# REFlex: An Efficient Web Service Orchestrator for Declarative Business Processes

Natália Cabral Silva, Renata Medeiros de Carvalho,
César Augusto Lins Oliveira, and Ricardo Massa Ferreira Lima

Center for Informatics (CIn),
Federal University of Pernambuco,
Recife – PE, Brazil
{ncs,rwm,calo,rmfl}@cin.ufpe.br

**Abstract.** Declarative business process modeling is a flexible approach to business process management in which participants can decide the order in which activities are performed. Business rules are employed to determine restrictions and obligations that must be satisfied during execution time. In this way, complex control-flows are simplified and participants have more flexibility to handle unpredicted situations. Current implementations of declarative business process engines focus only on manual activities. Automatic communication with external applications to exchange data and reuse functionality is barely supported. Such automation opportunities could be better exploited by a declarative engine that integrates with existing SOA technologies. In this paper, we introduce an engine that fills this gap. REFlex is an efficient, data-aware declarative web services orchestrator. It enables participants to call external web services to perform automated tasks. Different from related work, the REFlex algorithm does not depend on the generation of all reachable states, which makes it well suited to model large and complex business processes. Moreover, REFlex is capable of modeling data-dependent business rules, which provides unprecedent context awareness and modeling power to the declarative paradigm.

**Keywords:** declarative business process, business process flexibility, business rules, web services orchestrator, context awareness.

## 1 Introduction

Processes change very often in some business areas. Customer demands are volatile and business partners change frequently as new opportunities arise. To remain competitive, organizations need to be prepared to confront unforeseen situations. This requires flexible processes that can be adapted to cope with changes in the environment [10]. Such flexibility requirements motivated the development of the *declarative* business process paradigm. The declarative approach differs from most traditional, imperative modeling approaches because they model *what* must be done but do not prescribe *how* [13] to do it. In this way,

S. Basu et al. (Eds.): ICSOC 2013, LNCS 8274, pp. 222–236, 2013.

declarative business processes allow the participants to decide which activities are more appropriate for each particular situation.

Service Oriented Architecture (SOA) is a well-established software architecture for the design of enterprise applications. In SOA, a number of business activities may be performed by mature services provided by partners and third-party enterprises. These services deliver functionalities that can be shared and reused across the enterprise. Web service orchestration is an essential feature of most business process enactment frameworks. They enable the construction of automated workflows that explore the benefits provided by SOA. However, current technologies for declarative business process enactment (Declare and DCR Graphs) are focused in the modeling of manual activities. They lack adequate support for web service integration. This impairs their use in more complex applications in which both manual and automated activities are necessary.

The purpose of this paper is to introduce an approach for integrating the declarative paradigm with SOA. Our tool, called **REFlex** (*Rule Engine for Flexible Processes*), is a declarative process management system that provides a rule engine for declarative processes and a language for service orchestration [4][14].

REFlex is based on an efficient algorithm that does not rely on state-space generation [4]. Moreover, it offers support for data manipulation and data-dependent rules, which improves its context awareness and increases the range of modeling capabilities.

This paper is organized as follows. An overview of background and related work is presented in Sections 2 and 3, respectively. REFlex rule engine is described in Section 4. Next, Section 5 describes the orchestration mechanism and architecture. To demonstrate the use of the proposed approach, a case study is presented in Section 6. Finally, Section 7 discusses conclusions and future work.

## 2 Declarative Processes

When the company tasks are less repetitive and predictable, workflows are not able to properly represent the possible flows of work [10]. They often are either too simple, thus unable to handle the variety of situations that occur; or they are too complex, trying to model every imagined possible situation but being hard to maintain. In both cases, they may cause several problems to the company. To tackle these limitations, flexible processes surged as a shift paradigm from traditional workflow approaches [15].

Declarative business processes define the process behavior by *business rules* described in a declarative language. Traditional workflows take an "inside-to-outside" strategy, where all the executions alternatives must be explicitly represented in the process model. On the other hand, a declarative process takes an "outsite-to-inside" strategy, where the execution options are guided by constraints [13]. Adding new constraints reduces the number of execution options.

In this constraint-based approach, a process model is composed of two elements: activities and constraints. An activity is an action that updates the enterprise status and is executed by a resource. A constraint is a business rule

that must be respected during the entire process execution. Thereby, the permission to execute activities is controlled by business rules. Each activity has its execution enabled only if and when the business rules allow it.

## 3 Related Work

In this section we discuss some important works in the areas of declarative processes and web service composition.

### 3.1 Declarative Processes

*Declare* is a rule engine system proposed by Pesic et al. [12] for modeling and executing declarative processes through an extensible graphical language called **ConDec** [13]. This language offers a set of graphical representations to describe control-flow rules that constrain the execution of process activities. Declare uses Linear Temporal Logic (LTL) as its formalism for the internal representation of business processes. Process enactment requires the construction of a Büchi automaton that contains all possible states of the process. This strategy leads to the well known problem of *state space explosion*, which limits the size and complexity of the business processes that can be executed using Declare.

Hildebrandt and Mukkamala [8] propose a graph-based model called *DCR Graphs* to specify business process rules. At runtime, DCR Graphs control the dynamic evolution of the process' state. Since the states of the process are updated dynamically, this approach does not require a prior generation of all possible states.

In a previous work, we proposed an approach for declarative processes that is based on event-driven programming [11]. This approach aims to minimize the gap between the business rules and their implementation by systematically moving from business rules described in natural language towards a concrete implementation of a business process. We use complex event processing (CEP) to implement such a process. CEP is more expressive than the above mentioned languages. It can describe both control-flow and data dependencies. However, this approach is not based on an underlying formal model that guarantees the correctness of the resulting process models.

Except from our former approach using CEP, none of the aforementioned declarative approaches are data-aware, i.e., the activities and rules do not have the concept of context data. Furthermore, none of these works are integrated with SOA concepts, i.e., the activities are not executed by web service invocation.

Works on process mining have already considered data-aware rules in declarative processes. Through the SCIFF Checker [1], a set of execution traces can be classified as compliant or not compliant with a group of business rules. However, SCIFF's algorithm is only useful for verification purposes and is not applicable for business process enactment.

### 3.2 Web Service Composition

None of the current tools that support the execution of declarative processes [8][12] employ Service-Oriented Computing (SOC) concepts. In all these systems, the activities are manual. The user only informs when the activity starts, concludes, or is canceled. There is no built-in support for activity automation.

The *Business Process Execution Language* (BPEL) [2], is the *de facto* standard for web services orchestration. However, it is static and not easy to adapt [16]. In this regard, a number of works propose flexible alternatives to BPEL, to allow for the construction of more flexible and adaptable business processes [9][5][7][17].

*VxBPEL* [9] is an extension to the standard BPEL language that provides VariationPoint, which is a container of possible BPEL codes available for selection at runtime.

AO4BPEL [5] is another BPEL extension that improves the business process flexibility using aspect-oriented concepts. The BPEL structure is expanded to include aspects, which define fragments of business processes that can be inserted into one or more process models at runtime.

CEVICHE [7] is a tool that employs the AO4BPEL. CEVICHE's users do not activate the aspects. Instead, the aspects are activated when event patterns are recognized by a Complex Event Processing (CEP) engine. Thus, CEVICHE can automatically decide *when* and *how* to adapt the system by analyzing events with the CEP technology.

Xiao *et al.* [17] propose a constraint-based framework that employs process fragments. A process fragment is a portion of a process that can be reused across multiple processes. These fragments are selected and composed based on some business constraints and policies. The resulting process is a standard BPEL process, deployable on standard BPEL engines.

Another dynamic composition proposal is the SCENE service execution environment [6]. It allows the BPEL to be changed at runtime by choosing the correct service to be invoked based on business rules. These rules are used to realize the correct bindings between the BPEL engine and the services. For this purpose, there is a rule engine that makes the decisions about the services selection.

All the aforementioned works are extensions to BPEL aiming at making it more adaptive. However, none of them provide ways to execute declarative processes. Since declarative processes do not have any predefined structure, it is not possible to execute them using BPEL or its extensions. Therefore, integrating web services and the declarative approach requires the development of novel engine technology.

## 4 REFlex Rule Engine

In this section, we describe the rule engine used by REFlex to control the execution of declarative processes.

The declarative business process engines primary task is to guarantee that all process instances adhere to the business rules defined for that process. To accomplish this, the engine must prevent the user from executing activities that violate

the rules and must also oblige the execution of pending activities. Moreover, the engine should not let the user execute a sequence of activities that blocks the completion of the process. In other words, the engine must guarantee that the process never reaches a deadlock state, in which pending activities cannot be executed.

The REFlex rule engine is an efficient declarative process engine. It does not have the state explosion problem that is exihibited by DECLARE, since it does not require the previous generation of the complete set of reachable states [4]. The state of the process is updated dynamically. To avoid deadlocks, REFlex uses a *liveness-enforcement* algorithm that guarantee that a deadlock state is never reached.

REFlex models are *directed graphs* in which **nodes are activities** and **arcs define the relationship** between activities. Table 1 describes the rule types of REFlex, as well as their graphical representation (arcs) and semantics. An example of REFlex model can be seen in Section 6.

Observe that several ConDec rules can be implemented using REFlex rule types. The translation from ConDec to REFlex rules is shown in Table 2.

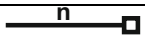
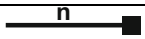
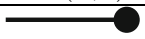
During runtime, activities change their state. An activity may be *enabled*, *disabled*, or *blocked*. An enabled activity may also be *obliged*. An *obliged* activity that has not be executed is called "pending activity". Activities that are *disabled* or *blocked* cannot be executed by the user. Moreover, the process can not be concluded if there are *pending activities* left.

When the user executes an enabled activity $X$, an *exec(X)* event is issued. This event causes an update in the state of the system according to the process rules (Table 1). Furthermore, certain rules are valid only at the first execution of an activity (see, for example, precedence). In such cases, the rule is *removed* from the process after its conditions have been fulfilled.

To guarantee deadlock freedom, i.e., that the process instance never reaches a deadlock state, REFlex inserts liveness-enforcing rules in the model. The objective of these rules is to disable execution paths that would certainly result in a deadlock in a future step of execution. As an example, consider a process with three activities (A, B, C) and two rules: *response*(A, B) and *not after*(C, B). Clearly, after the execution of $A$, the activity $C$ can not be executed until $B$ is executed. This is because if $C$ executes between $A$ and $B$ it would make activity $B$ both obliged and blocked, which configures a deadlock. A similar situation occurs if $A$ executes after $C$. Indeed, $A$ can *never* execute *after* $C$ in such a process.

The idea behind the liveness-enforcing algorithm is to avoid all situations that cause an activity to be simultaneously *blocked* and *obliged*. To accomplish that, we analyze the model statically and (transparently) insert new rules that are specific to control such situations. The model can execute only after applying the algorithm to analyze the model and inserting the rules to remove deadlock threats. The rules inserted by the algorithm do the following: 1) "propagate" blocking states (e.g., if $A$ obliges $B$ and $B$ obliges $C$, when $C$ is blocked, so are $A$ and $B$); 2) disable blocking when the activities that would be blocked are

**Table 1.** REFlex rule types

| **Existential Rules** | |
|---|---|
| *at least*(A, n) |   |
| **Behavior** | **Semantics** |
| $A$ is initialy obliged and remains obliged until it is executed $n$ times. | **when** *exec(A)* **do** if $(n > 0)$ then $n = n - 1$ |
| *at most*(A, n) |   |
| **Behavior** | **Semantics** |
| After $n$ executions of $A$, it is blocked. | **when** *exec(A)* **do** if $(n > 0)$ then $n = n - 1$; **when** $n = 0$ **do** block A |
| **Relational Rules** | |
| *response*(A, B) |  |
| **Behavior** | **Semantics** |
| After the execution of $A$, $B$ is obliged. | **when** *exec(A)* **do** oblige B |
| *responded existence*(A, B) |  |
| **Behavior** | **Semantics** |
| The first execution of $A$ obliges $B$. If $B$ is executed before $A$, remove the rule. | **when** *exec(A)* **do remove** *resp.existence*(A,B), oblige B; **when** *exec(B)* **do remove** *resp.existence*(A,B) |
| *precedence*(A, B) |  |
| **Behavior** | **Semantics** |
| While $A$ is not executed, $B$ is disabled. | **when** *exec(A)* **do remove** *precedence*(A,B) |
| *not after*(A, B) |  |
| **Behavior** | **Semantics** |
| After the execution of $A$, $B$ is blocked. | **when** *exec(A)* **do** block B |
| **Default Rules (valid for all activities in all processes)** | |
| *disable*(A) | |
| **Behavior** | **Semantics** |
| If exists $X$ such that *precedence*(X, A), $A$ is disabled. | **if exists** *precedence(X, A)* **then** disable A |
| *waive*(A) | |
| **Behavior** | **Semantics** |
| After the execution of $A$, if there is an *at least*(A, n), $n > 0$, then $A$ is obliged. If not, $A$ is not obliged. | **when** *exec(A)* **do if not exists** *at least(A, n)*, $n > 0$ **then** $A$ is not obliged **else** $A$ is obliged |

already obliged (e.g., in the previous example, if $D$ blocks $C$, when $A$, $B$, or $C$ are obliged, $D$ is disabled until the obligations are waived); 3) oblige precedences (e.g., if there is a *precedence* (A, B), obliging $B$ also obliges $A$); and 4) blocking precedences (e.g., if there is a *precedence* (A, B), blocking $A$ also blocks $B$). A proof for the *liveness* of REFlex models is described by Carvalho et al. [3].

**Table 2.** ConDec templates X REFlex rule types

| Condec Rules | Description | Reflex Rules |
|---|---|---|
| **Existential Rules** | | |
| *init*(A) | All activities but $A$ are disabled until the execution of $A$. | For all activities ($a_i$) except $A$, *precedence*(A, $a_i$) |
| *existence*(A, n) | $A$ is obliged until it is executed $n$ times. | *at least*(A, n) |
| *absence*(A, n) | After $n-1$ executions of $A$, it can not be executed anymore. | *at most*(A, n-1) |
| *exactly*(A, n) | $A$ must be executed exactly $n$ times in a process instance. | *at least*(A, n) and *at most*(A, n) |
| **Relational Rules** | | |
| *response*(A, B) | After the execution of $A$, $B$ is obliged. | *response*(A, B) |
| *precedence*(A, B) | While $A$ is not executed, $B$ is disabled. | *precedence*(A, B) |
| *succession*(A, B) | After the execution of $A$, $B$ is obliged, but it is disabled while $A$ is not executed. | *precedence*(A, B) and *response*(A, B) |
| *coexistence*(A, B) | $A$ and $B$ are either both executed or not executed at all. | *responded existence*(A, B) and *responded existence*(B, A) |
| *responded existence*(A, B) | The first execution of $A$ obliges $B$. | *responded existence*(A, B) |
| **Negation Rules** | | |
| *not response*(A, B) | After the execution of $A$, $B$ can not be executed. | *not after*(A, B) |
| *not coexistence*(A, B) | $A$ and $B$ can not be both executed in the same process instance. | *not after*(A, B) and *not after*(B, A) |

### 4.1 Data-Aware Extension

The semantics of REFlex is extended to support data-dependent rules. This kind of rule is applied only if certain conditions hold. Such data-aware extensions provide unprecedented expressive power to declarative business processes. Few engines today are able to model this type of constraints [11]. Yet, data-dependent rules are ubiquous. It is difficult to model large, realistic declarative processes without data-dependencies.

Data-dependent rules are constructed following the pattern:

**IF** predicate **THEN** *rule (...)*

Graphically, a data-dependent rule is represented by an arc that has an inscription attached. The inscription corresponds to the predicate that is the condition for the rule.

The semantics is the following. If the rule's predicate is true, the rule is part of the process. We say that the rule is *active* in the process instance. If the predicate is false, the engine ignores the rule. We say that it is *inactive*.

Some example of data-dependent rules are:

1. *A reimbursement for expenditure can not be sent after the grant is cancelled, unless the expenditure is prior to the cancellation date*

   **IF** date of expenditure > date of cancellation
   **THEN** *not after* ("cancel grant", "send reimbursement")

2. *If a rented car is returned after the expected return date, a charge must be issued*

   **IF** return date > expected return date
   **THEN** *response* ("return car", "issue charge")

Liveness-enforcement is a challenge for data-dependent rules. The reason is that it is not always possible to foresee which rules will be activated at runtime. For example, let us assume we have a data-dependent *response (A, B)* and a regular *not after (C, B)*. In the moment that $A$ is obliged, if the *response* rule is active, the engine disables $C$ until $A$ and $B$ execute. However, if the *response* rule is not active at this moment, it is ignored. So the engine lets the user execute $C$, which blocks $B$. Suppose that, after the execution of $C$, the conditions for activating the *response* are met. Now we have a situation that leads to a deadlock, once the execution of $A$ will oblige activity $B$, which is currently blocked.

The problem just described can be solved if we restrict the action of activity $C$ over the variables of the process. Once $C$ blocks an activity, we can not allow that the execution of $C$ itself creates conditions to oblige that activity.

This solution can be generalized as follows. First, we identify all activities that may be obliged in the next or in a future execution step (Def. 1).

**Definition 1 (Possible obligation).** *We say that an activity A possibly obliges B in a process instance if:*

- *the process contains the rule* response*(A, B) and the conditions for its activation are* satisfied*; or*
- *the process contains the rule* responded existence*(A, B) and the conditions for its activation are* satisfied*; or*
- *there is an activity X that* possibly obliges *B and A* possibly obliges *X*.

Next, we restrict the variables that can be affected by certain activities of the process, according to Def. 2.

**Definition 2 (Data-Restricted Activities).** *An activity A is not allowed to change the value of a process variable x if, for any activity B:*

1. *the process contains a rule* not after *(A, B) and*
   (a) *there is one or more conditional rules in the process that depend on x;*
   (b) *the rules that are conditioned to x affect whether B is* possibly obliged *or not in a process instance.*
2. *or the process contains a rule* at most*(A, 1) and*
   (a) *the rules that are conditioned to x affect whether A is* possibly obliged *or not in a process instance.*

The modeler is responsible for assuring the process's conformance to the data restrictions in Def. 2. With such an approach, we guarantee that, if an activity $A$ blocks another activity $B$ in the process, the latter will never be obliged in the future. If $B$ is obliged before $A$, the liveness-enforcing rules already inserted into the model will prevent $B$ from executing while $A$ is obliged. The liveness-enforcing rules will also guarantee that no activity will ever oblige $B$ after it has been blocked.

## 5 REFlex Orchestrator

Amongst the engines for declarative processes, only REFlex has the ability to execute external web services. Such feature allows for the modeling of semi-automated declarative business processes. This section describes how REFlex's orchestration mechanism works.

In most business processes, there are several opportunities for the automation of tasks through the use of web services. For example, we may want to query a database for product or customer information, to schedule an appointment into an on-line calendar, or perhaps to register an authorization for a new employee. We may want to be able to request an operation from a web service, send our process' data to it, and use the data from its response in other activities of the process.

To implement this feature, REFlex allows the user to set up variables and service bindings. *Variables* may be global (accessible in the entire process) or local (accessible in the scope of a single activity). The user can define any number of variables for a process. Currently, the data types supported by REFlex are *int, float, double, String, boolean* and *list* (an array of elements of any of the primitive types).

*Service bindings*, in turn, enable the linkage between an activity in the process and an external web service. A service binding describes which web service is linked to the activity, the location of its WSDL interface, and the binding, port type, and operation that should be called when the user wants to execute the activity.

REFlex uses the WSDL interface of the web service to automatically construct and interpret SOAP messages that are sent/received to/from the web service. The variables of the process are filled in the SOAP message body according

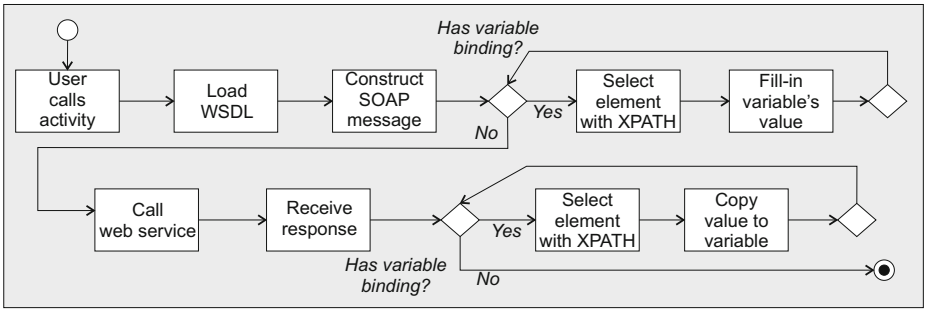



**Fig. 1.** REFlex orchestration process

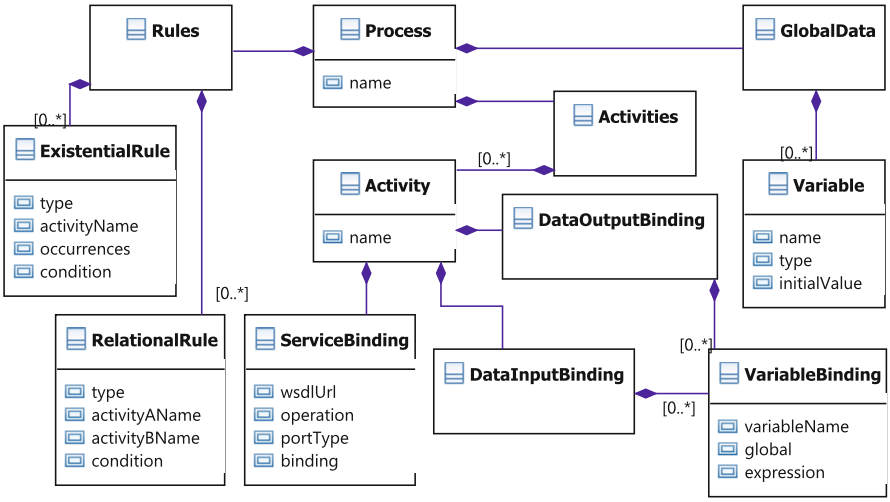



**Fig. 2.** Elements of a Process Definition XML

to the activity's *data input binding* and *data output binding*. These elements define *variable bindings*, which map a variable into an element inside the SOAP body using XPATH expressions. When calling the web service, the values of the process' variables are copied to the SOAP message. Once the response from the web service is received, its contents are copied into process' variables, according to the variable bindings.

The process just described is illustrated in Fig. 1.

The process definition is described in an XML file, which contains all information necessary to execute the process and to communicate with web services. A process definition includes the elements presented in the diagram shown in Fig. 2.

There are four major components in REFlex architecture [14]: REFlex is composed of four main components: the *Engine*, which interprets the rules and

updates the states of the activities; the *Data Manager*, which stores data variables and controls their access and updates; the *Service Manager*, which interprets WSDL descriptions, creates/interprets SOAP messages and invokes web services when demanded; and the *Process Instance Manager*, which manages the interaction of these components and communicates with the user. The Process Instance Manager interprets process models described in the XML language, initialize process instances, and interpret user inputs.

## 6 Case Study

This section presents a case study that demonstrates the use of REFlex. A business process of a travel agency is modeled. Common tasks performed by this travel agency are flight and/or hotel booking. Moreover, currency conversion is often needed for international trips. These activities compose the *Travel Arrangements* process.

The agency's information system offers three web services whose operations are useful for the Travel Arrangements process. Table 3 details the three services, their operations, and the input and output parameters of each operation.

The declarative approach is suitable for modeling the Travel Arrangements process. It can be described by the following rules:

1. It is not possible to *book a flight* without previously *checking its price*.
2. If a *flight is booked*, a *payment* for this booking is required.
3. If the customer wants to book a flight, its *price must be checked* at least in two different airlines.
4. The *check-in* is only available if the flight payment was confirmed.
5. It is not possible to *book a hotel* without *checking its information* before.
6. If a *hotel is booked*, a *payment* for this booking is required.
7. If the *booking payment* was confirmed, then a *voucher must be sent* to the costumer email.
8. If the booking payment was not confirmed, then the *costumer must be notified*.
9. A *discount* must be issued to groups of more than 10 persons that book a hotel.

One can notice that the currency service was not mentioned among the business rules. This means that the user can choose currency operations at any time while executing the business process, which characterizes the flexibility provided by declarative processes.

Fig. 3 shows how REFlex rule engine represents these rules graphically. Each node is an activity available in the process. Rules that have any dependency on context data are represented by arcs annotated with conditions. A rule is active only if the condition is evaluated to true. Otherwise, the rule is simply ignored. On other hand, rules that have no data dependency are always active. Thus, rules are enforced or not depending on the context.

**Table 3.** Services details

| Operations | Input Variables | Output Variables |
|---|---|---|
| *Flight Service* | | |
| checkFlightPrice | from, to, price, date, airline | flightId |
| bookFlight | flightID | bookID |
| payFlight | bookID, value, creditCard | confirmation, paymentCode |
| checkIn | bookID, passportNumber, email | - |
| *Hotel Service* | | |
| checkHotel | hotelName, checkInDate, checkOutDate | roomsAvailable |
| bookHotel | hotelName, checkInDate, checkOutDate, persons | bookID, bookValue |
| payHotelBooking | bookID, value, creditCard | confirmation, paymentCode |
| sendVoucher | bookID, email | - |
| notifyCostumer | email, message | - |
| giveDiscount | bookID, discountPercentage | finalValue |
| *Currency Service* | | |
| convertCurrency | value, fromCurrency, toCurrency | newCurrency |

To better understand how useful data-aware rules are, note the activities *send voucher* and *notify costumer* are dependent form the result of *pay hotel booking*. This result can only be analyzed at run-time. If the business rules were static, it would not be possible to model this dependency. Thus, in this case, there are two *response* rules that rely on the result of *pay hotel booking* activity. Each rule has a condition that expresses what must be checked at run-time. When this activity is executed, the global data *hotelBooked* is updated. If the payment is confirmed, activity *send voucher* is obliged by the *response* rule while *notify costumer* becomes optional. If the payment is not confirmed, *notify costumer* is obliged and activity *send voucher* will be disabled, due to a precedence on itself. Without data-dependent rules, this behavior would be hard to achieve. Thus, data-aware declarative models are more intuitive and expressive.

After modeling the activities and their relationship, we are able to link activities to their corresponding web service operations. The orchestrator uses this information to perform the service bindings and invoke operations when activities are executed. Listing 1.1 presents an excerpt of the XML definition of our process' variables and activity bindings. It shows how the *Pay Flight* activity is bound to the *payFlightBooking* operation of the *FlightService*. This activity references local variables, whose values are provided by the user prior to its

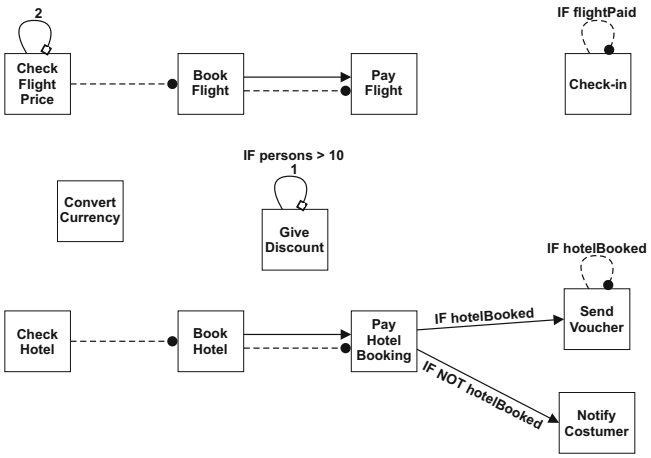



**Fig. 3.** REFlex model of the case study

execution. It also references global variables, which may be set by the user or by other services. For example the input parameter *bookValue* is the return of the *bookFlight* operation.

```
<process name="TravelProcess">
  <globalData>
    ...
    <variable name="bookID" type="STRING"/>
    <variable name="flightPaid" type="BOOLEAN"/>
    <variable name="flightPaymentCode" type="INT"/>
    ...
  </globalData>
  <activities>
    ...
    <activity name="Pay␣Flight">
      <serviceBinding operation="payFlightBooking"
      wsdlUrl="http://...FlightService?wsdl"
      portType="FlightServicePortType"
      binding="FlightServiceSOAP11Binding"/>
      <dataInputBinding>
        <variableBinding variableName="bookID" global="true"
        expression="xpath:/payFB/bookID"/>
        <variableBinding variableName="value" global="false"
        type="DOUBLE" expression="xpath:/payFB/value"/>
        <variableBinding variableName="creditCard" global="false"
        type="STRING" expression="xpath:/payFB/creditCard"/>
      </dataInputBinding>
      <dataOutputBinding>
        <variableBinding variableName="flightPaid"
        expression="//payFB/result/confirmation"/>
        <variableBinding variableName="flightPaymentCode"
        expression="//payFB/result/paymentCode"/>
      </dataOutputBinding>
    </activity>
    ...
  </activities>
</process>
```

**XML 1.1.** Snipet of the process definition of the case study

## 7 Conclusions

This work proposes a web-service orchestrator for declarative business processes, called REFlex. This kind of business process rely on business rules to describe the behavior of the process and to control the execution of process instances. Hence, the flow of activities is only determined at run-time.

Current engines for the enactment of declarative processes are not completely integrated with current SOA technologies. On the other hand, all professional engines for traditional workflow execution recognize the necessity for integration with web services. It is our contend that a web service orchestrator capable of interpreting declarative models brings enormous benefits to the field. It allows for the construction of semi-automated, flexible business processes where process participants interact with external tools to exchange data and reuse functionality. In this way, the flexibility of declarative models can be complemented by the efficiency offered by automation.

REFlex adopts both a graphical and an XML-based languages for the description of declarative business models. The graphical notation is useful for communication with business administrators. The XML model is used to model the technical details of the process, such as its data variables and web service bindings. SOAP messages are automatically constructed at run-time to perform the service invocations requested by the user.

REFlex uses a novel rule engine that offers the efficient and deadlock-free execution of declarative business processes [4]. Furthermore, a unique feature of REFlex engine among related work is its capacity to interpret data-aware rules. These rules depend on context information and enhance the expressive power of the modeling language.

To demonstrate our approach, we described a declarative business process modeled in REFlex notation. The process make use of web service bindings and data-dependent rules. Albeit simple, it illustrates the usefulness of such features for real world applications.

## References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The sciff framework. ACM Trans. Comput. Logic 9(4), 29:1–29:43 (2008)
2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S.: et al. Business process execution language for web services (2003)
3. de Carvalho, R.M., Silva, N.C., Oliveira, C.A.L., Lima, R.M.: Reflex: an efficient graph-based rule engine to execute declarative processes. In: Proceedings of the International Conference on Systems, Man and Cybernetics (2013)
4. de Carvalho, R.M., Silva, N.C., Oliveira, C.A.L., Lima, R.M.: A solution to the state space explosion problem in declarative business process modeling. In: Proceedings of the 25th International Conference on Software Engineering and Knowledge Engineering (2013)

5. Charfi, A., Mezini, M.: Ao4bpel: An aspect-oriented extension to bpel. World Wide Web 10(3), 309–344 (2007)
6. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)
7. Hermosillo, G., Seinturier, L., Duchien, L.: Using complex event processing for dynamic business process adaptation. In: 2010 IEEE International Conference on Services Computing (SCC), pp. 466–473 (July 2010)
8. Hildebrandt, T.T., Mukkamala, R.R.: Declarative event-based workflow as distributed dynamic condition response graphs. In: PLACES, pp. 59–73 (2010)
9. Koning, M., Sun, C.-A., Sinnema, M., Avgeriou, P.: Vxbpel: Supporting variability for web services in bpel. Inf. Softw. Technol. 51(2), 258–269 (2009)
10. Nurcan, S.: A survey on the flexibility requirements related to business processes and modeling artifacts. In: HICSS 2008: Proceedings of the 41st Annual Hawaii International Conference on System Sciences, p. 378. IEEE Computer Society, Washington, DC (2008)
11. Oliveira, C., Silva, N., Sabat, C., Lima, R.: Reducing the gap between business and information systems through complex event processing. Computing and Informatics 32(2) (2013)
12. Pesic, M., Schonenberg, H., van der Aalst, W.M.P.: Declare: Full support for loosely-structured processes. In: 11th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2007, p. 287 (October 2007)
13. Pesic, M.: Constraint-Based Workflow Management Systems: Shifting Control to Users. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands (2008)
14. Silva, N.C., de Carvalho, R.M., Oliveira, C.A.L., Lima, R.M.: Integrating declarative processes and soa: A declarative web service orchestrator. In: Proceedings of the 2013 International Conference on Semantic Web and Web Services (2013)
15. van der Aalst, W.M.P., Pesic, M.: Decserflow: Towards a truly declarative service flow language. In: Leymann, F., Reisig, W., Thatte, S.R., van der Aalst, W.M.P. (eds.) The Role of Business Processes in Service Oriented Architectures, July 16-July 21. Dagstuhl Seminar Proceedings, vol. 06291. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
16. Weigand, H., van den Heuvel, W.-J., Hiel, M.: Business policy compliance in service-oriented systems. Information Systems 36(4), 791–807 (2011), ¡/ce:title¿ Selected Papers from the 2nd International Workshop on Similarity Search and Applications SISAP 2009 ¡/ce:title¿
17. Xiao, Z., Cao, D., You, C., Mei, H.: Towards a constraint-based framework for dynamic business process adaptation. In: Proceedings of the 2011 IEEE International Conference on Services Computing, SCC 2011, pp. 685–692. IEEE Computer Society, Washington, DC (2011)