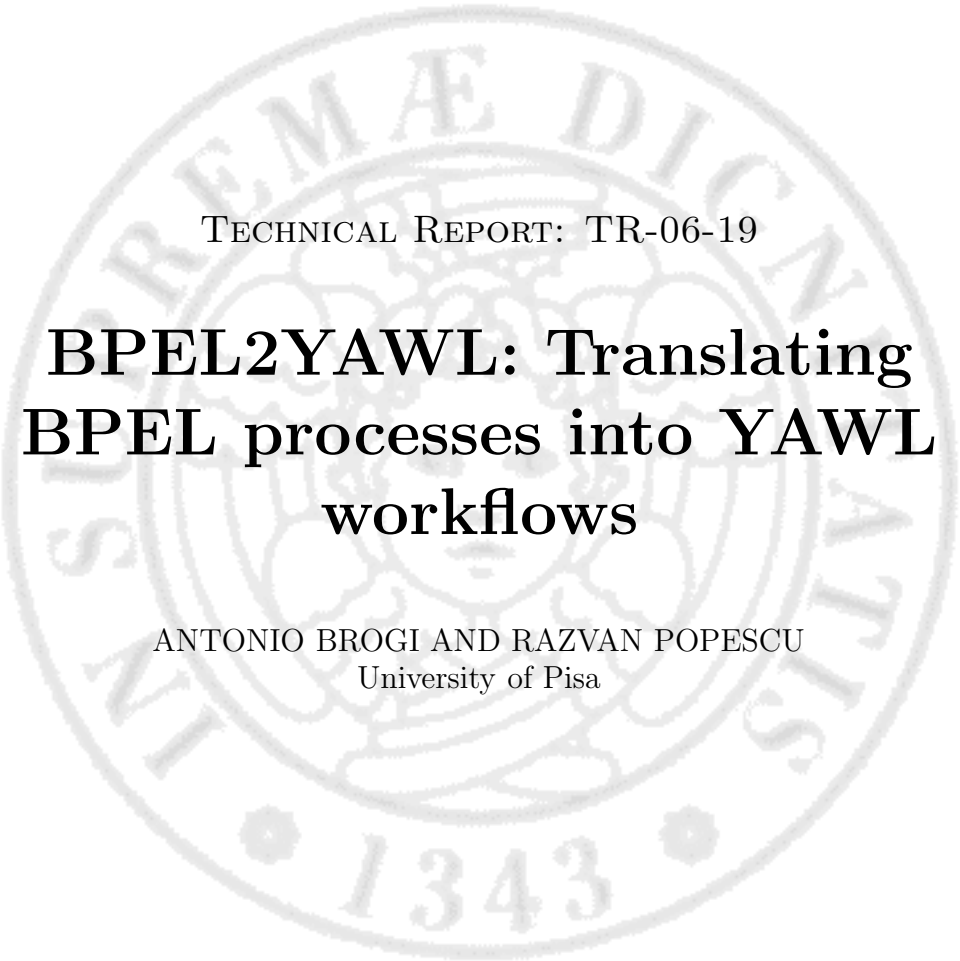
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-06-19

BPEL2YAWL: Translating BPEL processes into YAWL workflows

ANTONIO BROGI AND RAZVAN POPESCU
University of Pisa

December 21, 2006

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

BPEL2YAWL: Translating BPEL processes into YAWL workflows

ANTONIO BROGI AND RAZVAN POPESCU
University of Pisa

December 21, 2006

Abstract

The availability of different languages for the description of Web service behaviour hinders automated Web service aggregation, discovery, and adaptation, as currently there are no available tools for the automated translation of service protocols.

In this paper we motivate the choice of YAWL as a *lingua-franca* to express the interaction behaviour of Web services. Furthermore, we provide the specification of a translator of BPEL processes into YAWL workflows, thus paving the way for the formal analysis, aggregation, discovery, and adaptation of BPEL processes. In short, the specification defines a YAWL pattern for each BPEL activity. The translation of a BPEL process reduces then to suitably instantiating and interconnecting the patterns of its activities.

1 Introduction

Service-oriented computing [14] is emerging as a new promising computing paradigm that centres on the notion of *service* as the fundamental element for developing future distributed heterogeneous software applications. The W3C defines a Web service as "a software system designed to support interoperable machine-to-machine interaction over a network" [22].

Web services[1], similarly to software components, wrap applications and expose interfaces to be used by clients for invocation. The current standard to express Web service interfaces is WSDL [24], which defines the operations supported by the Web service, but not the order in which they should be invoked. As a consequence, service compositions may lock during their interaction.

Several proposals [2, 13, 21, 23] aim at providing languages to express Web service behaviour (viz., protocol information). Still, most of them lack formal semantics, and this hinders the formal verification of Web services' properties, such as lock-freedom. Another open problem is that behaviour-aware Web service discovery, aggregation, and adaptation cannot be automatically employed

[1] We shall use "service" and "Web service" interchangeably throughout the paper.

to create heterogeneous Web services. Indeed, the current lack of a standard to describe service behaviour allows heterogeneous descriptions, but tools for the automated translation of service protocols are not yet available.

In order to cope with the above issues, we argue for the usage of an abstract formal language as a *lingua-franca* for expressing the behaviour of Web services. The advantages one obtains are the possibility to formally analyse services, to discover, aggregate, and adapt services whose interaction protocols are described using different languages, as well as the possibility to translate service protocols from one language into another.

We argue that YAWL [17] is a good candidate for expressing the service behaviour. YAWL is a new proposal of a workflow/business processing system that supports a concise and powerful workflow language and handles complex data transformations and Web service integration. YAWL is an abstract language with a well-defined formal semantics, which implements twenty of the most common workflow patterns.

On the one hand, YAWL provides a formal basis for the analysis of Web service protocols. Being built on Petri nets, YAWL is an easy to understand and to use formalism, which features an intuitive (graphical) representation of services. Although it is a relatively recent language, YAWL analysis tools are starting to emerge (e.g., [19]). Moreover, YAWL can also benefit from the abundance of Petri net analysis techniques (e.g., [20]).

On the other hand, YAWL can be used as an intermediate language for expressing the service behaviour. In this way, one may translate e.g., OWL-S [13] service models into YAWL, and then deploy them e.g., as BPEL processes. Another advantage is that YAWL can be used as a basis for the automation of the discovery [3], aggregation [3, 4, 6], and adaptation [5, 7] of services (possibly presented with different service description languages). Moreover, one may check the YAWL workflow corresponding to a service composition against properties such as lock-freedom, reachability, liveness, and so on [3].

In this paper we present the specification of a translator of BPEL processes into YAWL workflows (BPEL2YAWL, for short). We chose BPEL since it is currently the most widely adopted approach for expressing the behaviour of Web services. BPEL describes business processes through the specification of control and data logic around a set of (WSDL) Web service interactions. Roughly, a BPEL process is constructed by wrapping basic activities into structured ones. The basic activities are used, for example, to exchange messages among the services involved in the business process, to delay the execution of the process, or to signal faults. The control-flow in BPEL is achieved, on the one hand, through structured activities such as sequences and switches, and on the other hand, through the use of links to synchronise activities executed in parallel. It is important to note that the semantics of activity execution in BPEL is not straightforward, mainly due to the synchronisation links and to the use of scopes, which wrap activities and provide them with event, fault, and compensation handlers. Our work is also motivated by the fact that most approaches that attempt to provide a formal (e.g., Petri net) semantics to BPEL processes [1, 9, 10] do not tackle BPEL synchronisation links and/or the exceptional behaviour

of a business process.

We present a compositional translation based on YAWL patterns. Basically, we define a YAWL pattern for each BPEL activity, as well as for the whole BPEL process. In more detail, we define a Basic Pattern Template (BPT) and a Structure Pattern Template (SPT) to translate basic and structured activities, respectively. The role of patterns is twofold – they provide a unique representation of activities, and they provide an execution context for them.

Given a BPEL process, the BPEL2YAWL translator automatically generates its YAWL translation by:

- Instantiating the pattern of each activity defined in the BPEL process, and by
- Suitably interconnecting the obtained patterns into the final workflow.

Patterns are linked using three types of lines: *green* lines – to represent the structural dependencies among activities, *blue* lines – to translate the synchronisation dependencies, as well as *red* lines – for the propagation of faults toward fault handlers.

To the best of our knowledge, our translator is the first attempt to translate BPEL processes into YAWL workflows. Its main features can be summarised as follows:

- It provides an automated pattern-based compositional translation of BPEL processes into YAWL workflows,
- It copes with all types of BPEL activities (including *flow*s with synchronisation links, and *scope*s),
- It handles exceptional behaviour – events, faults and (explicit) compensation,
- It can be straightforwardly plugged into our Web service discovery [3], aggregation [3, 4, 6], and adaptation [5, 7] methodologies,
- The patterns defined by the BPEL2YAWL translator provide the basis for the definition of an inverse YAWL2BPEL translator, which becomes straightforward, and
- It sets the basis for the formal analysis of BPEL processes.

We also argue that the present paper complements [2] by providing a "lightweight" semantics of BPEL processes in terms of YAWL workflows. As we will show in Section 3, almost all BPEL activities are provided with simple intuitive translations in terms of (YAWL) workflows. As a consequence, the description of the translation also provides an intuitive description of BPEL features.

Section 2 briefly introduces BPEL and YAWL. Section 3 is devoted to the specification of the BPEL2YAWL translator. Subsections 3.1 and 3.2 define the

patterns used for translating the BPEL basic and structured activities, respectively, while Subsection 3.3 describes the translation of BPEL processes. Section 4 thoroughly presents a simple translation example, while some concluding remarks are drawn in Section 5.

2 A Brief Introduction to BPEL and YAWL

The next two Subsections give a high-level view of both languages. More details on the two languages will be discussed in the next Section, while describing the translation methodology. A thorough description of the two languages can be found in [2] for BPEL and in [17] for YAWL.

2.1 BPEL: Business Process Execution Language

BPEL [2] is a language for expressing the behaviour of a business process through the specification of control and data logic around a set of Web service interactions. Basically, a BPEL process orchestrates the operations offered by the partner Web services through WSDL [24] interfaces, and in turn, it exposes a WSDL interface to clients.

A BPEL process can be either abstract, or executable. Abstract processes hide implementation details (i.e., private information), while executable processes describe the full interaction behaviour.

BPEL defines the notion of *partner link* to model the interaction between a business process and its partners. A partner link refers to at most two WSDL *port type*s, one of the interface to the business process (viz., operations offered by the process to the partner), and the other belonging to the interface of a partner (viz., operations offered by the partner to the business process).

BPEL is a hybrid language that combines features from both the block-structured language XLANG [15] and from the graph-based language WSFL [25]. The former contributed with basic activities (e.g., for sending and receiving messages, for waiting for a period of time) as well as with structured ones (e.g., sequential or parallel execution of activities, activity scoping) for combining activities into complex ones. The latter brought the definition of links to synchronise activities executed in parallel. Roughly, the execution of an activity that is the target of synchronisation links is delayed until all activities from where the links emerge are executed. Other features of BPEL are the instance management through correlation sets, event and fault handling, as well as compensation capabilities. The correlation sets are used to identify the various sessions that a business process can have with its clients. Event, fault and compensation handlers make the exceptional behaviour of a business process. Event handlers define message and alarm events, while fault handlers catch and process faults raised in the process. Furthermore, compensation handlers provide roll-back activities to compensate for faults in the process. More details on these topics will be given in Sections 3 and 4.

The BPEL basic activities are: *receive*/*reply* through which a BPEL process inputs/sends a message from/to a partner service, *invoke* through which a BPEL process asynchronously/synchronously invokes an operation of a partner service, *wait* for delaying the execution of a process, *throw* for signalling faults, *terminate* for explicitly terminating the execution of a process, *empty* for doing a "no-op", *assign* for copying values between variables, and *compensate* for invoking compensation handlers.

The structured activities are: *sequence*, *switch*, and *while* for sequential, conditional and repeated activity execution, *flow* for parallel activity execution, *pick* for managing the non-deterministic choice of the activity to be executed, and *scope* for providing an execution context for an activity.

For example, consider the following simplified BPEL process (snippet) that computes the greatest common divisor (GCD) of two numbers.

```
<process name="GCD" suppressJoinFailure="yes">
    <faultHandler>
        <catch fault="negNum">
            <reply fault="negNum"/>
        </catch>
    </faultHandler>
    <flow>
        <receive(a,b) createInstance="yes">
            <source link="RCV2THR" transitionCondition="a<=0 or b<=0"/>
            <source link="RCV2WHL" transitionCondition="a>0 and b>0"/>
        </receive>
        <throw fault="negNum">
            <target link="RCV2THR"/>
        </throw>
        <while condition="a!=b">
            <source link="WHL2SEQ"/>
            <target link="RCV2WHL"/>
            <scope>
                <faultHandler>
                    <catch fault="dec_a">
                        <assign a:=a-b/>
                    </catch>
                    <catch fault="dec_b">
                        <assign b:=b-a/>
                    </catch>
                </faultHandler>
                <switch>
                    <case condition="a>b">
                        <throw fault="dec_a"/>
                    </case>
                    <otherwise>
                        <throw fault="dec_b"/>
                    </otherwise>
                </switch>
            </scope>
        </while>
        <sequence>
            <target link="WHL2SEQ"/>
            <assign c:=a/>
            <reply(c)/>
        </sequence>
    </flow>
</process>
```

The GCD process defines a *flow* activity, which consists of four activities: a *receive*, a *throw*, a *while*, and a *sequence*. Furthermore, the *flow* defines three

synchronisation links. The first two, *RCV2THR* and *RCV2WHL*, emerge at the *receive* activity and target the *throw* and the *while* activities, respectively. The third one, *WHL2SEQ* emerges at the *while* and targets the *sequence*. It is important to note that each BPEL activity that is the target of at least one synchronisation link has a (possibly default) *joinCondition* logical expression that computes the synchronisation status based on the statuses of the input links. Furthermore, the *suppressJoinFailure* attribute serves for deciding the control-flow in case of a *false joinCondition*. If the *suppressJoinFailure* is set to *yes*, the BPEL engine simply skips the respective activity in order to achieve the dead-path-elimination. Otherwise, the BPEL engine raises a *joinFailure* fault.

At run-time, the execution of the *flow* structurally enables its four child activities, yet only the *receive* can be executed first as the other three activities are constrained from the synchronisation viewpoint (i.e., the statuses of their input links are not known). The *receive* inputs two numbers, *a* and *b*. On the one hand, if both numbers are greater then zero, the BPEL engine sets a negative status for the *RCV2THR* link and a positive status for the *RCV2WHL* link. As a consequence, the *throw* is skipped, and since the *suppressJoinFailure* attribute is set to *yes* for the entire BPEL process, a *joinFailure* fault is not signalled. The activity to be executed next is the *while*, which checks whether *a* is equal to *b*. If this is not the case, the process continues with the execution of the *scope* activity inside the *while*. The *scope* further consists of a *switch*, with two branches. If *a* is greater than *b* a *dec_a* fault is raised. Otherwise, the BPEL engine raises a *dec_b* fault. Both faults are to be caught by the *faultHandler* of the *scope* activity. In the former case, *a* is decreased by *b* (in the first *assign* activity), while in the latter case *b* is decreased by *a*. At this point the *scope* terminates and the execution of the process continues by checking whether *a* is now equal to *b*. If so, a new *while* cycle is performed. Otherwise, the *while* terminates and BPEL sets a positive status for the *WHL2SEQ* link. Consequently, BPEL executes the *sequence* that first stores the value of *a* into a new variable *c* and then it sends it to the invoker of the GCD process through the *reply* activity.

On the other hand, if at least one of the two numbers is negative or zero, *RCV2THR* gets a positive status, while *RCV2WHL* a negative one. Consequently, the BPEL engine executes the *throw*, during which the *while* activity is skipped. The execution of the *throw* raises a *negNum* fault that is caught by the *fault handler* of the *process*, which forwards it to the invoker of the business process through the *reply* activity.

2.2 YAWL: Yet Another Workflow Language

YAWL [17] is a new proposal of a workflow/business processing system, which supports a concise and powerful workflow language and handles complex data transformations and Web service integration. YAWL defines twenty most used workflow patterns divided in six groups – basic control-flow, advanced branching and synchronisation, structural, multiple instances, state-based, and cancellation. A thorough description of these patterns may be found in [18].

YAWL extends Petri nets by introducing some workflow patterns (for multiple instances, complex synchronisations, and cancellation) that are not easily expressed using (high-level) Petri nets. Being built on Petri nets, YAWL is an easy to understand and to use formalism, which features an intuitive (graphical) representation of services. Moreover, it can benefit from the abundance of Petri net analysis techniques. With respect to other workflow languages (mostly proposed by industry), YAWL relies on a well-defined formal semantics based on transition systems. Moreover, not being a commercial language, YAWL supporting tools (editor, engine) are freely available.

From a control-flow perspective, a YAWL file describes a *workflow specification* that consists of a tree-like structure of *extended workflow net*s (or EWF-nets for short). An EWF-net is a graph where nodes are *task*s or *condition*s, and edges define the control-flow relation. Each EWF-net has a single *input condition* and a single *output condition*. Tasks employ one *join* and one *split* construct, which may be one of the following: AND, OR, XOR, or EMPTY. Intuitively, the join of a task T specifies "how many" tasks before T are to be terminated in order to execute T, while the split construct specifies "how many" tasks following T are to be executed. It is worth noting that YAWL tasks may be interpreted as Petri net *transitions*, and YAWL conditions can be represented as Petri net *places*. The control-flow for tasks with XOR/OR splits is specified through *predicates* in the form of logical expressions. When a task finishes its execution, it places tokens in its output places, according to its split type. Dually, a task is enabled for execution depending on its join and on the tokens available in its input places. Another feature of YAWL is the use of *cancellation set*s consisting of conditions and tasks. When a task is executed all tokens from its cancellation set (if any) are removed. Cancelation sets are useful, for example, to prevent tasks from being executed given some particular circumstances, or even to terminate the execution of the entire workflow. For example, if multiple tasks are used to book each one flight ticket with a different airline company, the first one to be executed successfully should inhibit the other ones, while the impossibility to book a flight ticket with any company should immediately terminate the workflow, without having to hire a car, for example.

Consider the YAWL composite task given in Figure 1. It consists of a EWF-net and it employs a XOR-join and an OR-split. The execution of the composite task places a token into the *input condition* of the EWF-net, which enables the *ExecOrSkip* (atomic) task. *ExecOrSkip* then inputs the *parentSkip* and *joinCondition* variables[2] and it computes the value of the *skip* variable. The XOR-split of the *ExecOrSkip* task decides next the control-flow of the EWF-net. If *skip* is *true*, a token is sent to the *ComputeTransitionConditions* task. Otherwise, the token is sent to the *ActivitySpecificTask*, whose execution places a token in its output condition. The *deferred choice* made by the respective YAWL condi-

[2]Note that the meaning of the respective variables is not mandatory for understanding the semantics of executing the workflow in the Figure. However, the workflow will be thoroughly explained in Subsection 3.1, when describing the Basic Pattern Template.

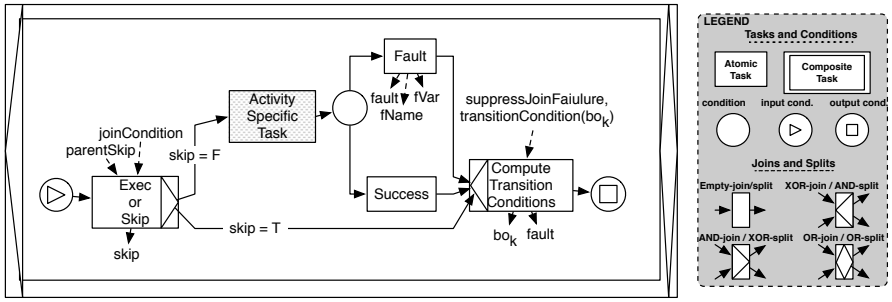


Figure 1: Example of a YAWL composite task.

tion together with the *Fault* and *Success* tasks corresponds to a non-determinitic choice in the EWF-net. In other words, the environment (viz., the invoker of the workflow) is in charge of selecting the task to be executed next. In both cases, the executed task forwards a token to *ComputeTransitionConditions*. Note that the *ComputeTransitionConditions* task is enabled by the reception of one token only, due to its XOR-join. Its execution marks the termination of the composite task by placing a token in the output condition of the EWF-net.

3 From BPEL to YAWL

The objective of this paper is to present a methodology for translating BPEL processes into YAWL workflows – with a special care to preserve the information in the BPEL processes so that the definition of an inverse YAWL2BPEL translator becomes straightforward. First, we define a YAWL pattern for each BPEL activity, as well as for the entire business process. Then, the workflow corresponding to a BPEL process is obtained by suitably instantiating and interconnecting the workflows of all its activities.

In Subsection 3.1 we first introduce the Basic Pattern Template, and then we uses it to define the patterns of the basic activities. Then, in Subsection 3.2 we define the Structured Pattern Template, which we use to define the patterns of the structured activities. Finally, in Subsection 3.3 we define the Process pattern template and describes the process of obtaining the final workflow.

In the following we shall use the term Pattern Template to refer to the pattern of a generic BPEL activity (viz., either basic or structured). The role of a pattern template is twofold: It provides the necessary elements for uniquely identifying an activity/process, as well as an execution context for the translated activity/process.

3.1 Patterns of BPEL basic activities

BPEL uses structured activities to specify the order in which activities have to be executed. For example, the second activity in a sequence can be executed

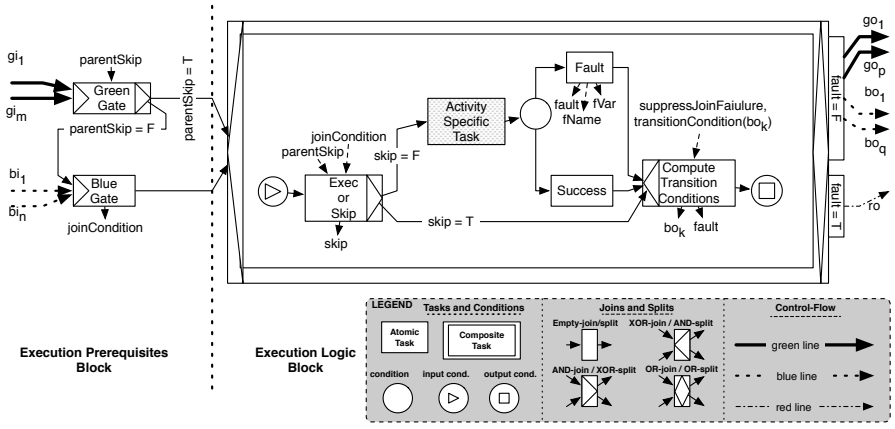


Figure 2: The Basic Pattern Template.

only when the first one has finished its execution. Moreover, the *flow* construct allows for synchronisation links to be defined among activities. As previously mentioned, when an activity is structurally enabled, BPEL waits for the statuses of all its incoming links (if any) to be determined. At that point BPEL computes the *joinCondition* (a logical expression), which guards the execution of the activity. A *true* value leads to the execution of the activity, while a *false* value leads to either raising a *joinFailure* fault, or to skipping the entire activity. It is important to note that a structured activity that is skipped leads to skipping all the activities nested within it. Skipping an activity leads to propagating negative (viz., false) statuses on its output links. This process is called *dead-path-elimination*.

We model the *structural relations* among BPEL activities through what we call *green lines*. A pattern has one or more green inputs, which are used to enable it from the structural point of view. Dually, it has one or more green outputs, to be sent upon completion of the pattern, which will be used to enable further patterns. For example, the patterns translating child activities of a BPEL *sequence* have to be linked through green lines. The pattern corresponding to the first activity in the sequence outputs a green line that is taken as input by the pattern of the second activity in the sequence (in lexical order, since this is the order of execution of the activities in the sequence). Then, the process of linking the patterns of the activities in the sequence through green lines is repeated until the last activity in the sequence. As we shall see in Subsection 3.2, the pattern of the first activity in the sequence inputs a green line from a special pattern that marks the beginning of the sequence pattern. Dually, the pattern of the last activity in the sequence outputs a green line to another special pattern that marks the end of the sequence pattern.

On the other hand, we model the *synchronisation links* among BPEL activities using *blue lines*. A pattern has one blue input for each synchronisation

link that targets the activity it translates. Analogously, it has one blue output for each link that emerges from the activity it translates. For example, inside a BPEL *flow*, a synchronisation link from activity A to activity B is translated into a blue line from the pattern translating A to the pattern translating B. Then, the pattern of A is in charge of computing the status of the respective link in a (global) variable, while the pattern of B first waits to receive a blue token on the respective link, and then it computes the value of the *joinCondition*.

Finally, in order to cope with *faults* we use *red lines*. Patterns that treat errors (viz., faults) have red inputs, while patterns that generate errors have red outputs. For example, the translation of the BPEL *throw* activity has a red line as output, while the translation of the BPEL *fault handler* inputs one.

The Basic Pattern Template is illustrated in Figure 2. It consists of an Execution Prerequisites Block and of an Execution Logic Block. Green input lines of a pattern are denoted by *gi*, and green outputs by *go*. Similarly, *bi* and *bo* denote blue inputs and outputs, and *ri* and *ro* red ones.

The Execution Prerequisites Block (EPB). The EPB is in charge of enabling the pattern. In order to execute, a pattern has to be enabled both from the structural and from the synchronisation point of view.

The *GreenGate* task of the EPB is in charge of waiting for the green tokens. It also inputs a *parentSkip* boolean variable from its parent[3] activity, whose value indicates whether the latter has been skipped or not. Indeed, since each structured activity could be skipped if it is the target of a synchronisation link, it outputs a *parentSkip* variable to all the patterns corresponding to its nested (child) activities. If *parentSkip* holds *true* then the pattern must be skipped, as one of its ancestors was skipped. In this case *GreenGate* will immediately enable the Execution Logic Block, without having to wait for the statuses of its incoming links to be computed. If instead *parentSkip* holds *false*, then the pattern is ready to be executed from the structural viewpoint. Consequently, the execution of the EPB continues with the *BlueGate* task, which waits for all blue tokens and then it computes the value of the *joinCondition* by taking into account the statuses of its incoming links stored into *bi* boolean variables.[4] Then, the *BlueGate* enables the Execution Logic Block.

The Execution Logic Block (ELB). The ELB has three possible execution scenarios: It can execute successfully, it can be silently skipped, or it can raise a fault. While the first and the second case correspond to executing and skipping, respectively, the pattern, the third behaviour corresponds to a false *joinCondition* (see next) or to an erroneous execution of the activity.

[3] When an activity A is directly nested within a structured activity S, we also say that S is the *parent* of A and that A is a *child* of S.

[4] Note that BPEL uses (possibly default) *transitionCondition*s for the synchronisation links, as well as (possibly default) *joinCondition*s for each activity that is the target of at least one synchronisation link. As a consequence, the *joinCondition* defined by a *BlueGate* task corresponds to the BPEL *joinCondition* except that the statuses of the synchronisation links are replaced by corresponding *bi* variables.

The *ExecOrSkip* task of the ELB computes the skipping condition (into the *skip* boolean variable) as a logical disjunction between the *parentSkip* and the negation of the *joinCondition* variables. Indeed, an activity is skipped either since one of its ancestors was skipped (*parentSkip* = *true*), or since its *joinCondition* is *false*. If *skip* evaluates to *false*, the *ActivitySpecificTask* is executed, otherwise the *ComputeTransitionConditions* task is executed.

The *ActivitySpecificTask* is the key task of the pattern. It uniquely identifies the translated activity and it provides the computations needed by the activity. Instantiating the Basic Pattern Template for a particular activity consists of equipping the *ActivitySpecificTask* with a name identifying the activity, and with the inputs and outputs defined by the activity. For example, the Wait pattern has an *ActivitySpecificTask* called *Wait* that inputs the duration of the delay, or the time threshold, similarly to the BPEL *wait*.

The execution of the *ActivitySpecificTask* is simulated through the deferred choice consisting of the *Fault* and *Success* tasks, together with their input place. The environment (viz., the client of the workflow) determines whether *Fault* or *Success* is executed. The execution of the *Fault* task corresponds to an erroneous execution of the activity (e.g., a *receive* activity has received an incorrect message). The *Fault* task outputs the name and data associated with the fault, and it sets the boolean *fault* flag to *true*. Dually, *Success* corresponds to a successful execution of the activity. It is important to note that the deferred choice must be defined only for activities whose execution may be erroneous (e.g., *receive*, *invoke*). Otherwise, the *ActivitySpecificTask* is to be directly connected to the *ComputeTransitionConditions* task (e.g., the Empty pattern template, Wait).

BPEL uses the *suppressJoinFailure* attribute to determine the process behaviour when the *joinCondition* is *false*. If the *suppressJoinFailure* attribute corresponding to an activity (defined by it or by one of its ancestors) is set to *no*, the BPEL engine raises a *joinFailure* fault. Otherwise, it employs the dead-path-elimination by propagating negative statuses on all its output links. The *ComputeTransitionConditions* task concludes the execution of the ELB and of the pattern. On the one hand, it computes the status of each output (synchronisation) link, as defined by the *transitionCondition* attribute of the respective BPEL link. Link statuses are stored into *bo* variables, which have to be mapped onto *bi* variables of other patterns when constructing the workflow of the business process (viz., bo_k = *(not skip) and transitionCondition(*bo_k*)*). On the other hand, it signals a *joinFailure* by setting the *fault* flag to *true* in case of a *false joinCondition* if the corresponding *suppressJoinFailure* attribute is set to *no* (viz., *fault = fault or (not joinCondition and (suppressJoinFailure = NO)*). Note that a red output link is to be defined for a pattern that does not employ the deferred choice (viz., that does not raise faults implicitly) if and only if the *suppressJoinFailure* corresponding to the activity being translated is set to *no* because, otherwise, the red line is redundant. We recall that the only pattern that inputs red lines is the pattern corresponding to the BPEL *fault handler*, which serves for catching and processing faults raised in the process. We will describe the Fault Handler pattern in Subsection 3.2.

Upon completion, the ELB outputs green and blue tokens if and only if the

pattern was successfully executed. Dually, it outputs a red token if and only if a fault was raised. (However, as we shall see later, the successful execution of a Throw pattern outputs green and blue, as well as red tokens.)

In order to obtain the pattern template of a basic activity, one has to:

1. Customise the *ActivitySpecificTask*,
2. Remove the deferred choice controlling the success of the activity if the activity cannot have an erroneous execution, and
3. Set the (maximum) number of inputs and outputs of the pattern.

The customisation of the *ActivitySpecificTask* regards the name of the task, which has to identify the pattern, as well as the inputs and the outputs of the task, which are obtained from the inputs and the outputs of the BPEL activity. Other possible modifications may involve the removal of the *BlueGate*, the employment of guards for the *BlueGate*, and so on, as we shall see in the following. Furthermore, note that a pattern always has at least one green input and one green output.

In the following we describe the pattern templates corresponding to the BPEL basic activities. Note however that we translate the *assign* and the *compensate* using structured pattern templates, due to the execution semantics of the two activities. Both patterns will be described in Subsection 3.2.

Empty. The BPEL *empty* activity has the following form:

```
<empty standard-attributes>
      standard-elements
</empty>
```

where the *standard-attributes* are:

```
name="ncname"?
joinCondition="bool-expr"?
suppressJoinFailure="yes|no"?
```

and the *standard-elements* are:

```
<source linkName="ncname" transitionCondition="bool-expr"?/>*
<target linkName="ncname"/>*
```

An *empty* activity performs a "no-op", and it may be useful e.g., inside *fault handler*s to suppress caught faults, or as milestones inside *flow* activities. The *joinCondition* is a boolean expression constructed using the statuses of the incoming synchronisation links as operands, and a *false joinCondition* leads to skipping the activity. If the *suppressJoinFailure* attribute is set to *yes*, then the activity is silently skipped, otherwise a *joinFailure* fault is raised by the BPEL engine. *Source* tags define synchronisation links emerging from the activity, and the statuses of the respective links are to be given by *transitionCondition* boolean expressions. Dually, an activity can be set as target of a synchronisation

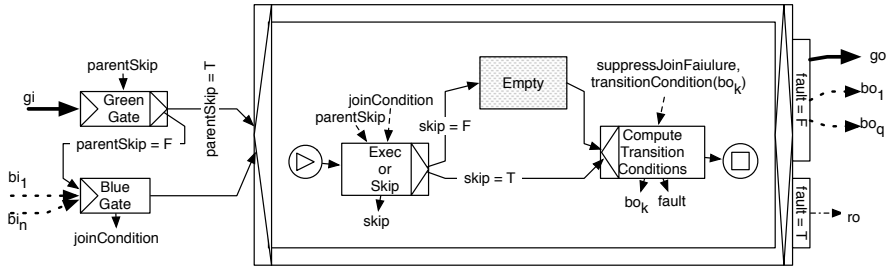


Figure 3: The Empty pattern template.

link using the *target* element. Note that both, the standard attributes and the standard elements, are optional.

The Empty pattern (see Figure 3) is the simplest pattern. It does not contain the deferred choice block (consisting of the YAWL condition together with the *Fault* and *Success* tasks) as the execution of an *empty* activity cannot raise an explicit fault. Consequently, the *Empty* task is directly connected to the *ComputeTransitionConditions* task. Furthermore, *Empty* does not employ any inputs and outputs (IOs) because the *empty* activity does not define any variables. Note that the Empty pattern has one green input and one green output only, because from the structural viewpoint an *empty* activity has one predecessor and one successor only.[5]

Receive. The BPEL *receive* has the following form:

```
<receive partnerLink="ncname" portType="qname" operation="ncname"
         variable="ncname"? createInstance="yes|no"?
         standard-attributes>
    standard-elements
    <correlations>?
         <correlation set="ncname" initiate="yes|no"?>+
    </correlations>
</receive>
```

BPEL uses *receive* activities to input messages either from the invoker of the BPEL process, or from partner Web services. Roughly speaking, the *receive* specifies the *partnerLink* it expects to receive from, and the *portType* and *operation* that it expects the partner to invoke. The *variable* (used to store the received message) and the *createInstance* (used to instantiate the business process) attributes are both optional. If the *createInstance* is set to *yes*, then the reception of a message by the *receive* leads to starting a new process instance. By default the *createInstance* is set to *no*. Since a business process typically holds one or more conversations with its partners, BPEL uses *correlation set*s to route the messages involved in a conversation to the correct service instance.

[5] Although some constructs are redundant, e.g., the AND-join of the *GreenGate* task, we shall keep them in the patterns in order to simplify the description of the methodology.

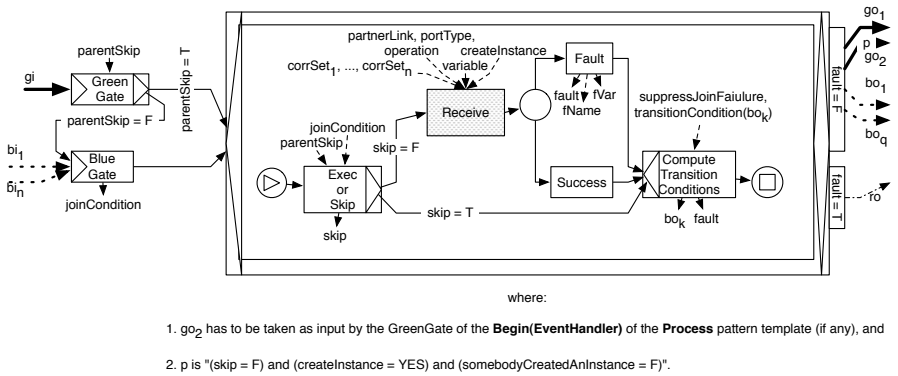


Figure 4: The Receive pattern template.

For example, the various conversations a seller process holds with its buyers may be distinguished by using e.g., the purchase order number (supplied by buyers at the initiation of the conversation) as correlation token. We shall not go into further details on the correlation of business process instances, since YAWL does not model multiple workflow instances in this dynamic way. In YAWL, the clients of a workflow are in charge of (manually) starting a new workflow instance (called workflow *case*) by instantiating a workflow specification in the YAWL engine. As [16] notes, the engine handles the execution of these cases, i.e. based on the state of a case and its specification, the engine determines which events it should offer to the environment. However, the translator we propose in this paper imports the correlation sets (and other information that is strictly related to, and mandatory only for the execution of the BPEL process, such as *partnerLink*s, *portType*s, and so on) into the YAWL workflow as (global) variables, which are useful e.g., for an inverse YAWL2BPEL translator.

The pattern of the *receive* activity is given in Figure 4. As expected, the *ActivitySpecificTask* is now called *Receive*. It inputs the *partnerLink*, *portType*, and *operation*, as well as the optional *variable*, *createInstance*, and *correlationSet* attributes of the *receive* activity. Note that instantiating the Receive pattern template for a particular business process to be translated, resumes to "hard-coding" all the inputs of the *Receive* task but the *variable* one, with the values of the corresponding attributes in the *receive* activity. The *variable* input receives a value at run time, which logically corresponds to the value inputted by the *receive* activity in the BPEL process. The Receive has one green input only (coming from the pattern of the activity structurally preceding it), and it can employ up to two outputs: one (mandatory) for the pattern of the activity structurally following it, and another (optional) used to enable the pattern for event handling of the entire business process. (More details on this later, when describing the pattern of the BPEL *process*.) Please note that the second (optional) output should be used only if the BPEL process to be translated has

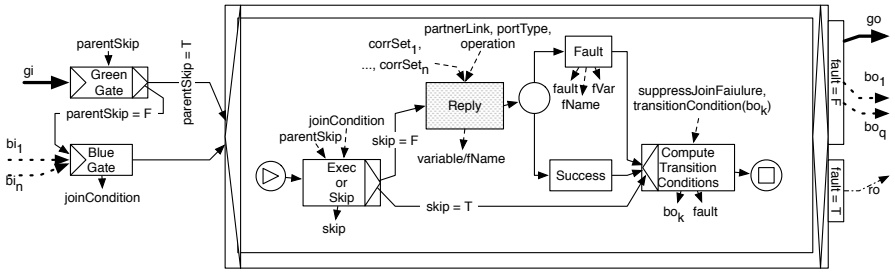


Figure 5: The Reply pattern template.

an *event handler* at the process level and if the *createInstance* attribute of the *receive* activity is set to *yes*. Consequently, a green token is sent on the second green output provided the following logical condition holds: *"(skip = F) and (createInstance = YES) and (somebodyCreatedAnInstance = F)"*. We use the *somebodyCreatedAnInstance* (global) variable in order to avoid multiple green tokens being sent to the pattern for event handling of the entire business process by multiple Receive patterns. For example, assume that the business process consists of a *flow* having two *receive* operations. In this case the first *receive* to be executed enables the *event handler* of the process.

Reply. The BPEL *reply* has the following form:

```
<reply partnerLink="ncname" portType="qname" operation="ncname"
            variable="ncname"? faultName="qname"?
            standard-attributes>
      standard-elements
      <correlations>?
            <correlation set="ncname" initiate="yes|no"?>+
      </correlations>
</reply>
```

BPEL uses *reply* activities to send response messages to requests previously accepted through *receive* activities. (The combination of a *receive* and a *reply* forms a request-response operation on the WSDL portType of the process.) Similarly to a *receive*, the *reply* is identified by the triple (*partnerLink*, *portType*, *operation*), and it may use *correlation set*s to identify the conversations with its business partners. A *reply* activity can indicate either a *normal* or a *faulty* execution of the request-response operation. On the one hand, a normal response does not include the *faultName* attribute, and the *variable* attribute (if present) provides the data to be sent to the partner. On the other hand, the presence of the *faultName* attribute indicates a faulty execution, and the *variable* attribute (if present) gives the fault data.

The Reply pattern template is presented in Figure 5. The inputs of its *Reply* task are the *partnerLink*, *portType* and *operation*, as well as the optional *correlationSet* attributes of the *reply* activity, while its output is either a *variable*

or a *faultName*, or both a *variable* and a *faultName*, as defined in the *reply* activity. It is important to note that a Reply can raise an error either explicitly, or implicitly. The former is due to the execution of the *Reply* task in the case of a *reply* activity that defines a *faultName* attribute. The latter corresponds either to the execution of the *Fault* task (e.g., corresponding to a mismatch between the types of the message outputted by the *reply* in the business process and of the message inputted by the Web service receiving it), or to skipping the *reply* activity when its corresponding *suppressJoinFailure* attribute is set to *no*. In both cases, a red token is generated by the Reply pattern. Finally, a Reply has one green input and one green output.

Invoke. The BPEL *invoke* has the following structure:

```
<invoke partnerLink="ncname" portType="qname" operation="ncname"
        inputVariable="ncname"? outputVariable="ncname"?
        standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiate="yes|no"?
            pattern="in|out|out-in"/>+
    </correlations>
    <catch faultName="qname" faultVariable="ncname"?>*
        activity
    </catch>
    <catchAll>?
        activity
    </catchAll>
    <compensationHandler>?
        activity
    </compensationHandler>
</invoke>
```

A BPEL process can *invoke* operations offered by the partner Web services. An asynchronous invocation (viz., of a one-way WSDL operation) requires only the *inputVariable* to be defined, while for a synchronous invocation (viz., of a request-response WSDL operation) the *invoke* should define both *inputVariable* and *outputVariable*. Similarly to the *receive* and *reply* operations, the *invoke* may use *correlation set*s. Note that a synchronous invocation returning with a WSDL fault (see *reply* before) can be caught locally by the *invoke* through the inline *catch*/*catchAll*, which will execute the activity it contains. Moreover, the *invoke* may define a "roll-back" activity through the inline *compensation handler*. However, the invoke with the inline fault and compensation handlers is semantically equivalent to the same *invoke* activity enclosed in a *scope* that defines the respective fault and compensation handlers. (See the translation of the *scope* structured activity for more information on the fault and compensation handlers.) Hence, in order to simplify the translation methodology we shall treat *invoke* activities with inline fault and compensation handlers as *scope*s immediately enclosing the *invoke*s and providing these handlers.

Similarly to the Receive and Reply pattern templates, the Invoke (depicted in Figure 6) has an *Invoke* task that inputs a *partnerLink*, *portType*, and *operation*, as well as (optional) *correlationSet* variables. When instantiating it for the

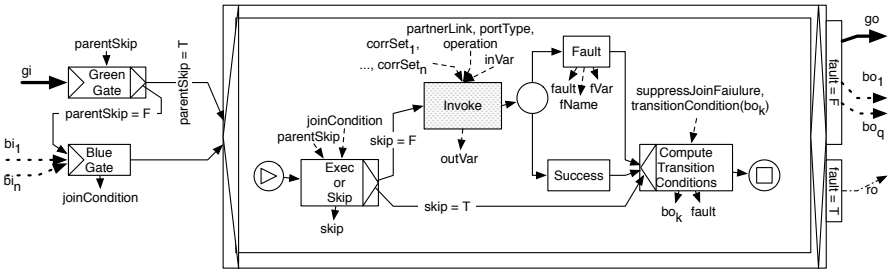


Figure 6: The Invoke pattern template.

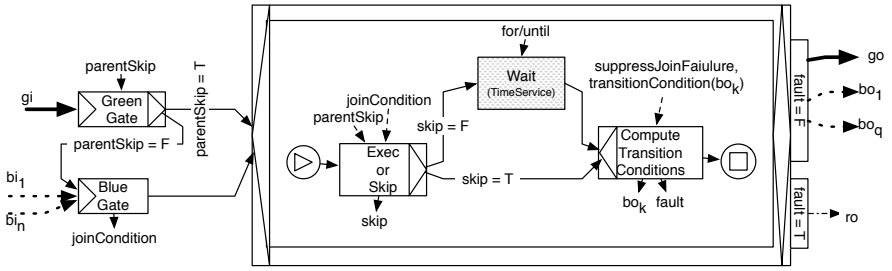


Figure 7: The Wait pattern template.

translation of a particular BPEL process, a *correlationSet* variable is defined for each *correlation* attribute of the *invoke* activity in the business process. The two last variables of the *Invoke* task are *inVar* and *outVar*. Both are optional and they are defined by an instance of the Invoke pattern template (i.e., an Invoke translating an *invoke* activity in a particular BPEL process) only when corresponding attributes exist in the *invoke* activity being translated. [6]

Wait. The BPEL *wait*:

```
<wait (for="duration-expr" | until="deadline-expr") standard-attributes>
      standard-elements
</wait>
```

delays the execution of a business process *either* for a certain period of time (through the *for* attribute) *or* until a certain deadline is reached (through the *until* attribute).

The Wait pattern template is given in Figure 7. Its construction is identical to the Empty pattern with the exception of the *ActivitySpecificTask*. The Wait pattern template allows the *Wait* task to have either a *for* or a *until* variable,

[6] Hence, the *outVar* output is defined by the *Invoke* task only when translating synchronous *invoke* operations.

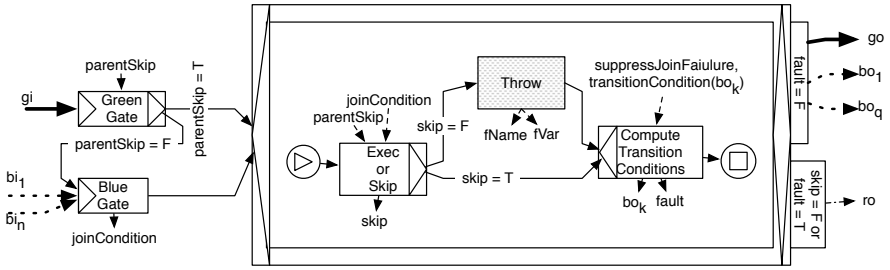


Figure 8: The Throw pattern template.

depending on the corresponding attribute defined in the business process to be translated. (The only particularity of the *Wait* task with respect to all other *ActivitySpecificTask*s is that it invokes the YAWL *TimeService* in order to delay the execution of the workflow.)

Throw. The structure of the BPEL *throw* is the following:

```
<throw faultName="qname" faultVariable="ncname"? standard-attributes>
      standard-elements
</throw>
```

A *throw* activity serves for explicit fault signalling. Each fault is defined by a (unique) *faultName* and an (optional) *faultVariable* containing the fault data.

The Throw pattern template is illustrated in Figure 8. It employs a *Throw* task that outputs a *faultName* and/or a *faultVariable*. It is important to note that Throw outputs a red token, either if a *joinFailure* fault is being raised (viz., *fault* = *T*), or if the *Throw* task is executed (viz., *skip* = *F*). While in the former case (corresponding to an erroneous execution of the BPEL *throw*) the Throw pattern only outputs one red token, in the latter case (corresponding to a successful execution of the BPEL *throw*) green and blue tokens (if any) are generated as well. In other words, an explicit fault raised by a Throw pattern is not considered as erroneous execution of the *throw* activity.

Terminate. *Terminate* activities are defined as follows:

```
<terminate standard-attributes>
      standard-elements
</terminate>
```

A *terminate* activity is used to end the execution of the entire business process instance. All running activities are to be terminated immediately without any fault or compensation handling. The semantics of activity termination [2] depends on the activity to be interrupted. For example, *assign* activities are allowed to finish their execution, while *wait* activities are ended immediately. Although it is not trivial, one can obtain this behaviour in the translated YAWL workflow by suitably equipping the pattern corresponding to the end of the

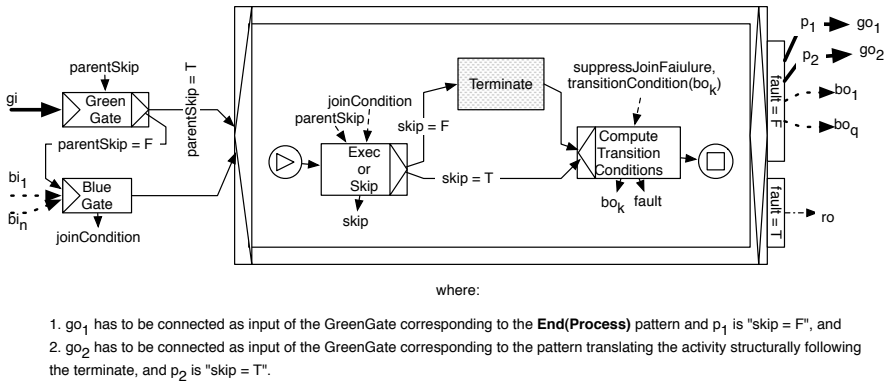


Figure 9: The Terminate pattern template.

business process (see End(Process) in Subsection 3.3) with a cancellation set including only the activities whose execution has to be interrupted. However, in order to keep the translation simple, our translator adds the entire Pattern Template corresponding to the activity defined by the *process* into the cancellation set of End(Process).

The pattern template of the BPEL *terminate* (see Figure 9) is quite similar to the Empty pattern. However, Terminate outputs only one green token on one of its two green outputs. If the Terminate is skipped without raising a *joinFailure* (i.e., *"fault = F and skip = T"*), then a green token is sent to the pattern translating the activity structurally following the *terminate* in the BPEL process. Otherwise, if the Terminate is executed (i.e., *"fault = F and skip = F"*) then the green token is sent to End(Process) in order to cancel the execution of the entire business process.

For simplicity, the BPEL *assign* and *compensate* activities are translated into structured patterns, as we shall see in the next Subsection. On the one hand, the BPEL *assign* may contain several *copy* tags each signifying a data exchange that may lead to a fault being raised. Hence, we treat the *assign* similarly to a *sequence* activity. On the other hand, the BPEL *compensate* leads to the execution of the activity defined in the *compensation handler* of the *scope* to be invoked. As a result, the *compensate* finishes its execution when the activity in the invoked *compensation handler* has finished its execution. This leads to the need of explicitly representing the beginning and the end of the *compensate*, and consequently we treat it as a structured activity.

3.2 Patterns of BPEL structured activities

A BPEL structured activity defines one or more activities to be executed in a certain order. In order to cope with this, we define the Structured Pattern

Template as a tuple consisting of a Begin pattern, an End pattern, as well as a Pattern Template for each child activity.

The purpose of the Begin and End patterns is to provide an identification for the activity being translated. More importantly, the execution of Begin logically corresponds to the initiation of the structured activity (as a whole), whereas the execution of End logically marks the termination of the structured activity. Both Begin and End patterns are generated from the Basic Pattern Template, and they are quite similar to the Empty pattern. On the one hand, Begin is in charge of enabling the structured pattern both from the structural and synchronisation viewpoints. Hence, Begin has to input the green and the blue lines and to raise a *joinFailure* in case of a *false joinCondition* if the corresponding *suppressJoinFailure* attribute is set to *no*. Furthermore, it provides a green output for each Pattern Template corresponding to a child activity that can be executed first. On the other hand, End has to wait for the green tokens from all Pattern Templates of the child activities that have to be executed last. Moreover, End is the source of the blue outputs corresponding to synchronisation links having as source the structured activity. In general, End cannot lead to any faults being raised, and hence it does not have a red output.

A structured activity introduces a new nesting level and consequently Begin has to output a *parentSkip* variable to the patterns of all the (child) activities nested inside the structured one, as well as to the End pattern. In this way we achieve the dead-path-elimination inside structured patterns.

Now, the pattern templates of all structured activities are obtained by adjusting the Begin and End patterns and by suitably interconnecting them with the Pattern Templates. Basically, both processes depend on the way in which the structured activity enables for execution its child activities. In the following we shall write Begin(X) and End(X) to refer to the Begin and End patterns of a structured activity X.

The BPEL structured activities are:

- *sequence*, *switch*, and *while*, that provide sequential control between activities,
- *flow*, which provides concurrency and synchronisation between activities,
- *pick*, which provides nondeterministic choice based on external events, and
- *scope*, which provides a behaviour context for activities.

Furthermore, as we already mentioned, in this subsection we shall describe the implementation of the BPEL basic activities *assign* and *compensate*.
The Sequence, Switch, Flow and Pick patterns all share the same structure:

Sequence	$\rightarrow$	Begin(Sequence)	PatternTemplate$^+$	End(Sequence)
Switch	$\rightarrow$	Begin(Switch)	PatternTemplate$^+$	End(Switch)
Flow	$\rightarrow$	Begin(Flow)	PatternTemplate$^+$	End(Flow)
Pick	$\rightarrow$	Begin(Pick)	PatternTemplate$^+$	End(Pick)

Sequence. The BPEL *sequence* defines one or more activities to be performed sequentially, in lexical order, and it has the following structure:

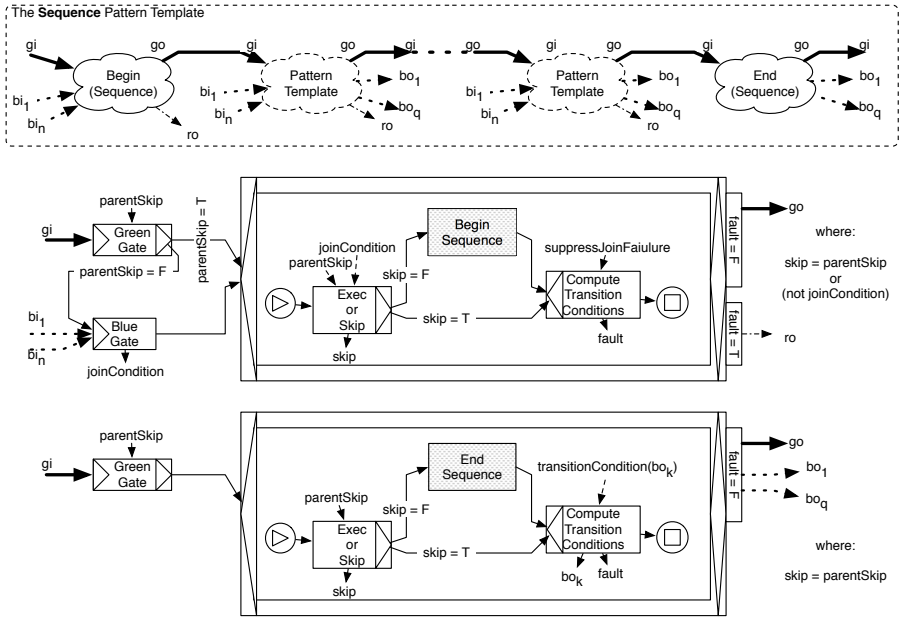


Figure 10: The Sequence pattern template.

```
<sequence standard-attributes>
        standard-elements
        activity+
</sequence>
```

The Sequence pattern (see Figure 10) is the simplest structured pattern template. Begin(Sequence) differs from the Empty pattern template in that it does not have blue output links. This is because the statuses of the BPEL synchronisation links emerging from a *sequence* are to be computed upon completion of the *sequence* activity. Consequently, the output blue links translating the emerging synchronisation links (if any) are defined by the End(Sequence) pattern template.[7] As End(Sequence) logically marks the termination of a *sequence*, it cannot be the target of a synchronisation link, and hence it does not have any blue inputs. Note however the red output of Begin(Sequence), which serves for signalling a *joinFailure* since the synchronisation links targeting a *sequence* activity are translated into blue inputs of the Begin(Sequence) pattern template. Furthermore, End(Sequence) does not have a red output as its execution cannot lead to faults being raised. On the one hand, the *ExecOrSkip* task of Begin(Sequence) computes the value of the *skip* variable as a disjunction between the *parentSkip* and the negation of the *joinCondition*. On the other hand, End(Sequence) directly sets the value of the *skip* variable to the value of

[7] Note that the *ComputeTransitionConditions* of Begin(Sequence) computes only the *fault* flag.

the *parentSkip*.

The top-part of Figure 10 presents the structural dependencies (i.e., how the sequence Pattern Templates are connected through green lines) among the Begin(Sequence) pattern, the Pattern Templates translating the child activities of the *sequence*, and the End(Sequence) pattern. In the following we shall use the "cloud" symbol as a simplified denotation of a Pattern Template. (We recall that a Pattern Template is used to denote the pattern of a generic BPEL activity.) Note that the cloud representing each Pattern Template is dashed as it may correspond to the translation of a structured BPEL activity (e.g., another *sequence*), and hence it may contain several other Pattern Templates (i.e., clouds). Begin(Sequence) has one green output only because only one activity can be executed first in a *sequence*. Consequently, the green output of Begin(Sequence) is linked as input of the Pattern Template translating the first activity in the BPEL *sequence*. Dually, End(Sequence) employs one green input only, which comes from the pattern of the last activity in the *sequence*. Furthermore, each Pattern Template translating a BPEL activity in the *sequence* (except the first and the last ones) has a green input from the Pattern Template of the previous activity in the *sequence*, and a green output for the Pattern Template of the next activity in the *sequence*.

Switch. The *switch* activity consists of one or more conditional branches guarded by boolean expressions as well as an (optional) *otherwise* branch. The activity to be executed by the *switch* is determined by the first guard that holds true in lexical order. The activity corresponding to the *otherwise* branch is executed provided no guard holds. When the *otherwise* branch is not specified, BPEL considers a default *otherwise* enclosing an *empty* activity.

```
<switch standard-attributes>
        standard-elements
        <case condition="bool-expr">+
                activity
        </case>
        <otherwise>?
                activity
        </otherwise>
</switch>
```

The pattern template of a BPEL *switch* (illustrated in Figure 11) is constructed similarly to the pattern of a *sequence* activity. It is composed of Begin(Switch), End(Switch), as well as one or more Pattern Templates for each (child) activity defined by a conditional branch (viz., case) of the BPEL *switch*. Begin(Switch) and End(Switch) are similar to Begin(Sequence) and End(Sequence), respectively. The former logically marks the beginning of the *switch* and it is in charge of activating the Switch pattern by waiting for the green token, as well as for the blue ones (if any). The latter marks the end of the BPEL *switch* and it sets the statuses of its output links (if any).

As previously mentioned, the guards of the *switch* branches are evaluated in the order in which they appear. This is the reason why the Switch pattern

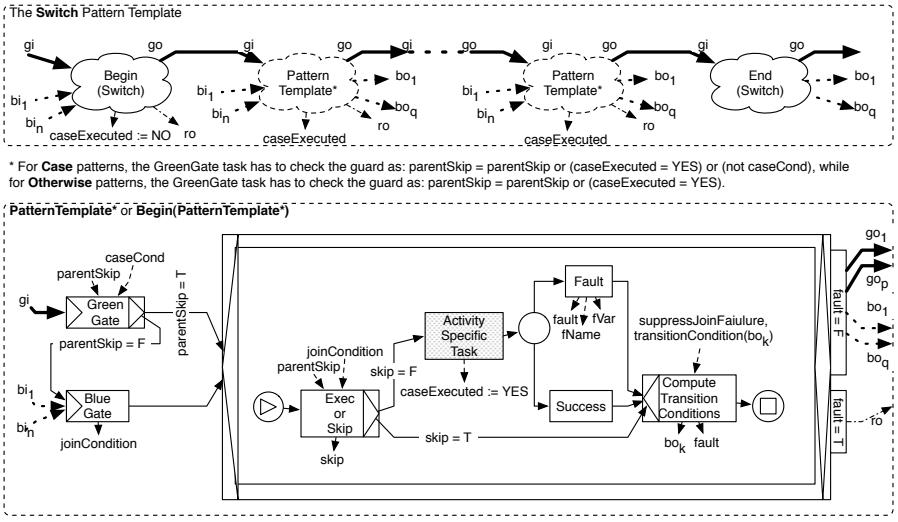


Figure 11: The Switch pattern template.

template is constructed by sequentially linking (through the green line) the Pattern Templates corresponding to all conditional branches. The particularity of the Pattern Template that translates an activity defined by a *case* or *otherwise* conditional branch is that its *GreenGate* task has to check whether a previous branch pattern was executed.[8] Furthermore, the Case patterns have to check further whether the guard condition holds (see the bottom-part of Figure 11). As a result, at run-time, if a branch pattern was already executed, or if the guard does not hold, the pattern of the respective branch is skipped in order to employ the dead-path-elimination. As the BPEL specification notes [2], if there is no *otherwise* branch defined, a default one with an *empty* activity has to be considered. Consequently, the translator automatically considers for the translation of such *switch* activities an Empty pattern.

Flow. The BPEL *flow* provides concurrency and synchronisation inside the business process. Its structure is as follows:

```
<flow standard-attributes>
      standard-elements
      <links>?
            <link name="ncname">+
      </links>
      activity+
</flow>
```

[8] Note that the *Switch* pattern makes use of a *caseExecuted* variable initially set to *no* by Begin(Switch), and further set to *yes* by the branch pattern executed first. In order to avoid the execution of multiple branches, each branch pattern guard simply checks the status of the *caseExecuted* variable (see the bottom-part of Figure 11).

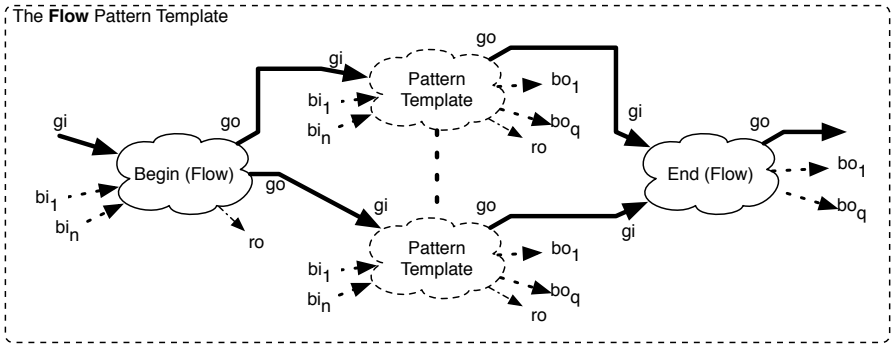


Figure 12: The Flow pattern template.

All child activities of the *flow* are executed as soon as the *flow* starts, provided they are not targeted by any synchronisation link. One may note below that the grammar of the *flow* activity allows for links to be defined. The execution of an activity that is the target of at least one synchronisation link is delayed until the statuses of all of its incoming links are known, and it will be executed only if its corresponding *joinCondition* holds *true*. Otherwise, depending on the (corresponding) value of the *suppressJoinFailure* attribute, the activity is either silently skipped, or a *joinFailure* is raised. Note that the dead-path-elimination process will forward negative (viz., false) statuses on all the output links (if any) of the activity being silently skipped.

The Flow pattern template (see Figure 12) employs similar constructs to the Sequence one. The only difference between Begin(Flow) and Begin(Sequence) is that the former has multiple green outputs, one for each Pattern Template translating a child activity of the *flow*. Dually, End(Flow) differs from End(Sequence) in that it has multiple green inputs, each coming from a Pattern Template. This is motivated by the fact that the execution of a BPEL *flow* starts by enabling from the structural viewpoint all its children activities. (Consequently, tokens are sent on all its green outputs provided the *fault* flag is *false*.) Dually, the *flow* terminates only when all its child activities have finished their execution. This is achieved through the AND-join of the *GreenGate* task of End(Flow).

Pick. The grammar of the BPEL *pick* is given hereafter.

```
<pick createInstance="yes|no"? standard-attributes>
      standard-elements
      <onMessage partnerLink="ncname" portType="qname"
                 operation="ncname" variable="ncname"?>+
            <correlations>?
                  <correlation set="ncname" initiate="yes|no"?>+
            </correlations>
            activity
      </onMessage>
      <onAlarm (for="duration-expr" | until="deadline-expr")>*
```

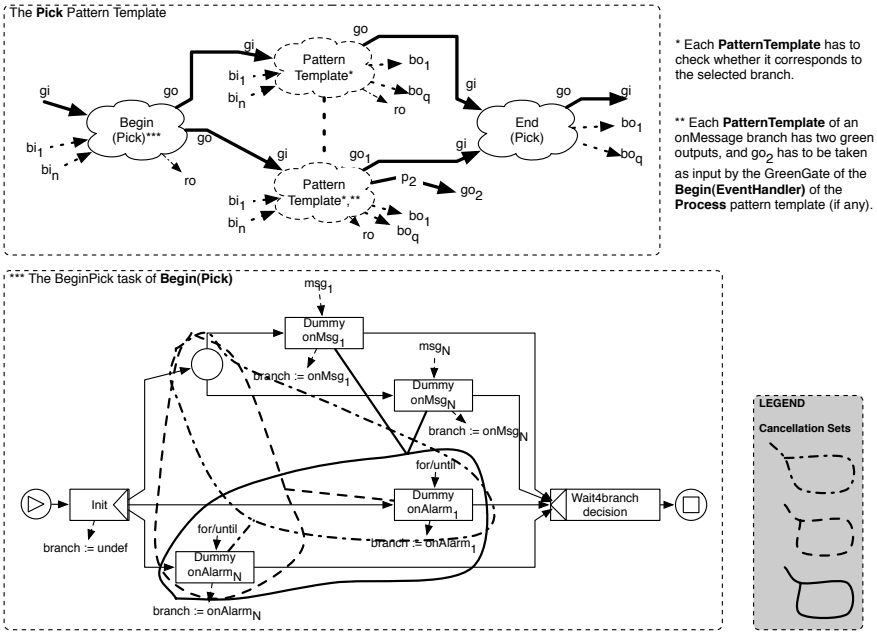


Figure 13: The Pick pattern template.

```
            activity
        </onAlarm>
</pick>
```

A *pick* defines one or more *onMessage* elements, as well as optional *onAlarm* elements. Through an *onMessage* element the business process waits for a message event from its partner Web services, similarly to a *receive*. Note that a message event can use correlation sets, as well as it may start a business process instance. Note that, differently from the deterministic choice made by the *switch* activity and concerning the activity to be executed, the *pick* makes a non-deterministic choice inside the business process, as the environment decides the activity to be executed next. Furthermore, an *onAlarm* waits for an alarm event to take place, similarly to a *wait*. Roughly, the execution of the *pick* resumes to waiting the occurrence of either a message or an alarm event, which leads to executing the activity associated with the event that took place. The occurrence of a message event immediately inactivates the other message events, as well as all the alarms, so that they cannot be triggered. Dually, if an alarm event goes off, all the message events are inactivated, as well as all the other alarms are set off. The *pick* finishes when the activity corresponding to the branch that was triggered terminates.

The high-level view of the Pick pattern template (Figure 13) is similar to the one of the *flow* activity. Begin(Pick), like Begin(Flow), outputs multiple

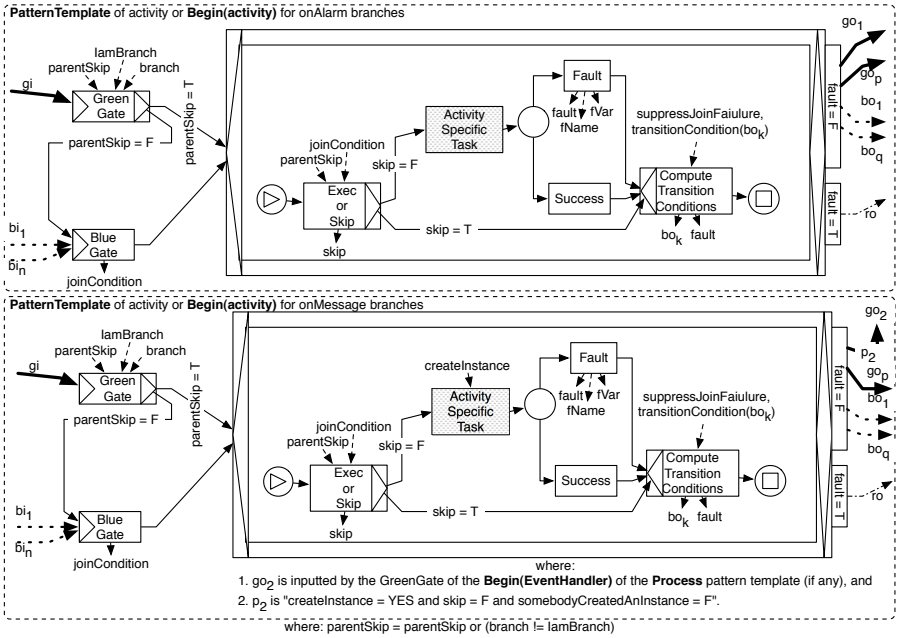


Figure 14: The pattern templates for the activities on the *pick*'s branches.

green lines, one for each of the *pick*'s branch activities. However, its *BeginPick* task is a composite task in charge of branch selection (see the bottom part of Figure 13). *BeginPick* employs one *Dummy onMsg*$_i$ task for each *onMessage* branch, as well as one *Dummy onAlarm*$_j$ task for each *onAlarm* branch of the *pick*. The execution of the *Init* task of *BeginPick* places a token in the condition of the deferred choice as well as it enables the alarm tasks. Note that all *Dummy onAlarm*$_j$ tasks use (similarly to the *Wait* task of the Wait pattern) the YAWL *TimeService* to implement the timer. Although all *Dummy onMessage*$_i$ are executable due to the token in the deferred choice condition, the first one to be executed (i.e., the first message that arrives) clears the token. As a consequence, *Dummy onMessage*$_i$ sets the value of the *branch* variable to its identification (which corresponds to the pattern of the activity being triggered). Furthermore, its execution leads to canceling all timers (see the solid-line cancellation set in Figure 13). Then, the *Dummy onMessage*$_i$ task forwards the token to the *Wait4BranchDecision* task, which marks the termination of the *BeginPick* composite task. The other possible execution scenario consists in the completion of a *Dummy onAlarm*$_j$ task when the respective timer sets off before any *Dummy onMessage*$_i$ is executed. The result is that *Dummy onAlarm*$_j$ sets the value of the *branch* variable to its identification, and then it cancels all the other timers and it clears the token in the deferred choice condition (so that no *Dummy onMessage*$_i$ tasks can be executed) – see the dashed-line cancellation

sets in Figure 13. Finally, the green token reaches the *Wait4BranchDecision* task and the execution of *BeginPick* terminates.

It is important to note that even though only one branch (i.e., the triggered one) will be executed, the Begin(Pick) pattern outputs green tokens for all branch patterns in order to achieve dead-path-elimination on the branches that were not selected. The End(Pick) pattern template is the same as End(Flow). It just waits for the green tokens from all branch patterns.

Another important characteristic of the Pick pattern is the slight modification it brings to the Pattern Templates translating its branch activities (see Figure 14). Each such Pattern Template has to check whether the respective branch was triggered, by comparing its identification (viz., the *IamBranch* variable in the Figure) with the one outputted by the *BeginPick* composite task of Begin(Pick) (viz., the *branch* variable in the Figure). The difference between the patterns of message and alarm branch activities is that the pattern for a message branch has to output (similarly to the Receive pattern template) a green token to enable the pattern for event handling of the entire business process if and only if the *createInstance* is set to *yes*, if no faults were raised by the branch pattern, if the Pick was not skipped and it can create a process instance, and if no other Receive or Pick branch has already created a process instance. (See also the Scope pattern template next.) Furthermore, note that this green output should be defined if and only if the *createInstance* attribute is set to *yes* in the *pick* activity being translated.

One last thing to note about the BPEL *pick* is that if a branch activity A of the *pick* is a basic one, then its pattern template is constructed as shown in Figure 14, depending on the branch type (message or alarm). Otherwise, if A is a structured activity, then its Begin(A) and End(A) patterns will incorporate the modifications shown in Figure 14. (Note that this applies to all other structured pattern templates that bring modifications to the patterns of their children activities.)

While. The BPEL *while* repeatedly executes its child activity for as long as the boolean while guard holds true. Its structure is the following:

```
<while condition="bool-expr" standard-attributes>
      standard-elements
      activity
</while>
```

The While pattern

$$\text{While} \rightarrow \text{Begin(While)} \quad \text{PatternTemplate} \quad \text{End(While)}$$

(see Figure 15) consists of Begin(While), a Pattern Template, as well as End(While), linked in a sequence. The main particularity of the pattern is that Begin(While) takes two green inputs – one from the pattern of the activity (structurally) preceding the *while*, and another from End(While). The former is inputted by the *GreenGate* task, while the latter directly enables the Execution Logic Block of the While as it corresponds to a new iteration and Begin(While) should not wait for more tokens on the blue inputs (if any). Note that at run-time only

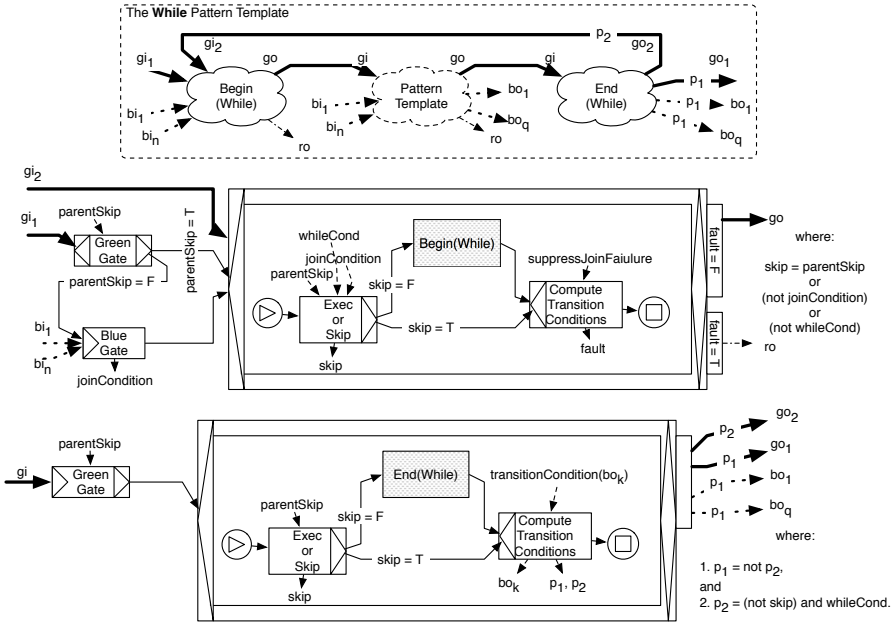


Figure 15: The While pattern template.

one of the two input green tokens is needed to structurally enable Begin(While). Furthermore, the *ExecOrSkip* task of Begin(While) is in charge of checking the loop guard. If the guard evaluates to *true*, the While is executed, otherwise it is skipped in order to achieve the dead-path-elimination.[9] The particularity of End(While) (with respect to End(Sequence)) is that it has two green output lines, one which goes to the pattern of the activity to be executed next, and another returning to Begin(While). The *ComputeTransitionConditions*, in addition to computing the statuses of the blue (synchronisation) output links, checks the loop guard as well. If the guard holds *true*, End(While) outputs only one green token, on the link to Begin(While). Otherwise, the green token is sent to the pattern to be executed next. We double-check the guard in End(While) because, if we would simply forward the green token to Begin(While), a *false* guard would lead to employing the dead-path-elimination inside the While, which would be incorrect as the loop has already been executed. Finally, it is important to note that blue outputs are sent by End(While) only when the While terminates, that is, after skipping it (viz., *skip* = *true*) or if the loop guard does not hold (viz., *whileCond* = *F*).

Assign. The BPEL *assign* can be used to copy data between variables, to

[9] Note that one cannot check the guard in the *GreenGate* as the guard may employ variables set by activities that target the *while*, and hence the verification of the guard has to be done after receiving all blue tokens.

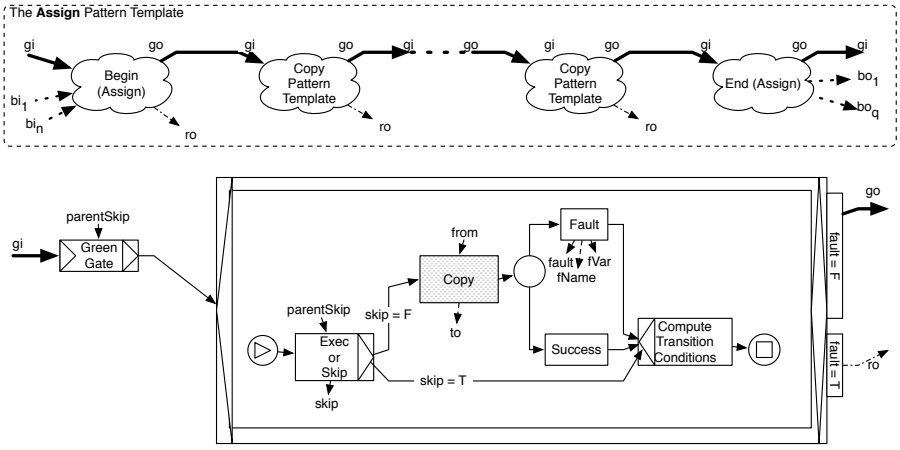


Figure 16: The Assign pattern template.

perform simple computations by mapping expressions onto variables, as well as to copy endpoint references to and from partner links. From the *assign* grammar given below, one may note that an *assign* may define several *copy* elements, each one performing an assignment.

```
<assign standard-attributes>
        standard-elements
        <copy>+
                from-spec
                to-spec
        </copy>
</assign>
```

where the *from-spec* and *to-spec* have the following structures:

```
<from variable="ncname" part="ncname"?/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
```

and

```
<to variable="ncname" part="ncname"?/>
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>
```

The Assign pattern:

$$\text{Assign} \quad \rightarrow \quad \text{Begin(Assign)} \quad \text{Copy}^{+} \quad \text{End(Assign)}$$

has the same structure as the Sequence pattern, but it includes Copy patterns rather than arbitrary Pattern Templates (see Figure 16). (We recall that we translate the BPEL *assign* to a structured pattern template due to the fact that an *assign* may contain several *copy* attributes, each requiring a data exchange

which may lead to a fault being raised.) Begin(Assign) and End(Assign) are identical to Begin(Sequence) and End(Sequence), respectively. Furthermore, due to the fact that BPEL evaluates the assignments in the order in which the *copy* attributes appear in the *assign*, we link the Copy patterns (through the structural green line) in a sequence. The Copy pattern template does not have blue IOs as the BPEL *copy* can be neither the source, nor the target of a synchronisation link. The assignment is carried out by the *Copy* task, which maps a (complex) input variable corresponding to the *from* element in the BPEL *copy* onto a (complex) output variable corresponding to the *to* element in the BPEL *copy*. Hence, an instance of the Copy pattern translating a particular BPEL *copy* defines the *from* and *to* variables of the *Copy* task depending on the similar attributes of the BPEL *copy* element (e.g., *variable*, *part*, *expression*). [10] Finally, the *Fault* task of the Copy pattern can be used to simulate an assignment mismatch, and in such case a red token is outputted by the faulty Copy pattern.

Scope. The BPEL *scope* is the most complex structured activity, and it employs the following structure.

```
<scope variableAccessSerializable="yes|no" standard-attributes>
        standard-elements
        <variables>?
                ...
        </variables>
        <correlationSets>?
                ...
        </correlationSets>
        <faultHandlers>?
                ...
        </faultHandlers>
        <compensationHandler>?
                ...
        </compensationHandler>
        <eventHandlers>?
                ...
        </eventHandlers>
        activity
</scope>
```

where the handlers are defined as follows:

```
<faultHandlers>?
        <catch faultName="qname"? faultVariable="ncname"?>*
                activity
        </catch>
        <catchAll>?
                activity
        </catchAll>
</faultHandlers>

<compensationHandler>?
        activity
</compensationHandler>

<eventHandlers>?
        <onMessage partnerLink="ncname" portType="qname"
```

[10] Note that both BPEL and YAWL support XML Schema type definitions and use XPath for data manipulation, and hence translating the BPEL *copy* into the mapping done by the *Copy* task is straightforward.

```
                operation="ncname"
                variable="ncname"?>*
            <correlations>?
                <correlation set="ncname" initiate="yes|no">+
            </correlations>
            activity
        </onMessage>
        <onAlarm for="duration-expr"? until="deadline-expr"?>*
            activity
        </onAlarm>
</eventHandlers>
```

Roughly, a BPEL *scope* provides a specific context for an activity. It allows for the definition of variables (that live only within the scope) and correlation sets. Furthermore, it contains a (possibly default) optional *fault handler*, a (possibly default) optional *compensation handler*, as well as an optional *event handler*.

The *fault handler* consists of one or more *catch* clauses for grabbing faults raised inside the *scope*. A *catch* is a container of an activity, guarded by a *faultName* and an optional *faultVariable*. The *fault handler* may also specify a *catchAll*, which is similar to a *catch*, yet it does not employ a guard so as to process all faults that reach it. It is important to note that the *catch*es are evaluated in lexical order, and the following rules apply:

1. If the fault has no fault data, BPEL selects the first *catch* with matching *faultName*, or the *catchAll* (if defined). Otherwise,

2. If the fault contains fault data, BPEL selects the first *catch* with matching *faultName* and *faultVariable*. If no such match exists, BPEL selects the first *catch* with matching *faultVariable* and no specified *faultName*, or the *catchAll* (if defined).

3. If the fault does not match any *catch* and if there is no *catchAll* defined, the fault is rethrown to the immediately enclosing scope. Note further that faults uncaught at the process scope lead to an abnormal process termination, as for a *terminate*. Moreover, a scope in which a fault occurs is considered to have ended abnormally, even if the fault is processed by the *scope*'s *fault handler*.

The *compensation handler* provides a (compensating) activity that can be invoked either explicitly (through a *compensate* that specifies the scope to be compensated), or implicitly (during the default compensation mechanism). The *compensation handler* is activated only when the *scope* finishes its execution successfully, and consequently, invoking a compensation handler that was not installed is equivalent to a no-op. Note that in this paper we deal with explicit compensation only, due to the troublesome default compensation mechanism (e.g., the process of compensating a *scope* inside a *while* has to invoke the instances of the *compensation handler* in each successive iteration in reverse order).

Last but not least, an *event handler* defines message events that can be triggered repeatedly and concurrently during the lifetime of the scope, as well as

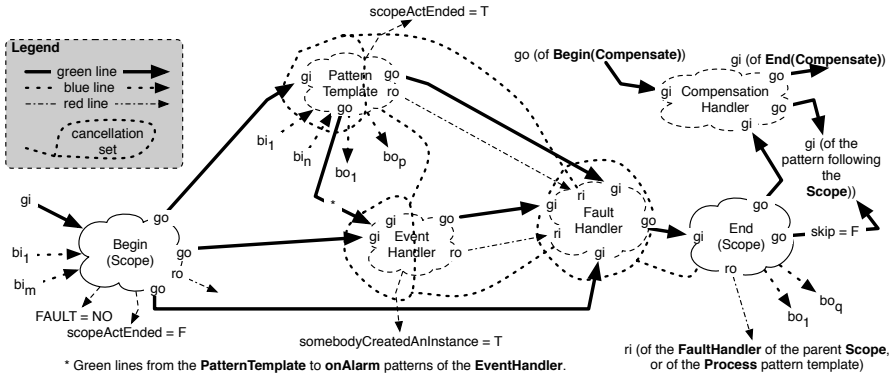


Figure 17: High level view of the *Scope* pattern template.

alarm events that can be triggered at most once while the corresponding *scope* is active. The former is different with respect to *pick* activities, which allow only one message event to take place. Note however that when the activity of a *scope* finishes its execution, the activities running inside an *event handler* are allowed to complete, yet no other message or alarm events may be triggered. Furthermore, the messages received by the *event handler* cannot start a business process instance (since the event handler cannot be enabled until the instance is created), as well as the use of the *compensate* activity inside *event handler*s is prohibited.

The *Scope* pattern template has the structure:

$$\mathsf{Scope} \rightarrow \mathsf{Begin(Scope)}\ \mathsf{PatternTemplate}\ \mathsf{FaultHandler} \\ [\mathsf{CompensationHandler}]\ [\mathsf{EventHandler}]\ \mathsf{End(Scope)}$$

and the structural dependencies among the various patterns involved are illustrated in Figure 17. Begin(Scope) (constructed similarly to Begin(Flow)) sends green tokens to the Pattern Template translating the (child) activity of the *scope*, to the EventHandler, and to the FaultHandler. Furthermore, the *BeginScope* task of Begin(Scope) sets the *FAULT* scope variable to *no* and the *scopeActEnded* to *false*. The former is used by the Scope pattern (and in particular by the FaultHandler) to deal with faults being raised inside the BPEL *scope*, while the latter serves for knowing when the activity defined by the *scope* terminates so that the EventHandler knows when to finish its execution.

The Pattern Template implementing the scope's activity (see Figure 18) has to set the *scopeActEnded* variable to *true* if the *scope* is executed (viz., not skipped), and it forwards at least one green token. A green (skipping) token is always sent to the *GreenGate0* task of the Begin(FaultHandler), while further green tokens are sent to the Pattern Templates belonging to onAlarm patterns (see the EventHandler below), if and only if the scope's activity was successfully executed (viz., *scopeActEnded* is set to *true*). The first one is used to achieve

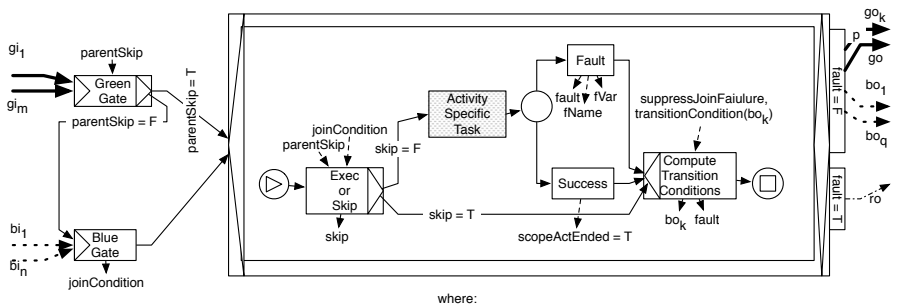


Figure 18: The Pattern Template (or Begin(activity)) translating the *scope*'s activity.

the dead-path-elimination inside the FaultHandler, while the other tokens are used to unlock the onAlarm patterns of the EventHandler after cancelling the respective timers.

After receiving a green token from Begin(Scope), the FaultHandler pattern template further receives either one green token from the Pattern Template (of the scope's activity) and one green token from the EventHandler (if any), or one red token from the Pattern Template or from the EventHandler. In the former case, the entire FaultHandler will be skipped either because the Pattern Template was completed successfully, or because the entire Scope has to be skipped. The latter case corresponds to a fault being raised (and uncaught) inside the Pattern Template, or inside the EventHandler. If the fault cannot be processed, the FaultHandler sends a green token to End(Scope), which has to output a red token further to the FaultHandler of the parent Scope pattern (if any), or to the FaultHandler of the Process pattern template. Note that only the FaultHandler forwards a (green) token to End(Scope). Furthermore, the FaultHandler pattern template outputs the *FAULT* variable (as we shall see later), while the EventHandler inputs the *scopeActEnded* and outputs a *true* value for the *somebodyCreatedAnInstance* variable. When the FaultHandler catches a fault it clears the tokens of the Pattern Template corresponding to the child activity of the scope, as well as the tokens of the activities defined by the *event handler*.[11]

End(Scope) (built similarly to End(Sequence)) is in charge of enabling the CompensationHandler when the Pattern Template translating the scope's activity is executed successfully. (Note that End(Scope) has to save a copy of all the scope variables as required by the *CompensationHandler* [2].) If the Scope is

[11] Note that in order to simplify the translation methodology, we do not treat differently the termination of the BPEL activities, as the BPEL semantics of activity termination [2] notes. Instead, we simply clear all tokens of the pattern corresponding to the activity enclosed by the *scope*.

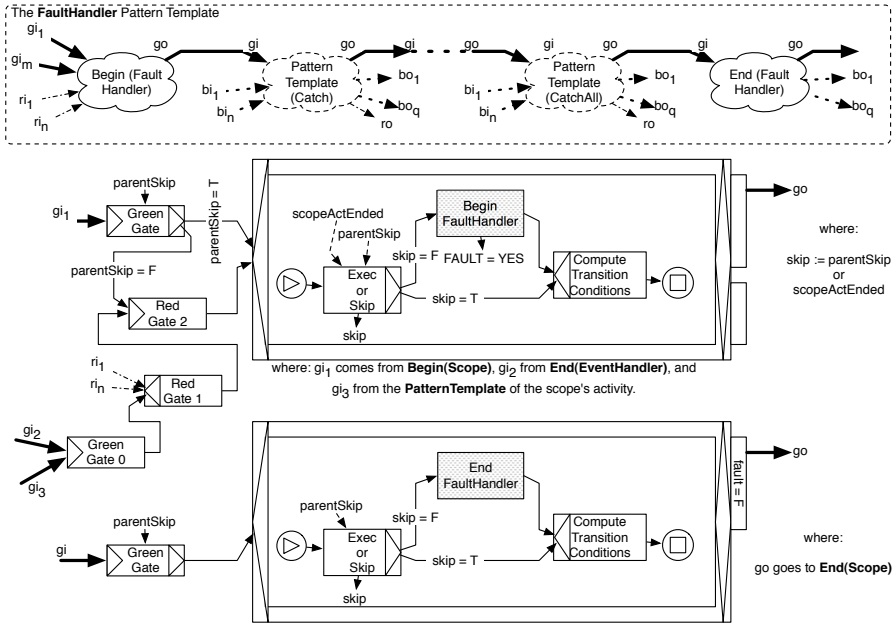


Figure 19: The FaultHandler pattern template.

skipped, End(Scope) has to clear the green tokens received by the FaultHandler from the Pattern Template and from the EventHandler as they are redundant due to the fact that the skipping green token sent by Begin(Scope) to the FaultHandler pattern reaches it first. Dually, the cancellation set of End(Scope) should clear red tokens stuck at the Begin(FaultHandler) pattern in case multiple faults reach it before it is executed. Furthermore, in this case it is unnecessary to perform the dead-path-elimination inside the EventHandler as links cannot cross its boundary. However, we do have to perform the dead-path-elimination inside the FaultHandler.

The FaultHandler pattern has a similar structure to the Sequence pattern (see Figure 19):

Begin(FaultHandler) PatternTemplate* End(FaultHandler)

except that each Pattern Template that corresponds to a *catch* activity has a guard condition checking the fault name and data (see Figure 20). The *GreenGate* task of the (*catch*/*catchAll*) Pattern Template has to check first if the *FAULT* variable is set to *yes*. Furthermore, for Catch patterns, it defines *myFaultName* and *myFaultVar* variables corresponding to the *faultName* and *faultVariable* attributes defined by the *catch* of the business process to be translated, as well as it uses the *faultName* and *faultVar* global (viz., EWF-net) variables that are set by patterns generating errors (e.g., Throw, Receive).

As previously mentioned, the match of the fault name and variable is done

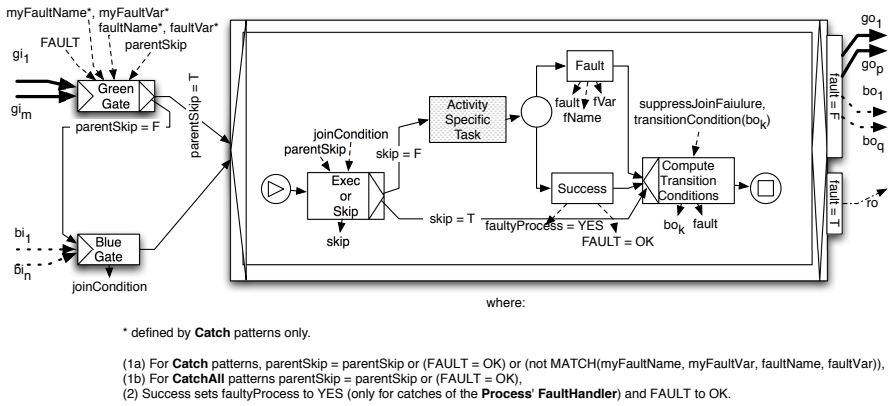


Figure 20: The Pattern Template translating a *catch/catchAll* activity.

with respect to the following rules [2]:

- A *catch* defining *myFaultName* only matches faults with *faultName*.
- A *catch* with *myFaultVar* only, matches faults with *faultVar* and any *faultName*.
- A *catch* defining both *myFaultName* and *myFaultVar* matches faults with the respective *faultName* and *faultVar*, and
- A *catchAll* matches all faults. (Hence the *GreenGate* task of the pattern template translating the activity of a *catchAll* does not need the guard.)

The patterns corresponding to the *catch/catchAll* activities are linked in a sequence for two reasons. A fault received by the Begin(FaultHandler) pattern is passed on to the first Catch pattern. If it cannot process the fault (as it does not match it), then the respective Catch is skipped (i.e., dead-path-elimination is employed) and the fault is passed on to the second Catch, and so on. However, if a Catch matches the fault name and data (if any), then the green token which is passed on to further Catch/CatchAll patterns achieves the dead-path-elimination. Hence, the sequencing allows, on the one hand, matching the first Catch that can process the fault (as in the BPEL process), and on the other hand, employing the dead-path-elimination inside the FaultHandler. For a Catch/CatchAll pattern (see Figure 20) its *Success* task (or the *ActivitySpecificTask* if the *Success* task is not defined) has to set the *FAULT* variable to *ok* as the fault has been caught and processed successfully. However, if the *catch/catchAll* pattern cannot process the error, as well as if the *FAULT* is set to *no* or *ok* then the pattern should be skipped.[12]

[12]The modifications brought here to the pattern template of a *catch/catchAll* activity are illustrated on the Basic Pattern Template, which has to be used if the respective activity is a

Please note as well, that an error generated by an activity inside a BPEL (*catch*/*catchAll* of a *fault handler* has to be signalled to the *fault handler* of the enclosing *scope* (or *process*), and hence the red output of the activity's pattern should be connected to the Begin(FaultHandler) of the parent *scope* of this current *scope* (or of the Process pattern if no parent *scope* exists).

Furthermore, Begin(FaultHandler) uses a *RedGate* (instead of a *BlueGate*) that waits for red tokens to be sent (viz., faults to be raised) from inside the Pattern Template (or from inside the EventHandler) of the *scope*'s activity. In order to interrupt the normal execution of the scope in case of a fault being raised, the *RedGate* uses a cancellation set that includes all patterns of the Pattern Template translating the scope's activity and EventHandler except the CompensationHandler patterns corresponding to scopes nested in its scope.

Begin(FaultHandler) inputs three green lines (see Figure 19): (1) from Begin-Scope, (2) from End(EventHandler), and (3) from the Pattern Template translating the *scope*'s activity. Furthermore, its successful execution sets the *FAULT* variable to *yes*, as a fault has been raised. It is important to note that if the Scope is skipped, then Begin(Scope) sends a green (skipping) token to Begin(FaultHandler) (see gi_1 in Figure 19). Still, two more green tokens can arrive at Begin(FaultHandler) (see gi_2 and gi_3 in Figure 19) from the End(EventHandler) (if any) and from the Pattern Template of the *scope*'s activity. This last two redundant green tokens are to be cancelled finally by End(Scope). If the *scope* activity terminates (viz., the *scopeActEnded* variable is set to *true*), the FaultHandler is skipped and dead-path-elimination is employed inside it.

Finally, if the BPEL process does not define a *fault handler*, the translator generates a default FaultHandler pattern consisting of Begin(FaultHandler) and End(FaultHandler) only, linked in a sequence. In this way, the faults received by this default FaultHandler will be forwarded (through EndScope) to the FaultHandler of the parent scope (or the one associated to the entire business process).

In the pattern of the *event handler* (Figure 21):

$$\text{Begin(EventHandler)} \quad \text{PatternTemplate}^{+} \quad \text{End(EventHandler)}$$

the Pattern Templates execute concurrently. On the one hand, the patterns of *onMessage* activities are placed in a loop with a guard that checks the end of the Pattern Template translating the activity inside the scope. On the other hand, the patterns of *onAlarm* activities are executed at most once as an alarm event is carried out at most once while the corresponding scope is active.

The Begin(EventHandler) pattern template of the outermost scope (or of the Process pattern) (Figure 22) has to wait for a green (enabling) token from a Receive or Pick onMessage, whose *createInstance* attribute is set to *yes*. Furthermore, Begin(EventHandler) outputs green tokens for all the onMessage and onAlarm patterns in order to enable them.

basic one (with the exception of the BPEL *assign* and *compensate*). Otherwise, the respective modifications are to be made to the Begin and/or End patterns of the corresponding Pattern Template translating the structured activity.

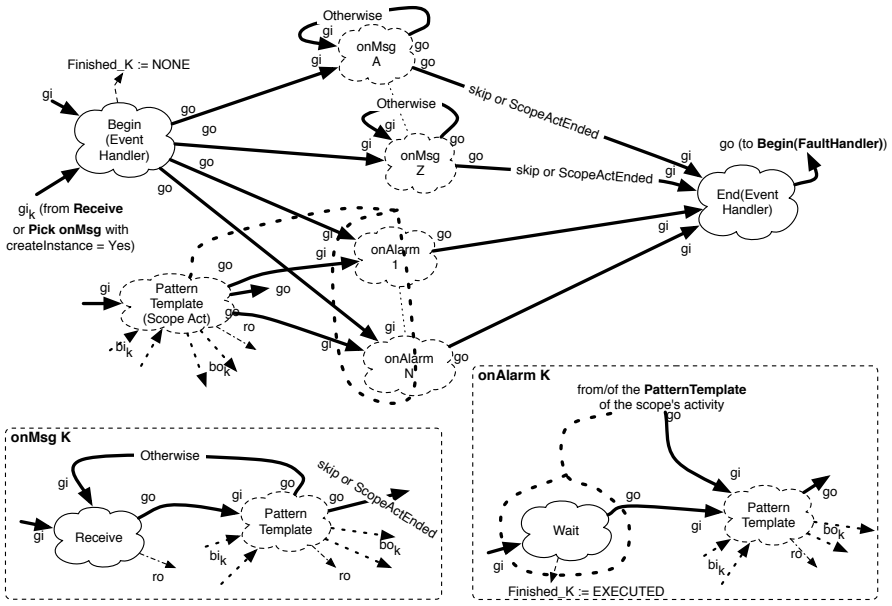


Figure 21: The EventHandler pattern template.

Each onMessage pattern (bottom-left part of Figure 21) is composed by a Receive-like pattern and by the actual Pattern Template translating the activity defined in the BPEL *onMessage*. In order to execute the Receive pattern either after the Begin(EventHandler) (viz., the first time the respective onMessage is executed), or after the Pattern Template (viz., successive executions of the respective onMessage pattern), its *GreenGate* employs a XOR-join. Both patterns of the onMessage have to check whether the scope's activity has ended (viz., *scopeActEnded* is *true*). On the one hand, the Receive does the check in its *GreenGate* task. If the scope's activity ended before the Receive, then both patterns are skipped. On the other hand, the Pattern Template does the check when outputting green tokens (see the respective predicate in Figure 21). We do so because the scope's activity may have finished after executing the Receive, and in this case, events that are running are allowed to finish their execution.

All onAlarm patterns (bottom-right part of Figure 21) are enabled by the Begin(EventHandler) pattern and they can be executed at most once. Each one consists of a Wait, linked in a sequence with a Pattern Template. The Wait implements the timer and if it finishes its execution, the Pattern Template translating the *onAlarm* activity gets executed. Note that the pattern of the scope's activity cancels the Wait timer and (in order not to lock the workflow) it forwards a green (skipping) token to the *onAlarm* Pattern Template. Similarly to a Receive for the onMessage pattern, the Pattern Template here employs a XOR-join for its *GreenGate*. However, the decision on whether to execute the

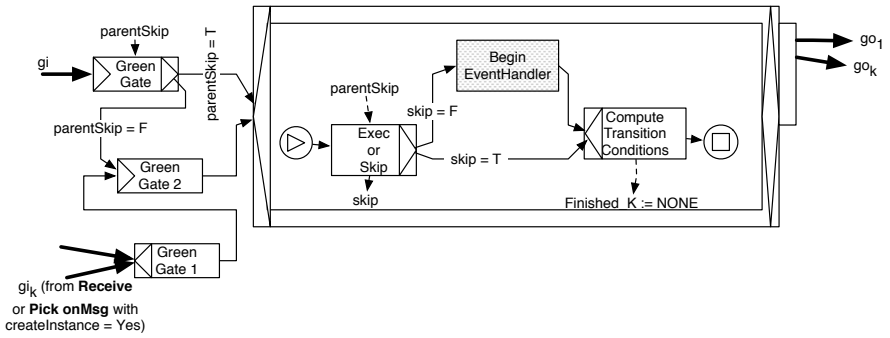


Figure 22: The Begin(EventHandler) pattern template.

Pattern Template is based on a $Finished_K$ variable (whose initial *none* value is given by the Begin(EventHandler) pattern, and further set to *executed* by the *ComputeTransitionConditions* of the Wait timer), and not on the the status of the *scopeActEnded* variable. We do so because the Pattern Template should be executed if and only if the Wait timer was executed successfully. Otherwise, it should be skipped.

Finally, the CompensationHandler pattern (Figure 23) consists of:
Begin(CompensationHandler) PatternTemplate* End(CompensationHandler)
If the scope completes successfully, the Begin(CompensationHandler) is activated and waits for a green token from a Begin(Compensate) pattern (see next). The green output of Begin(CompensationHandler) enables the Pattern Template implementing the actual compensating activity, which forwards (on its termination) the green token to End(CompensationHandler). Upon completion, the End(CompensationHandler) issues only one green token on one of its two green outputs. If the Scope is skipped, End(CompensationHandler) sends a green token to the pattern translating the activity (structurally) following the *scope* in the BPEL process. Otherwise, it sends a green token to the End(Compensate) pattern (see next). Note that if a BPEL *scope* does not define a *compensation handler* yet there is a *compensate* activity targeting the respective scope, the translator generates a default CompensationHandler consisting only of Begin(CompensationHandler) directly linked to End(CompensationHandler).

Compensate. The BPEL *compensate* has the following structure:

```
<compensate scope="ncname"? standard-attributes>
      standard-elements
</compensate>
```

The *compensate* activity serves for triggering roll-back activities as a result of e.g., faults occurring in the business process. We recall that specifying the name of the scope to be compensated leads to invoking the *compensation handler* of the respective *scope*. Otherwise, the default compensation mechanism is triggered,

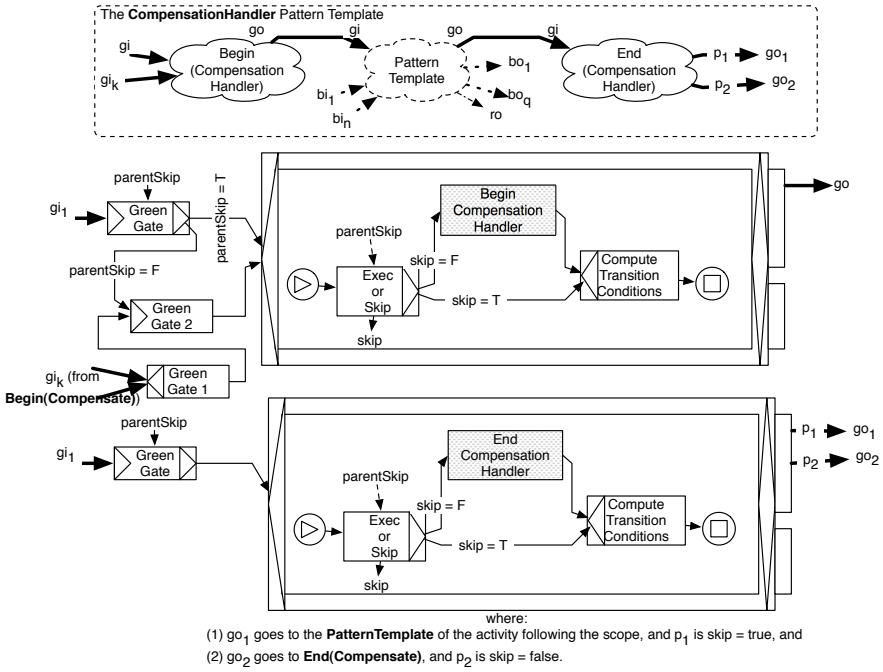


Figure 23: The CompensationHandler pattern template.

and it (roughly) involves the invocation of the the compensation handlers for the immediately enclosed scopes in the reverse order of the completion of these scopes. [13]

The pattern template corresponding to the BPEL *compensate* is given in Figure 24:

Begin(Compensate) End(Compensate)

since *compensate* terminates only when the invoked CompensationHandler finishes its execution. Recall that we consider only explicit compensation, that is *compensate* activities specifying the name of the scope to be compensated, and furthermore, without considering *scope*s nested inside *while* activities. Begin(Compensate) sends a green token directly to End(Compensate) if the *compensate* is skipped, or if the scope to be compensated did not finish its execution. Otherwise, the green token is sent to the Begin(CompensationHandler) of the scope to be compensated. Dually, End(Compensate) receives a green token either directly from Begin(Compensate), or from the End(CompensationHandler) of the scope to be compensated.[14] Then, it forwards it to the pattern structurally

[13]Note that a *compensate* may only be used inside the *fault handler* or the *compensation handler* of the scope that immediately encloses the scope to be compensated [2].

[14]Note that each Compensate should have an identification so that the End(CompensationHandler) can know which Compensate pattern invoked it.)

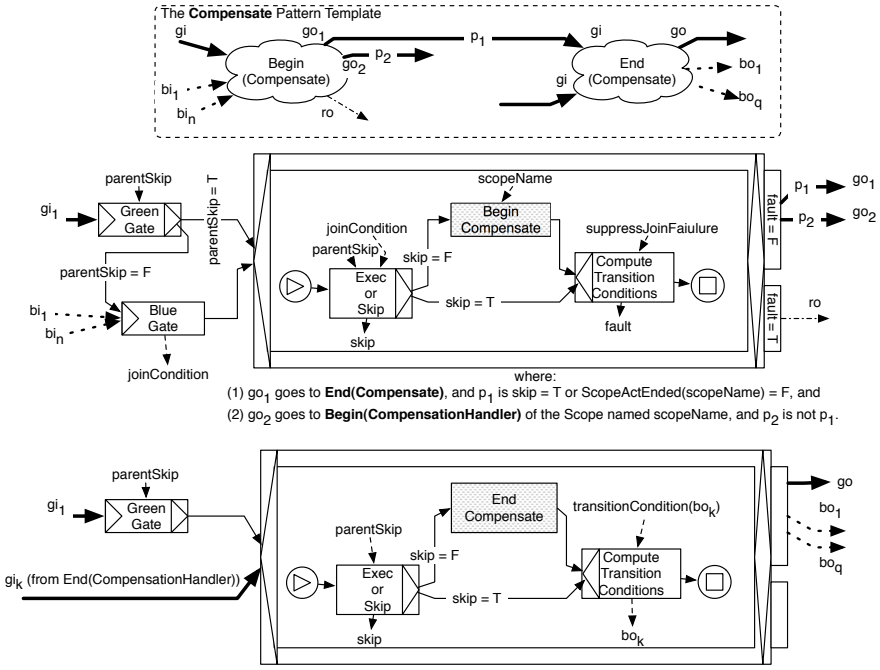


Figure 24: The Compensate pattern template.

following the Compensate.

3.3 BPEL processes

A BPEL *process* encapsulates the process activity and it can further define a *fault handler*, a *compensation handler*, as well as an *event handler*, similarly to a *scope*. Its structure is the following:

```
<process name="ncname" targetNamespace="uri"
         queryLanguage="anyURI"?
         expressionLanguage="anyURI"?
         suppressJoinFailure="yes|no"?
         enableInstanceCompensation="yes|no"?
         abstractProcess="yes|no"?
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/" >

    <partnerLinks>?
        < partnerLink name="ncname" partnerLinkType="qname"
              myRole="ncname"? partnerRole="ncname"?>+
        </partnerLink>
    </partnerLinks>

    <partners>?
        <partner name="ncname">+
            <partnerLink name="ncname"/>+
        </partner>
    </partners>
```

```
    <variables>?
        <variable name="ncname" messageType="qname"?
            type="qname"? element="qname"?/>+
    </variables>

    <correlationSets>?
        <correlationSet name="ncname" properties="qname-list"/>+
    </correlationSets>

    <faultHandlers>?
        <catch faultName="qname"? faultVariable="ncname"?>*
            activity
        </catch>
        <catchAll>?
            activity
        </catchAll>
    </faultHandlers>

    <compensationHandler>?
        activity
    </compensationHandler>

    <eventHandlers>?
        <onMessage partnerLink="ncname" portType="qname"
                operation="ncname" variable="ncname"?>
            <correlations>?
                <correlation set="ncname" initiate="yes|no"?>+
            <correlations>
            activity
        </onMessage>
        <onAlarm for="duration-expr"? until="deadline-expr"?>*
            activity
        </onAlarm>
    </eventHandlers>

    activity
</process>
```

Differently from the *scope*, a *process* may define *partnerLink*s and *partners* of the business process. The former represents a conversational relationship between two partner processes, while the latter represents the capabilities of a partner service as a subset of the partner links of the process. Furthermore, faults that reach the (possibly default) *fault handler*s lead to an abnormal termination of the business process (similarly to a *terminate*), even if they are processed successfully. Since the *compensation handler* is installed only after the successful termination of the activity defined by the process, a business process instance can be compensated only by platform-specific means. Note that in order to allow such compensation the *enableInstanceCompensation* process attribute has to be set to *yes*.
The Process pattern (Figure 25):

Begin(Process) FaultHandler [EventHandler] PatternTemplate End(Process)
resembles very much the Scope pattern, although there are some differences between the two, presented hereafter. For example, Begin(Process) and End(Process) have to be connected to the *input condition* and to the *output condition*, respectively, of the workflow. Begin(Process) enables the Pattern Template, the FaultHandler, as well as the EventHandler (if any), and it is in charge of setting the initial *false* values of the *somebodyCreatedAnInstance* and *faultyProcess*

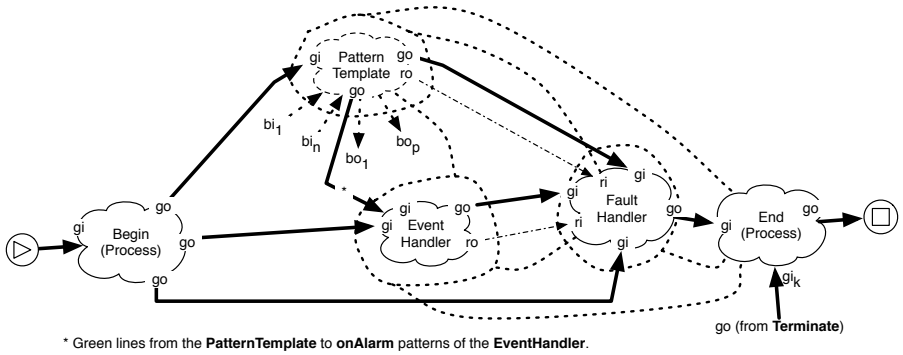


Figure 25: The Process pattern template.

variables. The former is set to *true* by the execution of a Receive or of a Pick onMessage with a *createInstance* variable having a *yes* value, while the latter is set to *true* if a fault is caught by the process FaultHandler. Due to the fact that the BPEL *process* cannot have a parent activity, the *GreenGate* task of its Begin(Process) pattern simply inputs an always *true parentSkip* variable. Dually, as the activity defined by the process cannot be skipped, Begin(Process) outputs an always *true parentSkip*.

If the BPEL process does not define a *fault handler*, or if it does but it does not contain a *catchAll* clause, one (default) FaultHandler with a default catchAll (viz., an *Empty* pattern) must be defined in the Process pattern. This is needed to catch all uncaught faults being raised within the process. Note that the reception of a fault by the process FaultHandler leads to an abnormal process termination, even if the fault is processed. Furthermore, faults being raised (and uncaught) inside the process FaultHandler lead to the immediate execution of the End(Process) pattern, as in the case of a Terminate (see next).

The EventHandler is active for the entire process lifetime and the Pattern Template of the activity defined by the process is in charge of clearing its tokens upon its completion, similarly to a Scope. In order to minimise the number of cancellation sets defined in the workflow, all Terminate patterns forward the green token to End(Process), which is in charge of immediately terminating the entire business process. It does so by clearing all the tokens of the Pattern Template corresponding to the activity defined by the process. Hence, End(Process) is enabled if it receives either a green token from a Terminate, or from the process FaultHandler.

As previously mentioned, the *compensation handler* can only be invoked by platform-specific means. However, we do not consider a CompensationHandler pattern for the entire business process, since YAWL does not allow for such invocation mechanism. (Note also that the CompensationHandler pattern of the process would block the workflow waiting for a green token.)

A BPEL process is translated into a YAWL workflow by instantiating the

Process pattern. This leads to recursively instantiating the Begin(Process), FaultHandler, EventHandler (if any), and End(Process) patterns, as well as the Pattern Template corresponding to the activity defined by the BPEL *process*. Note that the instantiation of a pattern takes into account the context in which the activity is placed inside the BPEL process. Namely, instantiating a pattern means adjusting the (number of) input and output lines, setting and mapping the inputs and outputs of the tasks in the pattern, as well as suitably interconnecting its child patterns. The instantiating process bottoms-out at *basic pattern template*s. More information on how to instantiate the pattern templates is given in the next Section, which illustrates the YAWL workflow one obtains by translating a simple BPEL process.

4 Example: Complete translation of a (simple) BPEL process

In this Section we shall describe the translation of the Greatest Common Divisor (GCD) BPEL process introduced in Section 2. We recall that the GCD process computes the greatest common divisor of two numbers by repeatedly raising an exception while one of the two numbers is bigger than the other and by decreasing its value in the corresponding catch. (Note that, for space issues, the GCD process uses a simplified notation of the activities.)

In the following, we first describe in detail the generation of the YAWL workflow translating the GCD process (Subsection 4.1), followed by an execution scenario of the obtained GCD workflow (Subsection 4.2).

4.1 Generation of the GCD Workflow

Figure 26 gives the high-level view of the YAWL workflow obtained from the GCD process, while a more detailed view of the GCD workflow is presented in Figure 27.[15]

Roughly, the BPEL2YAWL translator inputs the GCD process, it parses it and produces the corresponding GCD YAWL workflow as described hereafter. However, please note that for space issues we cannot depict each step of the translation in a separate figure. Furthermore, we shall not give an in-depth description of the pattern instances (e.g., mapping of all the task variables) in order to keep the discussion comprehensible.

Step 1: Instantiating the Process pattern template

The translation starts with a GCD workflow consisting only of the input and output conditions. Initially, the translator generates the Begin(Process) and End(Process) patterns, and it suitably connects them to the input and output

[15]The full BPEL process and the YAWL workflow of the example can be downloaded from: `http://www.di.unipi.it/~popescu/GCD_Example.zip`.

conditions of the GCD workflow (see Figure 26, top-left and top-right, respectively).

On the one hand, Begin(Process) sets the global (viz., EWF-net) variable *suppressJoinFailure* to *yes*, as well as it defines and sets the *name* input variable of its *BeginProcess* task to *GCD*, as defined in the <*process*> element of the BPEL process. Furthermore, it also maps a *true* boolean value into a (first) *parentSkip* EWF-net variable, which is to be inputted by both patterns translating the activity and the fault handler of the business process. (Although we shall not refer to indexes when discussing about e.g., *parentSkip* variables, the readers should note that each pattern instance that translates the beginning of a structured activity outputs a new *parentSkip* variable, which is to be inputted by the rest of the patterns translating the respective structured activity.) Begin(Process) is represented in Figure 27 (top-left) by the *GreenGate* atomic tasks and by the *BeginProcess* composite task. Note that the latter is in charge of mapping the IOs previously mentioned.

On the other hand, End(Process) is instantiated by default, that is, to an Empty-like pattern, without any significant inputs and/or outputs. A more detailed view of End(Process) reveals the *GreenGate* as well as the *EndProcess* tasks in Figure 27 (top-right).

BPEL2YAWL continues next by (recursively) translating the *fault handler* as well as the *flow* activity of the GCD process. As we shall see next, these two patterns are enabled by the green outputs of *Begin(Process)*.

Step 2: Instantiating the *process*' FaultHandler

The *fault handler* of the GCD process defines a *catch* that processes *negNum* faults. The activity of the *catch* is a *reply*, which forwards to the invoker of the business process the respective fault. Consequently, BPEL2YAWL generates a FaultHandler pattern instance consisting of a Begin(FaultHandler), End(FaultHandler), as well as one *Reply* and one *Empty* patterns, all linked sequentially by green lines as seen in Figure 26 (bottom-left). In Figure 27 (centre-left) one may see that Begin(FaultHandler) consists of the *GreenGate*, *RedGate2*, *RedGate1*, *GreenGate0*, and *BeginFaultHandler* tasks. The *GreenGate* task inputs the green output of the Begin(Process), as the initiation of the business process structurally enables the *fault handler*. Furthermore, as we shall see later, *RedGate1* serves for catching red tokens representing faults raised in the process (e.g., by the *throw* activity in the *flow*), while *GreenGate0* serves for inputting the green token of End(Flow), which enables and skips the FaultHandler if the process' activity (viz., the *flow*) completes successfully. Another characteristic of the *RedGate1* task consists of its cancellation set that will include the entire Flow pattern. The purpose of this cancellation set is to interrupt the execution of the activities inside the *flow* when the process' *fault handler* receives an error. Furthermore, BPEL2YAWL generates another cancellation set associated to the End(Process) pattern, which includes the *RedGate2*, *RedGate1*, and *GreenGate0* tasks of the Begin(FaultHandler) pattern. We recall that this is useful in order to cancel redundant red tokens if e.g., multiple failures reach Begin(FaultHandler) prior to

its execution, as well as redundant green tokens due to skipping the Scope. For simplicity, we represent this cancellation set in Figure 26 as including the entire Begin(FaultHandler) pattern. However, we do not represent the cancellation set in Figure 26 in order not to burden further the workflow. Please note that, in order to keep the translation manageable, we do not differentiate between activities that should be allowed to complete (e.g., *assign*s) and activities that are to be interrupted. We simply interrupt all running activities by removing all tokens of the pattern translating the *scope*'s activity. Begin(FaultHandler) forwards a green token to the Reply pattern, represented in Figure 26 by the *GreenGate* and *Reply* tasks. Since the *reply* in this scenario is a catch activity, the *GreenGate* of its Reply pattern is guarded by the *fault* = *negNum* boolean expression (see the G2 comment in Figure 27). (In order to ease the presentation we refer here and in the Figures to the "core" of the *catch* guard that also takes into account the *parentSkip* and *FAULT* variables, as defined in Section 3.) As a consequence, at run-time of the GCD workflow, if the process' *fault handler* receives a fault due to e.g., a message mismatch in the *receive*, the guard of the Reply pattern will evaluate to *false*, and hence the Reply will be skipped. However, as the *fault handler* of a BPEL business process has to catch all unprocessed faults in the process, BPEL2YAWL generates the pattern of a default *catchAll* represented by the Empty pattern immediately following the Reply. Note that, in order to suppress all uncaught faults, the *GreenGate* of the Empty pattern simply does not employ a guard testing the match with the *faultName* variable. Finally, the Empty pattern forwards a green token to End(FaultHandler), which similarly to End(Scope) is instantiated by default. As indicated in Figure 27 (centre-right) End(FaultHandler) outputs a green token for the *GreenGate* of theEnd(Process) pattern. Roughly, we recall that the FaultHandler is either executed in case of a failure, or skipped if the process' activity completes successfully, and in both such cases the FaultHandler is immediately followed by the termination of the business process.

Step 3: Instantiating the *process*' Flow

The activity defined by the GCD process is a *flow* and, as a result, the translator generates instances of the Begin(Flow) and End(Flow) patterns, and suitably links them to the Begin(Process) (Figure 26 and 27, top-left) and Begin(FaultHandler) (Figure 26 and 27, top-right), respectively. Furthermore, both patterns are instantiated by default, as the *flow* activity is included in a "simple context" (i.e., it is not subject to any guard, or it does not have any incoming or outgoing links). The *flow* consists of four activities – a *receive*, a *throw*, a *while*, and a *sequence*. The Receive pattern instance generated by BPEL2YAWL is composed of a *GreenGate* task and of a *Receive* task (see Figure 27, top-centre). The former structurally enables the *Receive* task (when the Flow is executed) by inputting the green output of Begin(Flow), while the latter forwards a green token to End(Flow) on its completion. Furthermore, the translator generates for the *Receive* a red output line that targets the *RedGate1* task of the process Begin(FaultHandler) pattern, necessary for signalling faults raised by e.g., mis-

matches in the input message of the *receive*. (We recall that such a failure is generated by the execution of the *Fault* task of the *Receive* composite task.)

The second activity in the *flow* is a *throw* which results in the Throw pattern given in the top-centre part of Figure 27. Similarly to the Receive it employs a *GreenGate* task and it is connected by green lines to Begin(Flow) and End(Flow). However, it also defines a *BlueGate* as the *throw* activity in the BPEL process is target of a synchronisation link having the *receive* as source. Consequently, the translator generates a blue line that suitably links the *Receive* composite task to the *BlueGate* of the Throw pattern. Moreover, the YAWL (boolean) predicate of the blue link is given by the (boolean) transition condition defined by the respective synchronisation link in the GCD process (see the TC2 annotation in Figure 27). Furthermore, since the activity being translated is a *throw*, BPEL2YAWL adds a red line linking the *Throw* composite task of the Throw pattern to the *RedGate1* task of the Begin(FaultHandler) pattern (belonging to the process' FaultHandler). Through this red line the Throw pattern interrupts the normal execution of the GCD workflow if one of the two numbers inputted by the Receive is negative or zero. (Note that the Throw pattern cannot raise a *joinFailure* since, even if it has to be skipped when one of the two inputted numbers is negative or zero, the corresponding *suppressJoinFailure* variable is set to *YES* for the whole process.)

Now, because the remaining two activities of the *flow* are structured ones, we shall describe them in further separate steps.

Step 4: Instantiating the While pattern template

The *while* activity initially leads to the generation of the Begin(While) (Figure 26 centre-left) and End(While) (Figure 26 centre-right) patterns. As shown in the centre-left part of Figure 27, Begin(While) consists of three tasks – a *GreenGate*, a *BlueGate*, as well as a *BeginWhile*. The *GreenGate* structurally enables the Begin(While) pattern by inputting the green output of Begin(Flow). The *BlueGate* task inputs the blue output of Receive and, at run-time, it decides whether to skip or to execute the While pattern. If both numbers inputted by the Receive are strictly positive (viz., the TC1 predicate in Figure 27 evaluates to *true*), the *Receive* task outputs a blue token that leads to the execution of the While. Otherwise, the blue token leads to skipping the While. Apart from the green inputs given by the *GreenGate* and *BlueGate*, the *BeginWhile* task inputs one more green tokens from End(While), which serves for re-cycling. Moreover, it employs one green output that enables its *scope* child activity. Furthermore, the *ExecOrSkip* task of *BeginWhile* employs a guard (corresponding to the guard of the *while* activity in the GCD process) that checks whether the two numbers inputted by the Receive are equal. At run-time, if the guard holds true the While pattern will be skipped, otherwise it will be executed.

The End(While) pattern (Figure 27 centre-right) is instantiated by default. It has a *GreenGate* that will have to input the green output of the pattern translating the *scope* (child) activity of the *while*, as well as a *EndWhile* task that outputs two green tokens – one for the *BeginWhile* task (guarded by the

TC3 boolean predicate in Figure 27), and another for the *GreenGate* of the End(Flow) pattern (guarded by the TC4 boolean predicate in Figure 27).

Step 5: Instantiating the Scope pattern template

The *scope* encloses a *switch* activity and it defines a *fault handler* as well. Consequently, BPEL2YAWL creates instances of the Begin(Scope) and End(Scope) patterns, and it suitably links them to Begin(While) (Figure 26, centre-left) and End(While) (Figure 26, centre-right), respectively. On the one hand, Begin(Scope) consists of a *GreenGate* task that enables the Scope, as well as of a *BeginScope* task that will have to enable both the Switch and *scope*'s FaultHandler pattern instances (Figure 27). On the other hand, End(Scope) employs a *GreenGate* that has to wait for the green token from the *scope*'s FaultHandler, and an *EndScope* task that will forward it to the *GreenGate* of the EndWhile pattern. Furthermore, *EndScope* outputs a red line that targets the *RedGate1* task of the process Begin(FaultHandler) in order to forward to it exceptions caught yet unprocessed by the scope FaultHandler. BPEL2YAWL continues next with the translation of the *scope*'s *fault handler* and of the *switch* activity.

Step 6: Instantiating the *scope*'s FaultHandler pattern template

The *fault handler* consists of two *catch*es each one wrapping an *assign* activity. As a result, the translator instantiates a Begin(FaultHandler), two Assign patterns, as well as an End(FaultHandler), all linked in a sequence. Begin(FaultHandler) is similar to the Begin(FaultHandler) pattern of the process' FaultHandler (Figure 27 bottom-left). The *GreenGate* task inputs the green token of *BeginScope*, while the *RedGate1* and *GreenGate0* input red and green tokens, respectively, of the Switch pattern enclosed in the Scope. The role of the BeginFaultHandler is to enable the pattern of the first Catch in the FaultHandler. Moreover, its *RedGate1* task defines a cancellation set in charge of interrupting the Switch pattern of the Scope. Dually, EndFaultHandler has a *GreenGate* that receives the green token from the last (viz., second) Catch in the FaultHandler, as well as an *EndFaultHandler* task that has to send the green token to the *GreenGate* of the End(Scope) pattern. Note that BPEL2YAWL also generates a cancellation set associated to the End(Scope) pattern, which includes the *RedGate2*, *RedGate1*, and *GreenGate0* tasks of the *scope*'s Begin(FaultHandler) so as to cancel redundant red and green tokens that might get stuck due to e.g., multiple (simultaneous) failures, or to skipping the Scope pattern. (For simplicity, the cancellation set is represented in Figure 26 as including the entire Begin(FaultHandler) pattern.)

Next, both *catch*es translate to Assign patterns composed of Begin(Assign), Copy, and End(Assign) pattern instances (Figure 26, bottom-right). A main characteristic of the two is that Begin(Assign) employs a *GreenGate* task that checks the catch guard. Consequently, the first Assign is executed if the fault name is *dec_a* (see the G6 comment in Figure 27, bottom-centre), while the second Assign is executed provided the fault name is *dec_b* (see the G7 comment

in Figure 27, bottom-centre). Otherwise, the respective *Assign*s are skipped. As indicated in the GCD process, the *Copy* task of the first Copy pattern maps the expression *a - b* into the variable *a*, and similarly, the *Copy* task of the second Copy pattern maps the expression *b - a* into the variable *b*. Furthermore, BPEL2YAWL adds for each of the two *Copy* tasks a red output linking them to the *RedGate1* task of the process' Begin(FaultHandler) in order to forward to it (possible) faults due to assignment issues (e.g., parameter types mismatches).

The translation continues next with the *switch* activity of the *scope*.

Step 7: Instantiating the *scope*'s Switch pattern template

The *switch* is translated into a Begin(Switch), two Throws, as well as an End(Switch) pattern, all linked sequentially (Figure 26, centre). Begin(Switch) and End(Switch) are instantiated by default, and include *GreenGate*s that enable the *BeginSwitch* and *EndSwitch* tasks, respectively (Figure 27, centre). Moreover, the former is enabled at the beginning of the *Scope*, while the latter enables the *scope*'s FaultHandler upon (successful) completion of the Switch. Now, since the first *throw* is a *case* branch, its Throw pattern defines a *GreenGate* that checks whether a previous branch was already executed, as well as the branch guard, as defined in the GCD process. As a consequence, the first *Throw* task is executed if and only if $a > b$ (see the G4 comment in Figure 26), while the second one (corresponding to the *otherwise* branch) is executed otherwise. Moreover, both *Throw*s output a red line that signal *dec_a* and *dec_b* faults, respectively, to the *FaultHandler* of the Scope.

Finally, BPEL2YAWL terminates by translating the *sequence* activity of the *flow*.

Step 8: Instantiating the *flow*'s Sequence pattern template

The *sequence* defines two activities – an *assign* followed by a *reply*, and hence it leads to the generation of a Begin(Sequence), an Assign, a Reply, as well as an End(Sequence) pattern (Figure 26 bottom).

Begin(Sequence) (Figure 27 top-left) defines a *GreenGate* that receives a green token from the *BeginFlow* task and which serves for structurally enabling the Sequence, as well as a *BlueGate* that is the target of a blue line from *EndWhile* due to the synchronisation link between the *while* and the *sequence* in the GCD process. Consequently, although Begin(Sequence) is always enabled when it receives both the green and blue tokens, it will be executed only when the status of the synchronisation link is positive (viz., the *joinCondition* computed by the *BlueGate*, which is given by the BPEL *transitionCondition*, holds *true*). Since the synchronisation link does not define a *transitionCondition*, BPEL assumes it to be *true* by default, and hence BPEL2YAWL considers an (always) *true* value for it in the *BlueGate* of Begin(Sequence), for the computation of the *joinCondition*. (We recall that the transition conditions do not translate to YAWL predicates, and hence blue tokens are outputted on blue lines even if

their corresponding transition conditions are *false* (viz., they have negative statuses). However, as just mentioned, each transition condition is mapped onto a EWF-net variable by the source pattern of the link, and it is taken into account by the *BlueGate* of the target pattern when computing the *joinCondition*.) It is important to note that, although the *sequence* is target of a synchronisation link, the *BeginSequence* task of Begin(Sequence) does not output a red line for the *RedGate1* task of the process' Begin(FaultHandler) pattern, since its corresponding *suppressJoinFailure* is set to *yes* and hence, it cannot raise *joinFailure*s. On the other hand, the End(Sequence) pattern (Figure 27 top-right) is instantiated by default; it simply waits for the completion of the Reply pattern, and it forwards the green token to the *GreenGate* of the End(Flow) pattern.

The Assign (Figure 27 top-centre) is constructed similarly to the previous ones defined in the FaultHandler of the Scope (yet in this case there is no boolean guard constraining the execution of the Assign). The *Copy* pattern maps the (input) variable a into an (output) variable c, as given by the respective *copy* tag in the BPEL process. Furthermore, since the mapping might raise faults, the *Copy* task outputs a red line that targets the *RedGate1* task of the process' Begin(FaultHandler) pattern.

Finally, the translator produces an instance of the Reply pattern, consisting of a *GreenGate* and of a *Reply* composite task (Figure 27 top-right), both instantiated by default and linked in the workflow correspondingly. Since the *reply* activity may raise errors, the *Reply* task is linked to the *RedGate1* task of the *process*' Begin(FaultHandler) pattern through a red line.

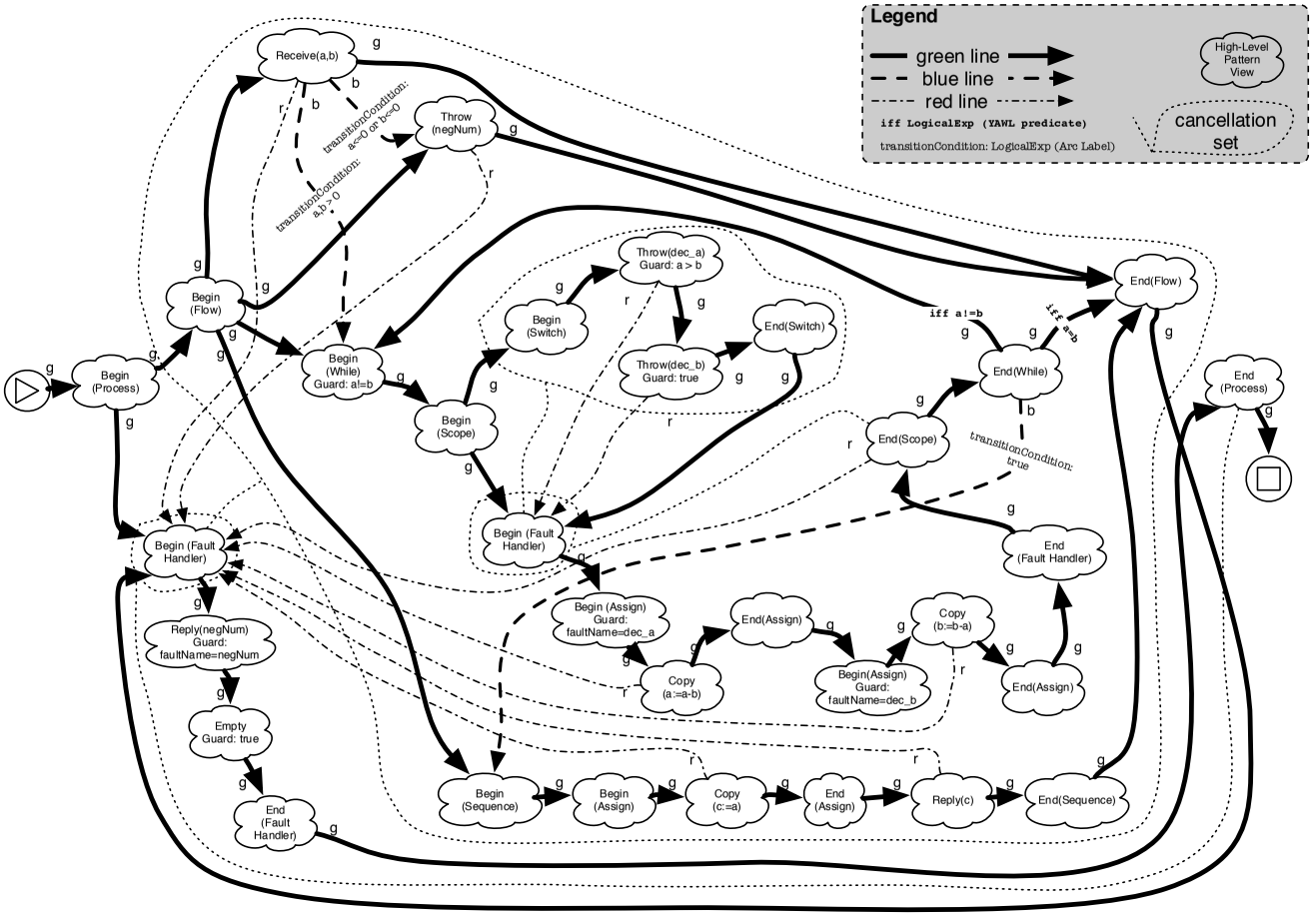


Figure 26: High-level view of the YAWL workflow translating the GCD BPEL process.

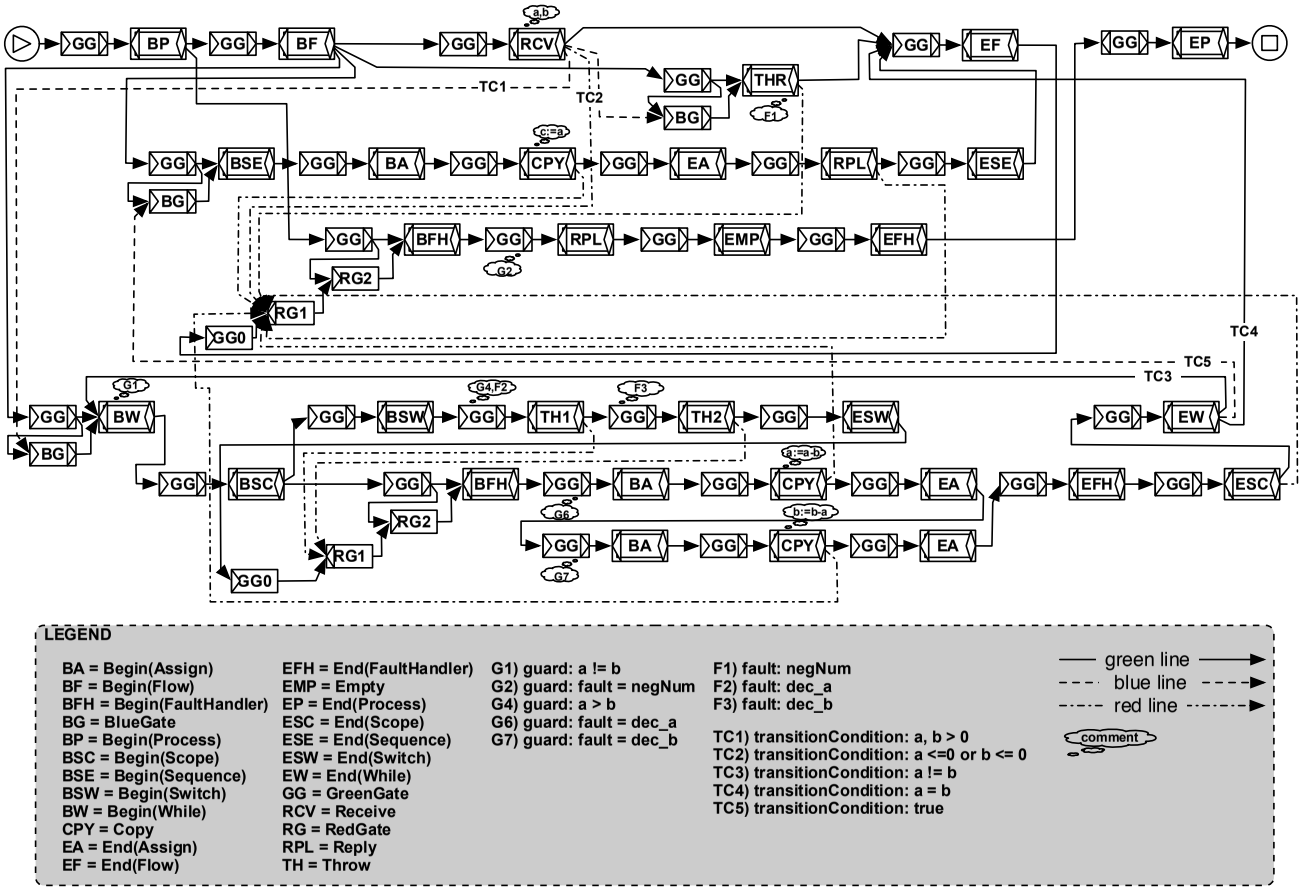


Figure 27: Detailed view of the YAWL workflow translating the GCD BPEL process.

4.2 Use Case of the GCD Workflow

Consider now an execution scenario in which the two input variables a and b take the values of 2 and 4, respectively. In the following we shall describe the step-by-step execution of the GCD workflow by referring to its high-level view given in Figure 26.

The workflow executes first Begin(Process) (that outputs two green tokens) followed by Begin(Flow) (that outputs four green tokens) and by Receive (that outputs one green token). As both numbers are strictly positive, Receive sends a blue token to Begin(While) and another blue (skipping) token to Throw. Because the suppressJoinFailure (set for the entire process only) has a yes value, skipping the Throw does not raise a joinFailure, but forwards the green token to End(Flow). The execution continues with Begin(While) and then with Begin(Scope) (as a *!=* b) that forwards a green token to Begin(Switch) and another to the Begin(FaultHandler) of the scope. The first Throw in the *switch* is skipped as $a<b$, yet the second one (of the *otherwise* branch) is executed, and a *dec_b* fault is raised. As a result, only a red token is sent further to the Begin(FaultHandler) of the scope. The first Assign is skipped (as *fault="dec_b"*), while the second Assign decreases the value of b by a. The green token will reach next End(FaultHandler) and then End(Scope) that forwards the green token to End(While) (as the fault was processed). Because $a=b=2$, End(While) sends a green token to End(Flow) and a blue token to Begin(Sequence), which enables the *sequence*. The execution of the Assign inside the Sequence leads to copying the value of a into c and to replying with the latter to the client. Finally, End(Sequence) outputs a green token that enables End(Flow), which has now gathered all its input (green) tokens. End(Flow) forwards a green token to End(Process) that sends the green token to the output condition, marking in this way the end of the workflow.

5 Concluding Remarks

In this paper, we have outlined the specification of a BPEL2YAWL[16] translator of BPEL processes into YAWL workflows. As we already anticipated in the Introduction, the main strengths of BPEL2YAWL are that (1) it provides an automated pattern-based compositional translation of BPEL processes into YAWL workflows, (2) it copes with all types of BPEL activities (including *flow*s with synchronisation links, and *scope*s), and (3) it handles the exceptional behaviour – events, faults and (explicit) compensation. Furthermore, (4) it can be straightforwardly plugged into our Web service discovery [3], aggregation [3, 4, 6], and adaptation [5, 7] methodologies, while (5) the patterns defined by the BPEL2YAWL translator provide the basis for the definition of an inverse YAWL2BPEL translator, which becomes straightforward. Last but not least, (6) BPEL2YAWL provides a lightweight semantics of BPEL processes, and (7) it sets the basis for the formal analysis of BPEL processes.

[16] A short description of the first version of the translator was given in [8].

Most of the approaches that translate BPEL processes into other languages or formalisms focus on the verification of properties of business processes. Fisteus et al. [1] describe VERBUS, a FSM-based framework for the formal verification of BPEL processes, but they do not treat synchronisation links, complex fault handling, and event and compensation handling. Koshkina and van Breugel [10] introduce the BPE-calculus in order to formalise the control-flow of BPEL and build upon it a tool for the analysis of business processes. Still, they do not tackle fault and compensation handling. Hinz et al. [9] give a PN semantics to BPEL processes by defining a pattern for each BPEL activity. However, they abstract from data and leave out transition guards. Consequently, control-flow decisions based on the evaluation of data are replaced by non-deterministic choices. Our approach does not suffer from this limitation as both BPEL and YAWL use XMLSchema and XPath for data manipulation, and hence the data translation between the two is straightforward. Ouyang et al. [12] formalise BPEL in terms of PNs with the purpose of analysing its control-flow. Although they handle both synchronisation links and exceptional behaviour, their approach is focused on the analysis of business processes, and it cannot be directly exploited to compose business processes.

Our main concern here was the translation of BPEL processes into YAWL workflows with the purpose of contributing to our long-term goal of aggregating and adapting heterogeneous Web services [7, 3, 4, 5, 6].

However, it is worth noting that the translation of BPEL processes into YAWL workflows also gives the possibility of formally analysing the business processes. YAWL is built on top of Petri nets, and it has a well-defined formal semantics based on transition systems, hence tools such as [19] and [20] can be employed to formally analyse YAWL workflows. Furthermore, in [3] we show how reachability graphs and modified reachability trees can be employed to check formal properties of YAWL workflows such as, lock-freedom, liveness, and so on.

A Java prototype of the BPEL2YAWL translator described in this paper has been implemented[17]. This first version of the translator can be successfully used to translate (simple) BPEL processes into YAWL workflows, which can be loaded and executed into the YAWL engine[18]. In short, the prototype consists of two Java packages – `BPELDoc` and `YAWLDoc`, for managing BPEL and YAWL documents, respectively. `BPELDoc` employs a data structure that models the hierarchy of a BPEL document. For example, the `BPELDoc` class has a `BPELProcess` object, which in turn has an `Activity` object, and optional `FaultHandler`, `EventHandler`, and `CompensationHandler` objects. Dually, `YAWLDoc` uses a data structure that models the nesting of a YAWL document[19]. For example, the `YAWLDoc` class refers to a `Decomposition` object, which can have multiple

[17] The source code of the prototype can be downloaded from: `http://www.di.unipi.it/~popescu/BPEL2YAWL.zip`. Moreover, a detailed discussion of the prototype is given in [11].

[18] `http://ga2377.campus.tue.nl:8080/worklist/`

[19] By "YAWL document" we refer to a YAWL XML file, and not to its binary representation generated with the YAWL editor.

`Task` and `Condition` objects. The control-flow is maintained by the `Tasks`, which store their input and output connections into `Mapping` objects. At runtime, `BPELDoc` first parses the input BPEL document into a `BPELDoc` object. Then, the `BPELDoc` object creates a `YAWLDoc` object, in which it suitably stores the transformation of the `BPELProcess` by recursive translations, starting with the `Activity` of the `BPELProcess` (as described in Section 3). Finally, `YAWLDoc` saves the obtained workflow as a YAWL document. Note that `YAWLDoc` also gives the possibility to load YAWL documents, so that one may use it to test the syntactic correctness of the translated workflow. Currently, the main limitation of our implementation of the BPEL2YAWL translator is that it does not cope with complex data structures and assignments such as mapping expressions to variables.

Our next step will be the further development of our Java prototype implementation of the BPEL2YAWL translator and its integration with the Java implementation of the core aggregation [3, 4, 6] and adaptation [5, 7] mechanisms, in order to yield a single tool supporting the disciplined, semi-automated aggregation and adaptation of BPEL services. Two further lines for future work are the development of the inverse YAWL2BPEL translator, as well as investigating the possibility of translating other types of Web service descriptions (e.g., OWL-S) into YAWL.

References

[1] J. Arias-Fisteus, L. S. Fernández, and C. D. Kloos. Formal Verification of BPEL4WS Business Collaborations. In K. Bauknecht, M. Bichler, and B. Pröll, editors, *EC-Web*, volume 3182 of *LNCS*, pages 76–85. Springer, 2004.

[2] BPEL4WS Coalition. Business Process Execution Language for Web Services (BPEL4WS) Version 1.1. (`ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`).

[3] A. Brogi and R. Popescu. Contract-based Service Aggregation. Technical Report TR-06-12, University of Pisa, April 2006. (`http://compass2.di.unipi.it/TR/Files/TR-06-12.pdf.gz`).

[4] A. Brogi and R. Popescu. Design and Implementation of Sator: a Web Service Aggregator. Technical Report, University of Pisa, 2006. (`http://www.di.unipi.it/~popescu/Sator.pdf`).

[5] A. Brogi and R. Popescu. Service Adaptation through Trace Inspection. In S. Gagnon, H. Ludwig, M. Pistore, and W. Sadiq, editors, *Proceedings of SOBPI'05*, pages 44–58, 2005. (`http://elab.njit.edu/sobpi/sobpi05-proceedings.pdf`).

[6] A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC'05*, volume 3826 of *LNCS*, pages 214–227. Springer, 2005.

[7] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In B. Benatallah, F. Casati, and P. Traverso, editors, *In Proceeding of the 4th International Conference on Service Oriented Computing (ICSOC 06), Chicago, USA, 4-7 December 2006*, LNCS, 2006. To appear.

[8] A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. In M. Bravetti, M. Nunez, and G. Zavattaro, editors, *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'06)*, volume 4184 of *LNCS*, pages 107–122. Springer, 2006.

[9] S. Hinz, K. Schmidt, and C. Stahl. Transforming BPEL to Petri Nets. In W. van der Aalst, B. Benatallah, F. Casati, and F. Curbera, editors, *Proceedings of the Third International Conference on Business Process Management (BPM 2005)*, volume 3649 of *LNCS*, pages 220–235, Nancy, France, Sept. 2005. Springer-Verlag.

[10] M. Koshkina and F. van Breugel. Verification of business processes for Web services. Technical Report CS-2003-11, York University, October 2003. (http://www.cs.yorku.ca/techreports/2003/CS-2003-11.ps).

[11] A. Lorenzani. Automated translation of web service descriptions, June 2006. CS Department, University of Pisa, Italy. (In Italian).

[12] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and A. H. ter Hofstede. Formal Semantics and Analysis of Control Flow in WS-BPEL. Technical Report 2174, Queensland University of Technology, February 2006. Available from: http://eprints.qut.edu.au/archive/00002174/01/BPM-05-15.pdf.

[13] OWL-S Coalition. OWL-S: Semantic Markup for Web Services Version 1.1. (`http://www.daml.org/services/owl-s/1.1/overview/`).

[14] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communication of the ACM*, 46(10):24–28, 2003.

[15] Satish Thatte. Web Services for Business Process Design, 2001. (`http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`).

[16] W. M. P. van der Aalst, L. Aldred, M. Dumas, and T. A. H. M. Hofstede. Design and Implementation of the YAWL System. In A. Persson and J. Stirna, editors, *In Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE'04), Riga, Latvia*, volume 3084 of *LNCS*, pages 142–159, 2004.

[17] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, 2005.

[18] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[19] E. Verbeek. WofYAWL Version 0.3. Technical report available online at `http://home.tm.tue.nl/hverbeek/wofyawl03.pdf`.

[20] E. Verbeek and W. M. P. van der Aalst. Woflan 2.0: A petri-net-based workflow diagnosis tool. In Nielsen, M. and Simpson, D., editors, *Lecture Notes in Computer Science: 21st International Conference on Application and Theory of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000*, volume 1825, pages 475–484. Springer-Verlag, 2000.

[21] W3C. Web Service Choreography Interface (WSCI) 1.0. World Wide Web Consortium (2002), http://www.w3.org/TR/wsci.

[22] W3C. Web Services Architecture, 2004. (`http://www.w3.org/TR/ws-arch/`).

[23] WSCDL Coalition. Web Services Choreography Description Language Version 1.0. `http://www.w3.org/TR/ws-cdl-10/`.

[24] WSDL Coalition. Web Service Description Language (WSDL) version 1.1. (`http://www.w3.org/TR/wsdl`).

[25] WSFL Coalition. Web Services Flow Language (WSFL) Version 1.0, 2001. (`http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`).