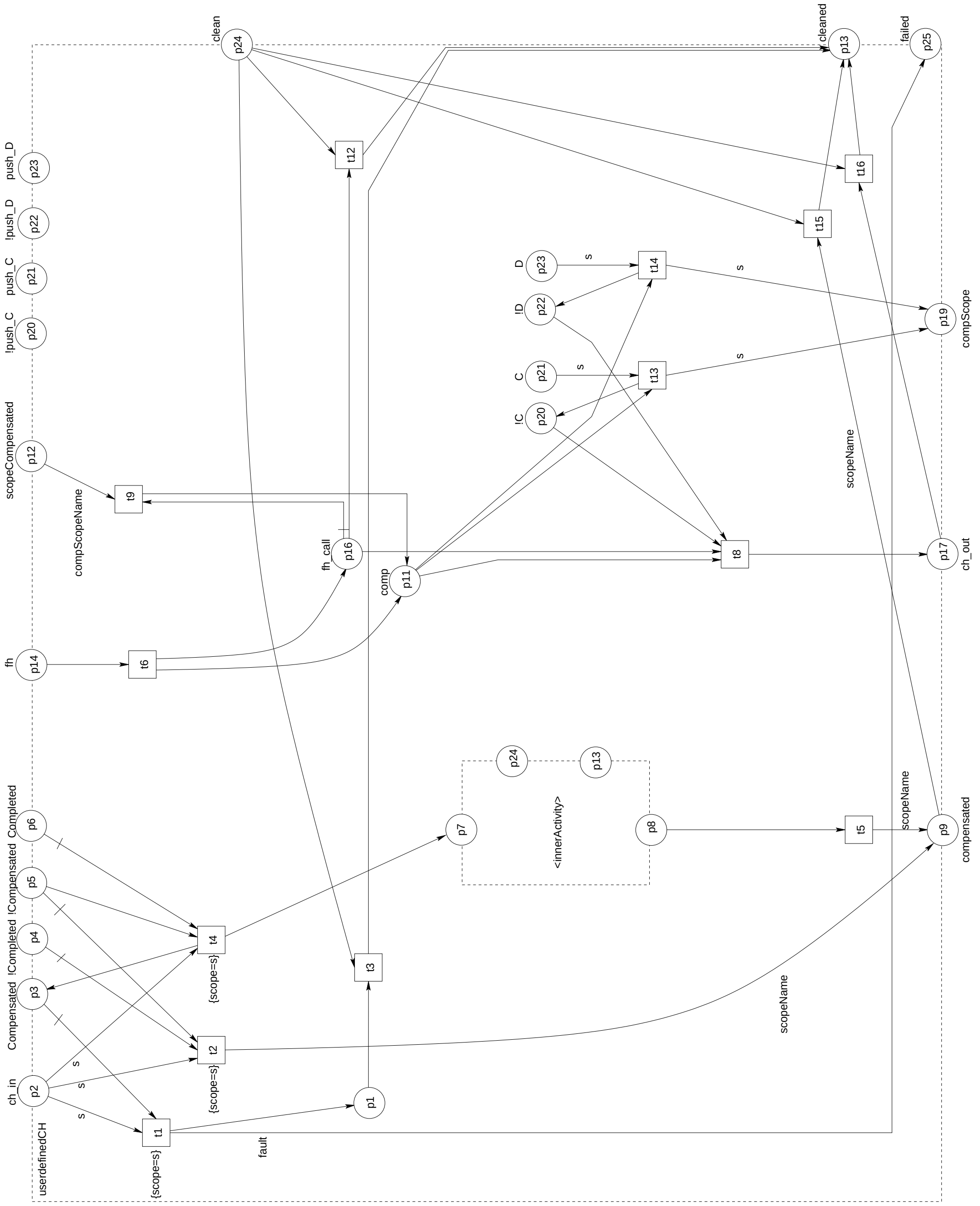


Figure 45: Pattern of the `compensation handler` without activity `compensate`.

**Scenario1a:** `Scope` B is completed when its `compensation handler` is invoked. So the handler activates its inner activity which is executed. Afterwards the handler is finished. **Realization1a:** Firing t4 activates the inner activity. After it is executed, the name of the enclosing scope (here B), stored in variable scopeName, is produced on place compensated.

# 11 Conclusions

In the last sections, we presented our approach of a formal semantics for BPEL based on Petri nets. Now, we want to summarize the properties of the Petri net patterns and classify the net type (Section 11.1). Afterwards, in Section 11.2, we give a short summary of our results and all modelling decisions we have made. The section closes with a preview of further work.

## 11.1 Classification of the Patterns

Let $P$ be a BPEL process transformed into a Petri net $N$. $N$ is an algebraic high-level Petri net, because in our patterns we model data: variables, `correlation sets`, the clock, and fault information. We distinguish three different arc types in $N$: *standard arcs*, *read arcs* and *reset arcs*. Because of the reset arcs $N$ is of type *Reset Net*. A reset arc is a very expressive construct. Thus, in a Reset Net less properties are decidable than in a net without such arcs [DFS98]. In the following, we will show how read arcs as well as reset arcs can be *unfolded* into a semantically equivalent low-level construct.

In our firing rule only one transition can fire at once. Therefore, a read arc is a short notation for a loop.

The reset arc is used in the stop pattern to remove all tokens from place fault_in. In the following, we draft the idea how such an arc can be modelled as a high-level construct which can be, in turn, unfolded into a low-level construct: It is possible to safely over-approximate the maximal number $k$ of tokens, i.e. the number of faults that can be produced on place fault_in. This is the number of activities of the enclosing scope that can throw a fault. Every scope encloses only a finite number of activities. Consequently $k$ is bounded. So place fault_in is a high-level place that is *k-bounded*, i.e. the number of tokens on fault_in is never greater than $k$. Then, unfolding the reset arc means to replace fault_in by $k+1$ places (0 tokens are possible, too). Furthermore, every transition of the pre-set or post-set of fault_in has to be replaced by $k+1$ transitions. It can be seen easily that a reset arc causes an increase of the net size. The value of $k$ can be narrowed, for instance, in the case of a sequence. Unaffected by the number of its inner activities only one fault can be thrown, because after this fault is thrown the control flow within the sequence is blocked. Calculating the best possible $k$ of place fault_in is ongoing research.

To drive the conclusion, $N$ can be unfolded into a high-level Petri net, more detailed an algebraic high-level Petri net that only contains standard arcs and data.

## 11.2 Results

In this paper, we presented a feature-complete Petri net semantics for BPEL. Our Petri net semantics consists of patterns: For each BPEL construct a pattern exists. The semantics has the following properties:

- Only one instance of a BPEL process can be transformed into a Petri net.

- The semantics abstracts from the connection of a BPEL process to its partner processes. The interface of a BPEL process is transformed into a set of message channels, i.e. places in the Petri net.

- In our Petri net patterns we model data, but we abstract from the definition of the functions which edit the data. Furthermore, we did not specify the transition guards and so we did not specify which circumstances are necessary that a specific fault can occur.

- Every activity is limited to one `correlation set` (except the synchronous `invoke` which is limited to two `correlation sets`).

To translate specific concepts of BPEL into our Petri net semantics we made the following design decisions:

- In order to stop the positive control flow, e.g. when a fault occurs or an activity `terminate` becomes active, we extended every activity's pattern by a stop component. Such a stop component can remove the tokens of the respective pattern. We furthermore extended the scope pattern by a so-called stop pattern which has no equivalent construct in BPEL. If a `scope` needs to be stopped, the stop pattern controls this procedure. In [Sta04] we proved that every `scope` and thus every `process` can be stopped using stop components.

- Modelling the `compensation handler` we met another problem: An implicit `compensation handler` has to invoke all `compensation handlers` of its child scopes in the reverse order of their execution. Thus, it was necessary to save all executed `scopes`. For that purpose, we made the following design decision: If a `scope` B is executed faultlessly, a token is produced on place push_B in the `compensation handler` of B's parent scope A. If the `compensation handler` of A is activated, it invokes the `compensation handler` of all child scopes which have already been executed, i.e. their respective push_scopeName place is marked. The order of invocation is chosen nondeterministically, because nondeterminism covers all possible orders.

In addition, we also detected a deadlock scenario in BPEL (see Section 9.2 for details).

## 11.3 Further Work

We have presented a translation from BPEL to Petri nets. This translation follows a feature-complete Petri net semantics. Thus, we can transform every BPEL process into a Petri net. Obviously, it is very laborious and takes too much time to generate the Petri net of a BPEL process manually. Therefore tool support was necessary to transform a BPEL process automatically into a Petri net.

Our tool, the parser *BPEL2PN* [Hin05], takes an BPEL process as an input. This process is transformed into a Petri net according to the Petri net semantics. The output of the tool is a Petri net in the data format of our model checker *LoLA* [Sch00]. In the current version BPEL2PN is limited to low-level Petri nets. That means, we decided to abstract from data, i.e. messages and data are modelled as black tokens, because we directed our attention to the control flow. Consequently all other high-level constructs like transition guards and variables were left out, too. So selecting one of two control pathes in the Petri net semantics, solved by the evaluation of data, is modelled by a nondeterministic choice, e.g. t2 or t3 in Figure 1.

At the moment our tool LoLA can analyze Petri nets where the respective BPEL process consists of 50 and more activities. Such a process embeds nested scopes. By the use of LoLA we are able to verify every temporal property of a transformed BPEL process. First results are presented in [SS04].

The models generated by the present version of our parser can be seen as brute force models. The generated models are significantly larger than typical manually generated models. This is due to the fact that the Petri net patterns are complete, i.e. applicable in every context. For a particular process, many of the modelled features are unused. For instance, if a basic activity cannot throw any error, many of the error handling mechanisms in the surrounding compound activity can be left out.

In ongoing projects, we aim at an improved translation where several Petri net patterns with different degree of abstraction are available for each BPEL activity. Using static analysis on the BPEL code, we want to select the most abstract pattern applicable in a given context. We believe that model sizes can be drastically removed. This way we can alleviate the state explosion problem inherent to model checking.

Our goal is a technology chain that, starting at a BPEL process, performs static analysis. Based on the analyzed information, the translator selects the most abstract pattern for each activity that is feasible in the analyzed context and synthesizes a Petri net model. On the Petri net model, a model checker evaluates relevant properties. The analysis results (e.g., counter example paths) are translated back to the BPEL source code.

## Acknowledgements

This work is based on the results of my diploma thesis. It would not have been possible without the help I received from the following people:

I would like to thank Axel Martens, Wolfgang Reisig, and Michael Weber who supervised my diploma thesis. They all took much time to listen to my problems and to give me support and useful hints.

Additional thanks go to Dirk Fahland, Carsten Frenkler, and Peter Massuthe for long and helpful discussions about BPEL.

Many thanks I would like to address to Daniela Weinberg for her careful reading of this document and her constructive suggestions for improvement.

I also thank all other colleagues of the “Kaffeerunde” for their discussions and hints concerning my work and for the pleasant and familiar atmosphere at the chair.

## References


[CGK+03] Curbera, Goland, Klein, Leymann, Roller, Thatte, and Weerawarana. Business Process Execution Language for Web Services, Version 1.1. Technical report, BEA Systems, International Business Machines Corporation, Microsoft Corporation, May 2003.

[DFS98] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In K. Spies and B. Schätz, editors, *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, Lecture Notes in Computer Science 1443, pages 103–115. Springer, 1998.

[Fah05] Dirk Fahland. Complete Abstract Operational Semantics for the Web Service Business Process Execution Language. Technical Report 190, Humboldt-Universität zu Berlin, June 2005.

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting BPEL web services. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[Fer04] Andrea Ferrara. Web services: a process algebra approach. In *ICSOC*, pages 242–251. ACM, 2004.

[FFK04] Jesús Arias Fisteus, Luis Sánchez Fernández, and Calos Delgado Kloos. Formal Verification of BPEL4WS Business Collaborations. In *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, LNCS. Springer, August 2004.

[FGV04] Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In *Abstract State Machines*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2004.

[FR05] Dirk Fahland and Wolfgang Reisig. ASM-based semantics for BPEL: The negative Control Flow. In E. Börger D. Beauquier and A. Slissenko, editors, *Proc. 12th International Workshop on Abstract State Machines, Paris, March 2005*, Lecture Notes in Computer Science. Springer-Verlag, March to appear, 2005.

[Hin05] Sebastian Hinz. Implementation einer Petrinetz-Semantik für BPEL4WS. Diplomarbeit, Humboldt-Universität zu Berlin, 2005.

[Ley01] Frank Leymann. *WSFL – Web Services Flow Language.* IBM Software Group, Whitepaper, May 2001. `http://ibm.com/webservices/pdf/WSFL.pdf`.

[LR99] Frank Leymann and Dieter Roller. *Production Workflow – Concepts and Techniques.* Prentice Hall, 1999.

[Mar04] Axel Martens. *Verteilte Geschäftsprozesse – Modellierung und Verifikation mit Hilfe von Web Services.* Dissertation, WiKu-Verlag Stuttgart, 2004.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.

[RWL+03] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank Qvortrup, Martin Stig Stissing, Michael Westergaard, Søren Christensen, and Kurt Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN 2003), Eindhoven, The Netherlands, June 23-27, 2003 — Volume 2679 of Lecture Notes in Computer Science / Wil M. P. van der Aalst and Eike Best (Eds.)*, pages 450–462. Springer-Verlag, June 2003.

[Sch00] Karsten Schmidt. LoLA – A Low Level Analyser. In Nielsen, M. and Simpson, D., editors, *International Conference on Application and Theory of Petri Nets*, LNCS 1825, page 465 ff. Springer-Verlag, 2000.

[Sch04] Karsten Schmidt. Controlability of Business Processes. Technical Report 180, Humboldt-Universität zu Berlin, December 2004.

[SR00] Peter H. Starke and Stephan Roch. Ina et al. In Kjeld H. Mortensen, editor, *Tool Demonstrations 21st International Conference on Application and Theory of Petri Nets*, pages 51–56. Department of Computer Science, University of Aarhus, june 2000.

[SS04] Karsten Schmidt and Christian Stahl. A Petri net semantic for BPEL4WS - validation and application. In Ekkart Kindler, editor, *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, pages 1–6. Universität Paderborn, october 2004.

[Sta04] Christian Stahl. Transformation von BPEL4WS in Petrinetze. Diplomarbeit, Humboldt-Universität zu Berlin, April 2004.

[Tha01] Satish Thatte. *XLANG – Web Services for Business Process Design.* Microsoft Corporation, Initial Public Draft, May 2001. `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c`.

[vdA98] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[Web03] Michael Weber. *Allgemeine Konzepte zur software-technischen Unterstützung verschiedener Petrinetz-Typen.* PhD thesis, Humboldt-Universität zu Berlin, 2003. URL `http://dochost.rz.hu-berlin.de/dissertationen/weber-michael-2002-12-16/PDF/Weber.pdf`.