From BPEL Processes to YAWL Workflows*

Antonio Brogi and Razvan Popescu

Computer Science Department, University of Pisa, Italy

Abstract. BPEL is currently the most widespread language for composing Web services, but it lacks formal semantics. YAWL is a workflow language with a well defined formal semantics that implements the most common workflow patterns. In this paper we provide a methodology for translating BPEL processes into YAWL workflows, thus paving the way for the formal analysis, aggregation and adaptation of BPEL processes. The approach we propose 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

The service-oriented computing paradigm [9] uses services as building blocks for developing future heterogeneous, distributed applications. Two main reasons for composing (Web) services are the need for rapid application development, and the need to answer complex queries that cannot be satisfied by one service alone.

WSDL [12] is the current standard for describing Web service interfaces, yet it provides only a syntactic description of the supported operations. This severely affects the process of (semi-)automated service composition as composed services may lock during their interaction.

BPEL [2] has emerged as a language for expressing Web service compositions. A BPEL process provides the behaviour of a Web service in terms of coordinating one or more WSDL services. A downside of BPEL is that clients of the business process are in charge of manually selecting the services to be composed, and of building the composite service. Furthermore, BPEL lacks a formal semantics and hence it does not provide suitable means for the analysis of service compositions.

YAWL [10] 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 integrations. As it implements the most common workflow patterns, YAWL can be used as a *lingua franca* for expressing the behaviour of Web services (described using BPEL or OWL-S [8], for example). Despite its graphical nature, YAWL has a well defined formal semantics. It is a state-based language and the semantics of a workflow specification is defined as a transition system. Furthermore, being based on Petri nets, it provides a firm basis for the formal analysis of real-world services.

Our long-term goal is to provide a methodology for the (semi-)automated aggregation and adaptation of Web services into new heterogeneous applications.

* This work has been partially supported by F.I.R.B. project TOCAI.IT.

M. Bravetti, M. Nuñes, and G. Zavattaro (Eds.): WS-FM 2006, LNCS 4184, pp. 107–122, 2006.

To cope with the previous issues we argue for the use of *service contracts* [4] consisting of (a) a (WSDL) signature, (b) an (OWL) ontological description, and (c) a (YAWL) behaviour (or protocol). The signature and the ontological information serve for enhancing the service discovery process and for overcoming signature mismatches. The protocol information can be employed for generating the behaviour of the aggregated service and for verifying properties of the aggregate (such as lock freedom), as well as for coping with behavioural mismatches.

In [4] we described a core aggregation process for composing YAWL services. The core aggregation process inputs a set of service contracts to be aggregated and it outputs the contract of the aggregated service. The control-flow of the aggregate is built, on the one hand, from the initial control-flow of the participant services, and on the other hand, from data-flow dependencies obtained by semantically matching service parameters. This paper complements [4] by devising a methodology for translating BPEL processes into YAWL workflows. As a result, BPEL services can be translated into YAWL workflows, then aggregated, and finally deployed as a new BPEL service. It is worth stressing the importance of the last two features. As we will see, handling synchronisation links, scope activities, events, faults, and compensations, sensibly complicates the translation. Probably because of their complexity, these mechanisms have not usually been considered by the formalisations of BPEL that have been proposed so far (e.g., [6,1]). On the other hand, since these features are indeed exploited in real BPEL descriptions, and do contribute to the expressiveness of "real" BPEL, we argue that they cannot be ignored.

The translation approach we describe here defines a YAWL pattern for each BPEL activity, as well as for a whole BPEL process. The role of an activity pattern is twofold – to provide a unique representation of the activity, and to provide an execution context for it. Given a BPEL process, the approach automatically generates its YAWL translation by:

1. Instantiating the pattern of each activity defined in the BPEL process, and
2. Suitably connecting the obtained patterns into the final workflow.

The main features of the translation methodology can be summarised as follows:

- It is a pattern-based, compositional approach,
- It copes with all types of BPEL activities, and
- It handles events, faults and (explicit) compensation.

2 A Brief Introduction to BPEL and YAWL

The next two Subsections give a very high-level view of both languages. Some other details on the two languages will be discussed in the next Section, while describing the translation methodology. For a complete description of the two languages, please see [2] for BPEL, and [10] for YAWL.

2.1 BPEL: Business Process Execution Language

BPEL is a language for expressing the behaviour of a business process. It enables the specification of control and data logic around a set of Web service interactions. A BPEL process exposes a WSDL interface to its clients.

A BPEL process can be either abstract or executable. An abstract process hides implementation details (i.e., private information), while an executable process provides 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 of 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 and from the graph-based language WSFL. The former contributed with basic activities (e.g., for sending and receiving messages, for waiting for a period of time, and so on) as well as with structured ones (e.g., sequential or parallel execution of activities, activity scoping, and so on) for combining activities into complex ones. The latter brought the definition of links to synchronise activities executed in parallel. Other features of BPEL are instance management through correlation sets, event and fault handling, as well as compensation capabilities.

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 the process, *throw* for signalling faults, *terminate* for explicitly terminating the execution of the process, a dummy *empty* for doing a "no-op", *assign* for copying values between variables, and *compensate* for invoking a compensation handler.

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.

2.2 YAWL: Yet Another Workflow Language

YAWL 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 [11].

YAWL extends Petri Nets by introducing some workflow patterns (for multiple instances, complex synchronisations, and cancellation) that are not easy to express 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 the 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 managed through *predicates* in the form of logical expressions. When a task finishes its execution, it places tokens in its output places, depending on 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.

From a data-flow perspective, YAWL uses XMLSchema, XPath and XQuery for dealing with data. Variables are defined at both EWF-net and task levels, and bindings between them are realised through XQuery expressions.

3 From BPEL to YAWL

The objective of this paper is to present a methodology for translating BPEL processes into YAWL workflows. 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.

Subsections 3.1 and 3.2 introduce the *basic pattern template* and the *structured pattern template*, which are used to define the patterns of the basic and structured activities, respectively. Subsection 3.3 defines the *process pattern* 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 The Basic Pattern Template

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 only

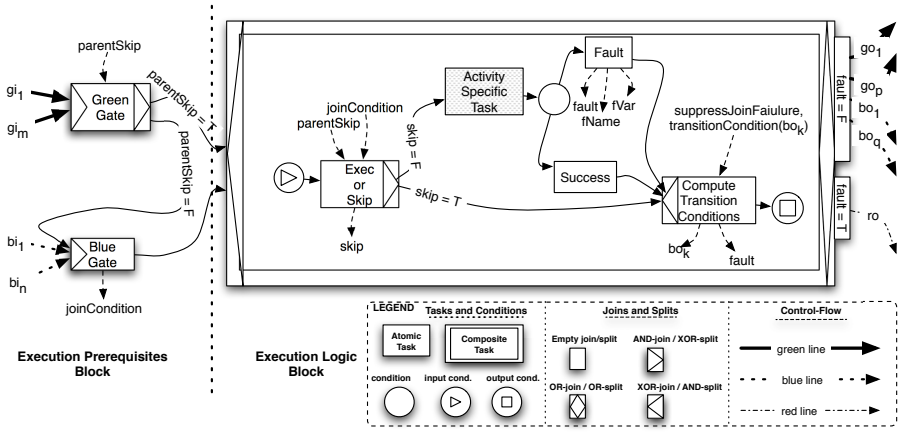

Fig. 1. The *basic pattern template*

when the first has finished its execution. Moreover, the *flow* construct allows for synchronisation links to be defined among activities. 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 other patterns. For example, the patterns translating child activities of a BPEL *sequence* have to be linked through green lines. On the other hand, we model the synchronisation links among BPEL activities using *blue line*s. 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. Finally, in order to cope with faults we use *red line*s. Patterns that treat errors have red inputs, while patterns that generate errors have red outputs. For example, the translation of the BPEL *throw* activity has to have a red line as output, while the translation of the BPEL *fault handler* should input a red line.

The *basic pattern template* is illustrated in Figure 1. 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 ouputs, 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[1] activity, whose value indicates whether the latter is 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. In this case, 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. Then, the *BlueGate* enables the *Execution Logic Block*.

The Execution Logic Block (ELB). The ELB has three possible behaviours: It can execute successfully, it can be silently skipped, or it can raise a fault. The third behaviour corresponds to a false *joinCondition* (see next) or to an erroneous execution of the activity.

The *ExecOrSkip* task of 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* evalutes to *false* then 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 *wait*.

The execution of the *ActivitySpecificTask* is simlutated through the deferred choice consisting of the *Fault* and *Success* tasks, together with their input place. The environment (viz., the "client" of the workflow) will determine whether *Fault* or *Success* will be 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

[1] 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.

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*, and so on). Otherwise, the *ActivitySpecificTask* is directly connected to the *ComputeTransitionConditions* task.

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. 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*.

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.

BPEL Basic Activities. Space limitations do not allow us to present the patterns of all the basic BPEL activities. We shall resume to presenting some general guidelines for customising and instantiating the basic pattern template.

In order to obtain the pattern of a basic activity, one has to (1) customise the *ActivitySpecificTask*, and (2) remove the deferred choice controlling the success of the activity if the activity cannot have an erroneous execution, as well as (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. Note that a pattern has at least one green input and one green output.

The *Invoke*, *Receive* and *Reply* patterns all have one green input. *Invoke* and *Reply* patterns have only one green output (for the pattern of the following activity, if any), while *Receive* can have at most two green outputs (the second to enable the pattern for event handling of the entire business process, if the *createInstance* attribute of the BPEL *receive* is set to *yes*).

The patterns *Throw*, *Wait*, *Terminate* and *Empty* have one green input and one green output, and they do not need the deferred choice block, as their execution cannot be erroneous. The *ActivitySpecificTask* will be hence directly connected to the *ComputeTransitionConditions* task. Note that a fault raised by a *Throw* pattern is not considered as an erroneous execution of the *throw* activity. Some other particularities are that the *ActivitySpecificTask* of the *Wait* pattern invokes the YAWL TimeService in order to delay the execution of the workflow, while the successful execution of a *Terminate* pattern leads to the cancellation

of all tokens inside the pattern translating the process activity. (Further details on the latter will be given later on when describing the *process* pattern.)

The *assign* and the *compensate* activities are treated as structured patterns, as we will see in the next Subsection.

3.2 The Structured Pattern Template

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 of a *PatternTemplate* 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 *PatternTemplate* corresponding to a child activity that can be executed first. On the other hand, *End* has to wait for the green tokens from all *PatternTemplate*s 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 fault 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 *End*. In this way we achieve the dead-path-elimination inside structured patterns.

Now, the patterns of all structured activities are obtained by adjusting the *Begin* and *End* patterns and by suitably interconnecting them with the *PatternTemplate*s. 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*.

BPEL Structured Activities. Space limitations do not allow us a detailed description of all the structured patterns. However, we shall try to describe the most relevant features of each pattern.

The *Sequence*, *Switch*, *Flow* and *Pick* patterns all share the same structure:

$$\begin{array}{rcccc} Sequence & \rightarrow & Begin(Sequence) & PatternTemplate^{+} & End(Sequence) \\ Switch & \rightarrow & Begin(Switch) & PatternTemplate^{+} & End(Switch) \\ Flow & \rightarrow & Begin(Flow) & PatternTemplate^{+} & End(Flow) \\ Pick & \rightarrow & Begin(Pick) & PatternTemplate^{+} & End(Pick) \end{array}$$

The *Sequence* pattern consists of a *Begin* and an *End* pattern, together with at least one *PatternTemplate*. *Begin(Sequence)* must enable the execution of only the first *PatternTemplate* in the sequence, each *PatternTemplate* enables the next one in the sequence, and *End[Sequence]* must wait for the last *PatternTemplate* to finish its execution.

The *Switch* pattern includes one *PatternTemplate* for each conditional branch, and each *PatternTemplate* must verify the guard condition of the corresponding branch. A *false* guard leads to skipping the corresponding branch and hence to dead-path-elimination inside the corresponding pattern. The *PatternTemplate*s are linked in the order in which the conditional branches occur in the *switch* activity. If no *otherwise* branch is defined, a default one with an *empty* activity guarded by an (always) *true* condition is considered.

A *flow* activity concurrently executes a bag of activities among which synchronisation links can be defined. *Begin(Flow)* has to enable the patterns of all its child activities, and hence it has one green output for each *PatternTemplate*. Dually, the execution of *End(Flow)* is delayed until all *PatternTemplate*s finish their execution.

A *pick* basically waits for a message or an alarm event to take place. Its pattern is slightly more complicated due to the fact that the first event that is triggered causes all other events to be cancelled. *Begin(Pick)* mainly differs from *Begin(Flow)* in that its *ActivitySpecificTask* is a composite task in charge of branch selection. Moreover, each *PatternTemplate* of the pick has a guard condition that checks whether its branch id matches the id of the branch selected in *Begin(Pick)*. Although only one branch will be actually executed, *Begin(Pick)* sends green tokens to all *PatternTemplate*s in order to perform the dead-path-elimination on the branches that were not selected. *End(Pick)*, similarly to *End(Flow)*, waits for the green tokens from all branch patterns.

The *While* pattern

$$While \ \rightarrow \ Begin(While) \ \ PatternTemplate \ \ End(While)$$

differs from the *Sequence* pattern as *Begin(While)* has two green input lines and a guard condition. A green input token comes either from the pattern structurally preceding the while, or from *End(While)* in order to loop. Dually, *End(While)* outputs a green token either for the pattern structurally following the while, or for *Begin(While)*. The guard condition is checked again by *End(While)* in order to avoid skipping the whole while in case of a *false* guard at the end of a cycle.

Although *assign* is a basic activity, it is translated with a structured pattern since it can contain several *copy* tags, each of which requiring a data exchange which may lead to a fault being raised. The *Assign* pattern:

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

has the same structure of the *Sequence* pattern, but it includes *Copy* patterns rather than arbitrary *PatternTemplates*. A *Copy* pattern is obtained from the *basic pattern template* by replacing the *ActivitySpecificTask* with a task named *Copy*, which inputs the "source" variable and which outputs the "target" variable. In this way, the assignment is achieved through the data mappings of the

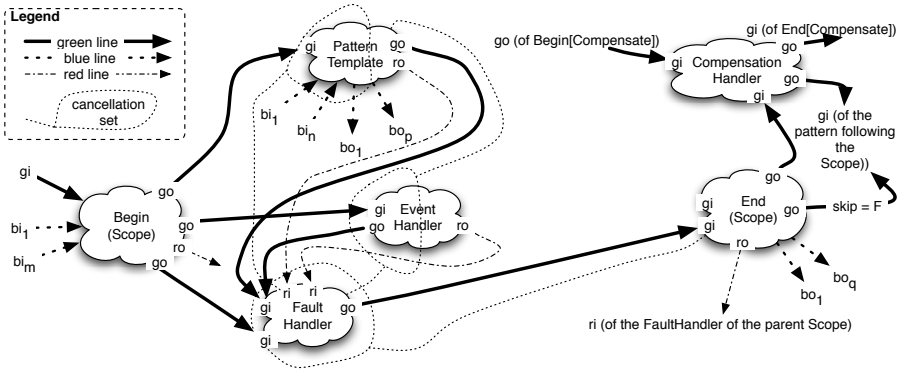

Fig. 2. High level view of the *Scope* pattern

Copy patterns. Furthermore, the *Copy* pattern does not have blue inputs and outputs.

A BPEL *scope* provides a specific context for an activity. It contains a (possibly default) *fault handler*, a (possibly default) *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 fault name and an optional fault variable. A *catchAll* has an always true guard and no fault name or variable. The *compensation handler* provides a (compensating) activity that can be invoked either explicitly (through a *compensate*), or implicitly (in case of a fault). The *compensation handler* is activated only when the scope finished its execution successfully. (In this paper we deal with explicit compensation only, due to the troublesome default compensation mechanism, e.g., compensating a *scope* inside a *while*). Last but not least, an *event handler* defines message and alarm events that can be triggered repeatedly and concurrently during the lifetime of the scope. The *Scope* pattern has the structure:

$$Scope \;\rightarrow\; Begin(Scope)\ PatternTemplate\ FaultHandler \\ [CompensationHandler]\ [EventHandler]\ End(Scope)$$

and the structural dependencies among the various patterns involved are illustrated in Figure 2. *Begin(Scope)* sends green tokens to the *PatternTemplate*, to the *EventHandler*, and to the *FaultHandler*. The *FaultHandler* will further receive either one green token from the *PatternTemplate* and one green token from the *EventHandler* (if any), or one red token from the *PatternTemplate* or from the *EventHandler*. In the former case, the entire *FaultHandler* will be skipped either because the *PatternTemplate* 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 *PatternTemplate*, or inside the *EventHandler*. In case that the fault cannot be processed, the *FaultHandler* will send a green token to *End(Scope)*, which has to send a red token further to the *FaultHandler* of the

parent scope pattern (if any). Note that only the *FaultHandler* will forward a (green) token to *End(Scope)*. *End(Scope)* is in charge of enabling the *CompensationHandler* if the *PatternTemplate* was successful. It is important to note that *End(Scope)* has to save a copy of all the scope variables as required by the *CompensationHandler*. If the scope is skipped, *End(Scope)* has to clear the green tokens received by the *FaultHandler* from the *PatternTemplate* and from the *EventHandler* as they are redundant. 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:

$$Begin(FaultHandler)\ \ PatternTemplate^*\ \ End(FaultHandler)$$

except that each *PatternTemplate* corresponds here to a *catch* activity and hence it has a guard condition checking the fault name and data. 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 *PatternTemplate* (or from inside the *EventHandler*) of its scope. 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 scope's *PatternTemplate* and *EventHandler* except *CompensationHandler* patterns corresponding to scopes nested in its scope. 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. In this way, the faults received by this default *FaultHandler* will be forwarded (through *EndScope*) to the *FaultHandler* of the parent scope (or of the entire process). In the pattern of the *EventHandler*:

$$Begin(EventHandler)\ \ PatternTemplate^+\ \ End(EventHandler)$$

the *PatternTemplate*s execute concurrently, and each one is placed in a loop with a guard that checks the end of the *PatternTemplate* pattern translating the activity inside the scope. Note that the scope's *PatternTemplate* is in charge of clearing all tokens of the *PatternTemplate*s that implement alarms upon its completion. Finally, the *CompensationHandler* pattern is:

$Begin(CompensationHandler)\ \ PatternTemplate^*\ \ End(CompensationHandler)$

If the scope completes successfully, the *Begin(CompensationHandler)* is activated and waits for a green token from a *Compensate* pattern. Upon completion, the *End(CompensationHandler)* returns the green token to the *Compensate*. 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)*.

Finally, the BPEL *compensate* is translated with the pattern:

$$Begin(Compensate)\ \ End(Compensate)$$

since *compensate* terminates only when the invoked *CompensationHandler* finishes its execution. Recall that we consider only simple explicit compensation, that is *compensate* activities specifying the name of the scope to be compensated,

without considering scopes 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. Then, it forwards it to the pattern structurally following the *compensate*. Further details on the *Scope* pattern will be commented in the Section dedicated to discussing an use case.

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*.

The *Process* pattern:

Begin(*Process*) *FaultHandler* [*EventHandler*] *PatternTemplate End*(*Process*)

resembles the *Scope* pattern, altohough there are several differences between the two. 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 *PatternTemplate*, the *FaultHandler*, as well as the *EventHandler* (if any). If the BPEL process does not define a *FaultHandler*, 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). Differently from the *Scope*, there are no green tokens being sent from the *PatternTemplate* and from the *EventHandler* to the *FaultHandler*. This is due to the fact that the *FaultHandler* cannot be skipped because neither the *Process* can be skipped nor the dead-path-elimination must be employed inside its *FaultHandler* pattern. The *PatternTemplate* and *EventHandler* forward each one green token to *End(Process)*.

The *EventHandler* is active for the entire process lifetime and the *PatternTemplate* of 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 *PatternTemplate* corresponding to the activity defined by the process. Hence, *End(Process)* is enabled if it receives either one green token from the process *PatternTemplate* and another from the *EventHandler* (if any), or one green token from a *Terminate*, or from the process *FaultHandler*.

The *compensation handler* can only be invoked by platform-specific means. Consequently, we do not consider a *compensation handler* for the entire busi-

ness process. Furthermore, the process *compensation handler* 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 *PatternTemplate* corresponding to the process activity. 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.

4 A Use Case

Consider a simple BPEL process that computes the greatest common divisor (GCD) of two numbers. Basically, the GCD is computed by repeatedly raising an exception if one of the two numbers is bigger than the other and by decreasing its value in the corresponding catch. Due to space limitations we present hereafter a simplification of the BPEL process. Figure 3 gives the high-level view of the YAWL workflow obtained from the GCD process. The interested reader is kindly asked to download the archive containing the full BPEL process as well as the YAWL workflow of the example from the following address: `http://www.di.unipi.it/~popescu/GCD_Example.zip`.

```
<process name="S" 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>
```

Consider an execution scenario in which the two input variables a and b take the values of 2 and 4, respectively. 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 $a \mathrel{!}= b$) that forwards a green token to *Begin(Switch)* and another to the *Begin(FaultHandler)* of the scope. The first

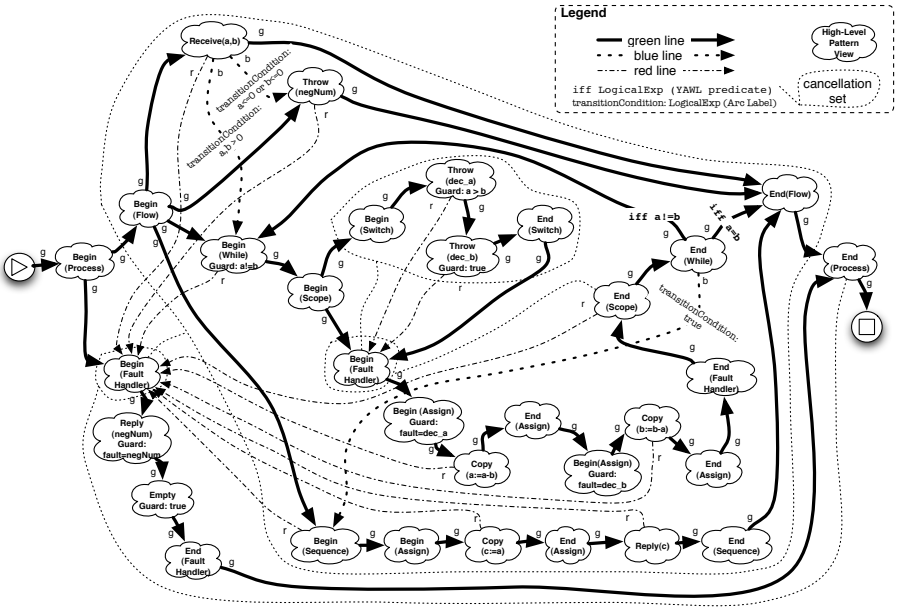

Fig. 3. YAWL workflow obtained from the GCD process

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 (that clears all tokens of the *Switch*). 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)*. 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 first clears all input tokens of the *Begin(FaultHandler)* of the process, and then it sends the green token to the output condition, marking in this way the end of the workflow.

5 Concluding Remarks

We have outlined a methodology for translating BPEL processes into YAWL workflows. Its main strengths are that (1) it defines YAWL patterns for all BPEL activities, (2) it provides a compositional approach to construct structured patterns from suitably interconnecting other patterns, and (3) it handles events, faults and (explicit) compensation.

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, as well as event and compensation handling. Koshkina and van Breugel [6] introduce the BPE-calculus in order to formalise the control-flow of BPEL and build upon it a tool for the analysis (e.g., deadlock freedom) of business processes. Still, they do not tackle fault and compensation handling. Hinz et al. [5] 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. [7] formalise BPEL in terms of PNs with the purpose of analysing its control-flow. Although they handle both synchronisation links and exceptional behaviour, their focus is on the analysis, and not on the composition of business processes.

We believe that the translation described in this paper constitutes an important brick for the development of formal analysis and transformations of BPEL processes. It also directly contributes to our long-term goal of aggregating and adapting heterogeneous Web services [4,3]. In this perspective, our next step will be the integration of our Java prototype implementation of the BPEL2YAWL translator with the Java implementation of the core aggregation mechanism of [4], in order to yield a single tool supporting the disciplined, semi-automated aggregation of BPEL services. A further line for future work is the development of other translators to convert 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. 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`).
4. 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.
5. 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.
6. 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`).

7. 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.
8. OWL-S Coalition. OWL-S: Semantic Markup for Web Services Version 1.1. (http://www.daml.org/services/owl-s/1.1/overview/).
9. M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communication of the ACM*, 46(10):24–28, 2003.
10. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, 2005.
11. 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.
12. WSDL Coalition. Web Service Description Language (WSDL) version 1.1. (http://www.w3.org/TR/wsdl).