A Guided Tour through SAVVY-WS: a Methodology for Specifying and Validating Web Service Compositions

Domenico Bianculli[1], Carlo Ghezzi[2], Paola Spoletini[3], Luciano Baresi[2], and Sam Guinea[2]

[1] University of Lugano
Faculty of Informatics
via G. Buffi 13, CH-6900, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch
[2] Politecnico di Milano
DEEP-SE Group - Dipartimento di Elettronica e Informazione
piazza L. da Vinci, I-20133, Milano, Italy
{carlo.ghezzi, sam.guinea, luciano.baresi}@polimi.it
[3] Università dell'Insubria
Dipartimento di Scienze della Cultura, Politiche e dell'Informazione
via Carloni 78, I-22100, Como, Italy
paola.spoletini@uninsubria.it

Abstract. Service-Oriented Architectures are emerging as a promising solution to the problem of developing distributed and evolvable applications that live in an open world. We contend that developing these applications not only requires adopting a new architectural style, but more generally requires re-thinking the whole life-cycle of an application, from development time through deployment to run time. In particular, the traditional boundary between development time and run time is blurring. Validation, which traditionally pertains to development time, must now extend to run time. In this paper, we provide a tutorial introduction to SAVVY-WS, a methodology that aims at providing a novel integrated approach for design-time and run-time validation. SAVVY-WS has been developed in the context of Web service-based applications, composed via the BPEL workflow language.

1 Introduction

Software systems have been evolving from having static, closed, and centralized architectures to dynamically evolving distributed and decentralized architectures where components and their connections may change dynamically [1]. In these architectures, *services* represent software components that provide specific functionality, exposed for possible use by many clients. Clients can dynamically discover services and access them through network infrastructures. As opposed to the conventional components in a component-based system, services are developed, deployed, and run by independent parties. Furthermore, additional services can be offered by service aggregators composing third-party services to provide new added-value services.

This emerging scenario is *open*, because new services can appear and disappear, *dynamic*, because compositions may change dynamically, and *decentralized*, because no single authority coordinates all developments and their evolution.

Service-Oriented Architectures (SOAs) have been proposed to support application development for these new settings. An active research community is investigating the various aspects for service-oriented computing; research progress is documented, for example, by the International Conference on Service-Oriented Computing [2]. Several large research projects have also been funded in this area by the European Union, such as, amongst many others, SeCSE [3], PLASTIC [4], and the S-Cube [5] network of excellence, in which the authors are involved. The European Union has also promoted many initiatives to foster services-based software development and research, such as NESSI [6].

We strongly believe that a holistic approach is necessary to develop modern dynamic service-based applications. A coherent and well-grounded methodology must guide an application's life cycle: from development time to run time. SAVVY-WS (Service Analysis, Verification, and Validation methodologY for Web Services) is intended to be a first attempt to contribute to such a methodology, by focusing on lifelong verification of service compositions, which encompasses both design-time and run-time verification. SAVVY-WS is tailored to Web service technologies [7, 8]. The reason of this choice is that although SOAs are in principle technology-agnostic and can be realized with different technologies —such as OSGi, Jini and message-oriented middleware— Web services are the most used technology to implement SOAs, as corroborated by the many on-going standardization efforts devoted to support them. SAVVY-WS has been distilled by research performed in the context of several projects, most notably the EU IST SeCSE [3, 9] and PLASTIC [4] projects, and the Italian Ministry of Research projects ART DECO [10] and DISCoRSO [11]. A preliminary evaluation of the use of SAVVY-WS has been reported in [12–14]. SAVVY-WS is supported by several prototype tools that are currently being integrated in a comprehensive design and execution environment.

This paper provides a tutorial introduction to SAVVY-WS. Section 2 briefly summarizes the main features of the BPEL language, which is used for service compositions, and the ALBERT language, which is used to formally specify properties. Section 3 gives an overview of SAVVY-WS, which is based on ALBERT, a design-time verification environment based on model checking, and a run-time monitoring environment. Section 4 introduces two running examples that will be used throughout the rest of the paper. Section 5 discusses how ALBERT can be used as a specification language for BPEL processes. Section 6 shows how verification is performed at design time via model checking, while Sect. 7 shows how continuous verification of the service composition can be achieved at run time. Section 8 discusses the related work. Section 9 provides some final conclusions.

2 Background Material

2.1 BPEL

BPEL —Business Process Execution Language (for Web Services)— is a high-level XML-based language for the definition and execution of business processes [15]. It supports the definition of workflows that provide new services, by composing external Web services in an orchestrated manner. The definition of a workflow contains a set of global variables and the workflow logic is expressed as a composition of *activities*; variables and activities can be defined at different visibility levels within the process using the *scope* construct.

Activities include primitives for communicating with other services (*receive*, *invoke*, *reply*), for executing assignments (*assign*) to variables, for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of a process (*terminate*). Moreover, conventional constructs like *sequence, while,* and *switch* provide standard control structures to order activities and to define loops and branches. The *pick* construct makes the process wait for the arrival of one of several possible incoming messages or for the occurrence of a time-out, after which it executes the activities associated with the event.

The language also supports the concurrent execution of activities by means of the *flow* construct. Synchronization among the activities of a *flow* may be expressed using the *link* construct; a link can have a guard, which is called *transitionCondition.* Since an activity can be the target of more than one link, it may define a *joinCondition* for evaluating the *transitionCondition* of each incoming link. By default, if the *joinCondition* of an activity evaluates to false, a fault is generated. Alternatively, BPEL supports *Dead Path Elimination*, to propagate a false condition rather than a fault over a path, thus disabling the activities along that path.

Each *scope* (including the top-level one) may contain the definition of the following handlers:

– An *event handler* reacts to an event by executing —concurrently with the main activity of the *scope*— the activity specified in its body. In BPEL there are two types of events: message events, associated with incoming messages, and alarms based on a timer.
– A *fault handler* catches faults in the local *scope.* If a suitable *fault handler* is not defined, the fault is propagated to the enclosing *scope.*
– A *compensation handler* restores the effects of a previously completed transaction. The *compensation handler* for a *scope* is invoked by using the *compensate* activity, from a *fault handler* or *compensation handler* associated with the parent *scope.*

The graphical notation for BPEL activities used in the rest of the paper is shown in Fig. 1; it has been devised by the authors and it is freely inspired by BPMN [16].

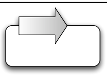
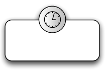
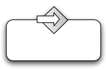
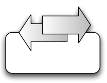
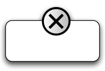
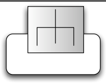
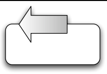
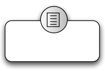
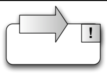
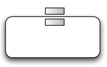
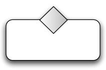
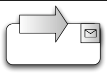
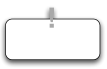
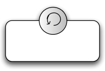
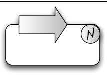
Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

Fig. 1. Graphical notation for BPEL

2.2 ALBERT

ALBERT [12] is an assertion language for BPEL processes, designed to support both design-time and run-time validation.

ALBERT formulae predicate over *internal* and *external* variables. The former consist of data pertaining to the internal state of the BPEL process in execution. The latter are data that are considered necessary to the verification, but are not part of the process' business logic and must be obtained by querying external data sources (e.g., by invoking other Web services, or by accessing some global, persistent data representing historical information).
ALBERT is defined by the following syntax:

$$\begin{array}{l}
\phi ::= \chi \quad | \quad \neg\phi \quad | \quad \phi \wedge \phi \quad | \quad (\ \mathsf{op}\ \mathsf{id}\ \mathsf{in}\ \mathsf{var}\ ;\ \phi\ ) \quad | \\
\mathit{Becomes}(\chi) \quad | \quad \mathit{Until}(\phi,\phi) \quad | \quad \mathit{Between}(\phi,\phi,K) \quad | \quad \mathit{Within}(\phi,K) \\
\chi ::= \psi\ \mathsf{relop}\ \psi \quad | \quad \neg\chi \quad | \quad \chi \wedge \chi \quad | \quad \mathit{onEvent}(\mu) \\
\psi ::= \mathsf{var} \quad | \quad \psi\ \mathsf{arop}\ \psi \quad | \quad \mathsf{const} \quad | \quad \mathit{past}(\psi, \mathit{onEvent}(\mu), n) \quad | \\
\mathit{count}(\chi, K) \quad | \quad \mathit{count}(\chi, \mathit{onEvent}(\mu), K) \quad | \quad \mathsf{fun}(\psi, K) \quad | \\
\mathsf{fun}(\psi, \mathit{onEvent}(\mu), K) \quad | \quad \mathit{elapsed}(\mathit{onEvent}(\mu)) \\
\mathsf{op} ::= \mathtt{forall} \quad | \quad \mathtt{exists} \\
\mathsf{relop} ::= < \quad | \quad \leq \quad | \quad = \quad | \quad \geq \quad | \quad > \\
\mathsf{arop} ::= + \quad | \quad - \quad | \quad \times \quad | \quad \div \\
\mathsf{fun} ::= \mathit{sum} \quad | \quad \mathit{avg} \quad | \quad \mathit{min} \quad | \quad \mathit{max}
\end{array}$$

where id is an identifier, var is an internal or external variable, *onEvent* is an event predicate, *Becomes*, *Until*, *Between* and *Within* are temporal predicates, *count*, *elapsed*, *past*, and all the functions derivable from the non-terminal fun are temporal functions of the language. Parameter μ identifies an event: the *start* or the *end* of an *invoke* or *receive* activity, the receipt of a message by a *pick* or an *event handler*, or the execution of any other BPEL activity. K is a positive real number, n is a natural number and const is a constant.

The above syntax only defines the language's core constructs. The usual logical derivations are used to define other connectives and temporal operators (e.g., $\vee$, *Always*, *Eventually*, ...). Moreover, the strings derived from the non-terminal ϕ are called *formulae*; the strings derived from the non-terminal ψ are called *expressions*.

The formal semantics of ALBERT is provided in Appendix A.

3 A Bird-eye View of SAVVY-WS

This section illustrates the principles and the main design choices of SAVVY-WS. Its use is then illustrated in depth in the rest of this paper, which shows SAVVY-WS in action through two case studies.

SAVVY-WS's goal is to support the designers of composite services during the validation phase, which extends from design time to run time. SAVVY-WS assumes that service composition is achieved by means of the BPEL workflow language, which orchestrates the execution of external Web services.

Figure 2 summarizes the use of SAVVY-WS within the development process of BPEL service compositions. When a service composition is designed (step 1), SAVVY-WS assumes that the external services orchestrated by the workflow are only known through their specifications. The actual services that will be invoked at run time, and hence their implementation, may not be known at design time. The specification describes not only the syntactic contract of the service (i.e., the operations provided by the service, and the type of their input and output parameters), but also their expected effects, which include both functional and non-functional properties. Functional properties describe the behavioral contract of the service; non-functional properties describe its expected quality, such as its response time.

Specifying functional and non-functional properties only at the level of interfaces is required to support lifelong validation of dynamically evolvable compositions, which massively use late-binding mechanisms. Indeed, at design time a service refers to externally invoked services through their *required* interface. At run time, the service will resolve its bindings with external services that provide a matching interface, i.e., their *provided* interface conforms to the one used at design time.

The SAVVY-WS methodology supports the ALBERT language to specify required service interfaces. The language specifies the required interface in terms of logical formulae, called *assumed assertions* (AAs). Based on the AAs of all services invoked by the workflow, in turn, the composition may offer a service whose properties can also be specified via ALBERT formulae, called *guaranteed assertions* (GAs). Therefore, the second step of the SAVVY-WS-aware development process is to annotate the BPEL process with assumed and guaranteed assertions written in ALBERT (step 2 in Fig. 2).

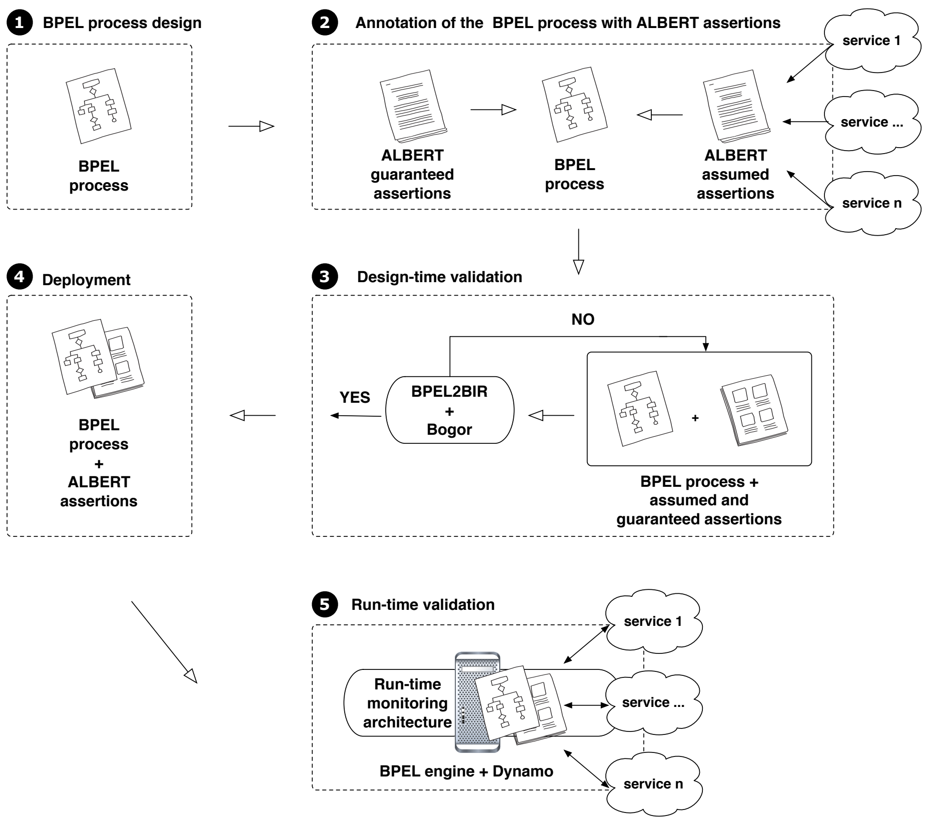

1 BPEL process design
BPEL process
2 Annotation of the BPEL process with ALBERT assertions
ALBERT guaranteed assertions
BPEL process
ALBERT assumed assertions
service 1
service ...
service n
4 Deployment
BPEL process + ALBERT assertions
3 Design-time validation
NO
YES
BPEL2BIR + Bogor
+
BPEL process + assumed and guaranteed assertions
5 Run-time validation
Run-time monitoring architecture
BPEL engine + Dynamo
service 1
service ...
service n

Fig. 2. SAVVY-WS-aware development process

The SAVVY-WS methodology is supported at design time by a formal verification tool (Bpel2Bir) that is used to check (step 3 in Fig. 2) that a composite service delivers its expected functionality and meets the required quality of service (both specified in ALBERT as GAs), under the assumption that the external services used in the composition fulfill their required interfaces (specified in ALBERT as AAs). The SAVVY-WS verification tool is based on the Bogor model checker [17].

Design-time verification does not prevent errors from occurring at run time. In fact, there is no guarantee that a service implementation eventually fulfills the contract promised through its provided interface. The service provider may either be malicious, by offering a service with an inferior experienced quality of service and/or a wrong functionality to increase its revenue on the service provision, or it might change the service implementation as part of its standard maintenance process: in this case, a service that worked properly might be changed in a new version that violates its previous contract.

Furthermore, during design-time verification, it is not possible to model the behavior of the underlying distributed infrastructure, which plays an important role in the provision of networked services. Although service providers' specifications could take into account, to some extent, the role of the distributed infrastructure, it is virtually impossible to foresee all possible conditions of the infrastructure components (e.g., network links) at design time.

To solve these problems, SAVVY-WS supports continuous verification by transforming —when a BPEL process is deployed on a BPEL execution engine (step 4 in Fig. 2)— ALBERT formulae into run-time assertions that are monitored (step 5 in Fig. 2) by Dynamo —our monitoring framework embedded within the BPEL engine— to check for possible deviations from the correct behavior verified at design time. If a deviation is caught, suitable compensation policies and recovery actions should be activated.

4 Running Examples

In this section, we describe the two running examples used in the rest of this paper to illustrate our lifelong validation methodology.

The first example is inspired by one of the scenarios developed in the context of the EU IST project SENSORIA [18]. We considered the *On Road Assistance* scenario, which takes place in an automotive domain, where a SOA interconnects (the devices running on) a car, service centers providing facilities like car repair, towing and car rental, and other actors. As will be described in Sect. 5.1, this example is used to show how to express (and validate) in ALBERT properties related to the timeliness of events.

The second example is inspired by a similar one described in [19, 20] and it illustrates a BPEL process realizing a *Car Rental Agency* service. It interacts with a *Car Broker Service*, which controls the operations of the branch; with a *User Interaction Service*, through which customers can make car rental requests; with a *Car Information Service*, which maintains a database of cars availability and allocates cars to customers; with a *Car Parking Sensor Service*, which exposes as a Web service the sensor that senses cars as they are driven in or out of the car parking of the branch. As will be illustrated in Sect. 5.2, this example will be used to show how to express ALBERT properties about sequences of events.

4.1 Example 1: *On Road Assistance*

The *On Road Assistance* process (depicted in Fig. 3), is supposed to run on an embedded module in the car and is executed after a breakdown, when the car becomes not driveable.

The *Diagnostic System* sends a message with diagnostic data and the driver's profile (which contains credit card data, the allowed amount for a security deposit payment, and preferences for selecting assistance services) to the workflow, which

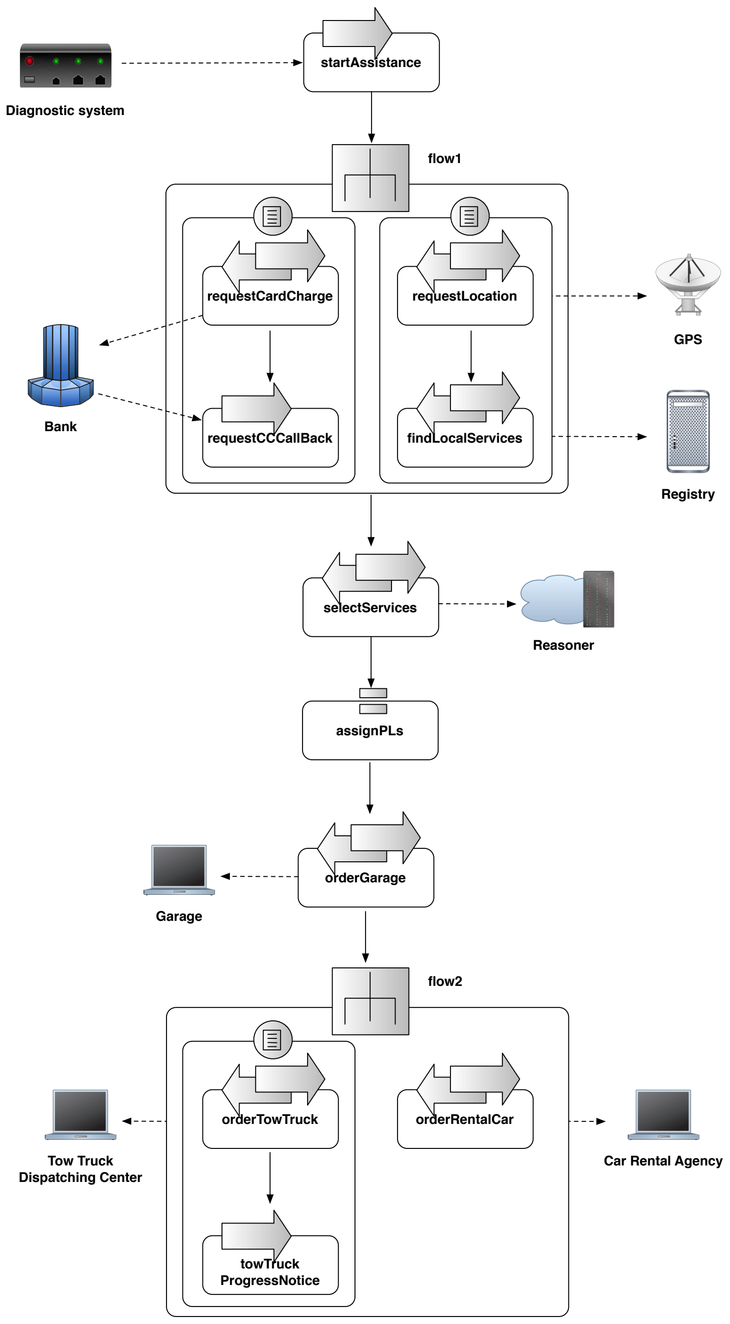


Fig. 3. The *On Road Assistance* BPEL process

starts by executing the `startAssistance` *receive* activity. Then, it starts a *flow* (named `flow1`) containing two parallel *sequence*s of activities.

In one *sequence*, the process first requests the *Bank* service to charge the driver's credit card with a security deposit payment, by invoking the operation `requestCardCharge` and passing the credit card data and the amount of the payment. Then, it waits for the asynchronous reply of the *Bank*, modeled by the `requestCCCallBack` *receive* activity.

In the other parallel *sequence*, the process first asks the *GPS* service —which represents a Web service interface for the GPS device installed on the car— to provide the position of the car (`requestLocation` *invoke* activity). The returned location is then used to query (`findLocalServices` *invoke* activity) a *Registry* to discover appropriate services close to the area where the car pulled out. The *Registry* service will return a sequence of triples —each of which contains a suitable combination of locally available services providing car repair shops, car rental, and tow trucking— stored in the `foundServices` process variable.

Subsequently, this variable is used as an input parameter in the `selectServ-ices` operation of the *Reasoner* service, which is supposed to select the best available service triple matching the driver's preferences, and to store the selected services' endpoint references in the `bestServices` process variable. After assigning (`assignPLs` *assign* activity) the endpoint references to *partner link*s corresponding to the *Garage*, *Car Rental Agency* and *Tow Truck Dispatching Center* services, the process first sets an appointment with the garage, by sending to it the car diagnostic data (`orderGarage` *invoke* activity). The garage acknowledges the appointment by sending back the actual location of the repair shop.

Afterwards, the process starts a *flow* (named `flow2`) with three activities. Two activities are grouped in a *sequence*, where the process first contacts the towing service dispatching center (`orderTowTruck` *invoke* activity), and then it waits for an acknowledgment message `ack` confirming that a tow truck is in proximity of the car; this message is consumed by the `towTruckProgressNotice` *receive* activity.

The other activity is executed in parallel to the *sequence* mentioned above, and is used to contact the car rental agency (`orderRentalCar` *invoke* activity). In both *invoke* activities of `flow2`, the garage location is sent as an input parameter, representing the coordinates where the car is to be towed to and where the rental car is to be delivered.

To keep the example simple, we assume that at least one service triple is retrieved after invoking the *Registry*, and that the selected garage, towing service, and car rental agency can cope with the received requests.

4.2 Example 2: *Car Rental Agency*

The *Car Rental Agency* process (sketched in Fig. 4) is supposed to run on the information system of a local branch of a car rental company.

The process starts as soon as it receives a message from the *Car Broker Service* (`startRental` *receive* activity). Then, the process enters an infinite loop:

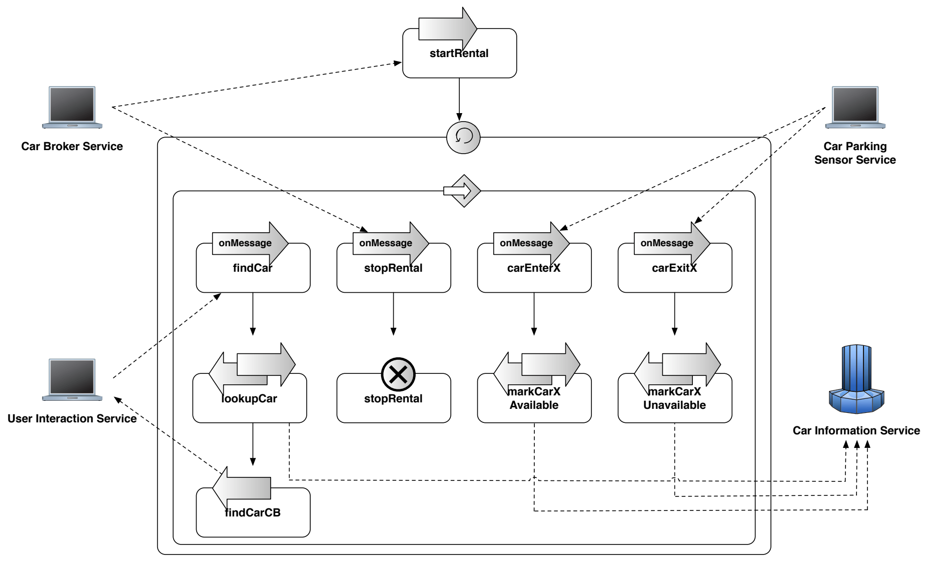


Fig. 4. *Car Rental Agency* BPEL process

every iteration is a *pick* activity that suspends the execution and waits for one of the following four messages:

– `findCar`. A customer asks to rent a car and provides her preferences (e.g., the car model). Then, the process checks the availability of a car that matches customer's preferences by invoking the `lookupCar` operation on the *Car Information Service*. The result of this operation, which can be either a negative answer or an identifier corresponding to the digital key to access the car, is returned to the customer with the `findCarCB` *reply* activity.
– `carEnterX` and `carExitX`. These two messages are sent out by the *Car Parking Sensor Service* when a car enters (respectively, exits) the car parking. The process reacts to this information by updating the cars database, invoking, respectively, the `markCarAvailableX` and `markCarUnavailableX` operations on the *Car Information Service*. Actually, the `X` in the name of each message or operation is a placeholder for a unique id associated with a car; therefore, if `A` is a car id, the actual messages associated with it have the form `carEnterA` and `carExitA`, and the corresponding operations are named `markCarAvailableA` and `markCarUnAvailableA`.
– `stopRental`. The *Car Broker Service* stops the operations of the local branch, making the process terminate.

To keep the example simple, we assume that the local branch where the *Car Rental Agency* process is run is the only one accessing the *Car Information Service* and that cars in the car parking can be only rented through the local car rental branch.

5 Specifying Services with ALBERT

The ALBERT language defines formulae that specify *invariant* assertions for a BPEL process. Two kinds of assertions can be specified using ALBERT:

- *assumed assertions (AAs)*, which define the properties that partner services are required to fulfill when interacting with the BPEL process;
- *guaranteed assertions (GAs)*, which define the properties that the composite service should satisfy, assuming that external services operate as specified.

Both kinds of assertions allow for stating functional and non-functional properties of services. As an example, an AA that should hold after the execution (as a post-condition) of an *invoke* activity `Act` on an external service S can be written in the following form:

$$\mathit{onEvent}(\texttt{end\_Act}) \rightarrow \texttt{\$myVar=EDS::getData()/var}$$

The antecedent of the formula contains the *onEvent* predicate, which is used to identify a specific point in the execution of the workflow. This point is represented by its argument: in this case, the keyword `end` denotes the point right after the end of the execution of activity `Act`. The consequent of the formula states that the value of the internal variable `myVar` of the process (the variable returned after invoking service S) must be equal to the value obtained by accessing an external data source (the *EDS* Web service endpoint), invoking the `getData` operation on it and retrieving the `var` part from the return message.

It is also possible to express non-functional AAs, such as latency in a service response. The following ALBERT formula specifies that the duration of an *invoke* activity `Act` should not exceed 5 time units:

$$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 5)$$

As in the previous formula, the antecedent identifies a certain point in the execution of the process where the consequent should then hold to make the assertion evaluate to true. This point is represented by the event corresponding to the `start` of activity `Act`. The consequent contains the *Within* operator, which evaluates to true if its first argument evaluates to true within the temporal bound expressed by the second argument; otherwise it evaluates to false. In other words, the consequent states that the event corresponding to the `end` of activity `Act` should occur within 5 time units from the current instant, which is the time instant in which the antecedent of the formula is true. We leave the choice of the most suitable timing granularity to the verification engineer, who can then properly convert the informal system requirements to formal, real-time specifications [21].

ALBERT can also be used to express GAs. For example, one may state an upper bound to the duration of a certain sequence of activities, which includes external service invocations, performed by a composite BPEL workflow in response to a user's input request.

As said above, ALBERT can be used to specify both AAs and GAs for BPEL processes. However, when defining AAs, formulae should only refer to the BPEL activities that are responsible for interacting with external services. Typically, AAs express properties that must hold after the workflow has completed an interaction with an external service. Hereafter we list a few specification templates which proved to be useful to express AAs in practical cases. In the templates, μ is an event identifying the start or the end of an *invoke* or *receive* activity, the reception of a message by a *pick*, or an *event handler*; ϕ and χ are ALBERT formulae; ψ and ψ' are ALBERT expressions; n is a natural number which is used to retrieve a certain value upon the nth-last occurrence of event μ in the past; K is a positive real number denoting time distances; fun is a placeholder for any function (e.g., average, sum, minimum, maximum) that can be applied to sets of numerical values.

- $\mathit{onEvent}(\mu) \rightarrow \phi$: it allows for checking property ϕ only in the states preceding or following an interaction with an external service;
- $\mathit{past}(\psi',\ \mathit{onEvent}(\mu),\ n) =\ \psi \rightarrow \phi$: it allows for checking property ϕ on the basis of past interactions of the workflow with the external world;
- $\mathit{Becomes}(\mathit{count}(\chi,\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain number of specific events;
- $\mathit{Becomes}(\mathsf{fun}(\psi',\ \mathit{onEvent}(\mu),\ K) =\ \psi) \rightarrow \phi$: it allows for checking property ϕ only if past interactions with the external world have led to a certain value of an aggregate function.

5.1 Specifying the *On Road Assistance* Process

Hereafter we provide some properties of the *On Road Assistance* process: each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

- BankResponseTime
  After requesting to charge the credit card, the *Bank* will reply within 4 minutes when a low-cost communication channel is used, and it will reply within 2 minutes if a high-cost communication channel is used. In ALBERT this AA can be expressed as follows:

$$\begin{aligned}&\mathit{onEvent}(\texttt{end\_requestCardCharge}) \rightarrow\\ &\qquad(\texttt{VCG::getConnection()/cost='low'}\wedge\\ &\qquad\ \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 4) \vee\\ &\qquad(\texttt{VCG::getConnection()/cost='high'}\wedge\\ &\qquad\ \mathit{Within}(\mathit{onEvent}(\texttt{start\_requestCCCallBack}), 2))\end{aligned}$$

  where `VCG` is the Web service interface for the local vehicle communication gateway, providing contextual information on the communications channels currently in use within the car. `VCG::getConnection()/cost` represents an external variable retrieved by invoking the `getConnection` operation on the `VCG` service and accessing the `cost` part of the returned message.

– AllButBankServicesResponseTime
  The interactions with all external services but the *Bank*, namely *GPS*, *Registry*, *Reasoner*, *Garage*, *Tow Truck Dispatching Center* and *Car Rental Agency* will last at most 2 minutes. This AA is expressed as a conjunction of formulae, each of which follows the pattern:

$$\mathit{onEvent}(\texttt{start\_Act}) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{end\_Act}), 2)$$

  where `Act` ranges over the names of the *invoke* activities interacting with the external services listed above.

– AvailableServicesDistance
  The *Registry* will return services whose distance from the place where the car pulled out is less than 50 miles. This AA can be expressed as follows:

```
onEvent(end_findLocalServices) →
                    (forall t in $foundServices/[*] ;
                        (forall s in $foundServices/t/[*] ;
                        s/distance < 50))
```

  where `foundServices` contains a sequence of triples, where elements contain a `distance` message part.

– TowTruckServiceTimeliness
  The *Tow Truck Dispatching Center* service selected by the *Reasoner* will provide assistance within 50 minutes from the service request. This AA can be expressed as follows:

$$\mathit{onEvent}(\texttt{end\_selectServices}) \rightarrow (\texttt{\$bestServices/towing/ETA} \leq 50)$$

  where the `ETA` message part represents the maximum time bound guaranteed by a service to provide assistance.

– TowTruckArrival
  The time interval between the end of the order of a tow truck and the arrival of the `ack` message (notifying that the tow truck is in proximity of the car) is bounded by the ETA of the *Tow Truck Dispatching Center* service, that is 50 minutes. This AA can be expressed as follows:

$$\mathit{onEvent}(\texttt{end\_OrderTowTruck}) \rightarrow$$
$$\mathit{Within}(\ \mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 50)$$

– AssistanceTimeliness
  The tow truck that will be requested will be in proximity of the car within 60 minutes after the credit card is charged. This property must be guaranteed to the user by the *On Road Assistance* process. It is a GA, whose validity is

(rather trivially) assured at design time by the AllButBankServicesResponseTime, the TowTruckServiceTimeliness and the TowTruckArrival AAs, and by the structure of the process. The property can be expressed as follows:

$$\begin{aligned}&\mathit{onEvent}(\texttt{end\_requestCCCallBack}) \rightarrow \\ &\qquad\qquad \mathit{Within}(\mathit{onEvent}(\texttt{start\_TowTruckProgressNotice}), 60)\end{aligned}$$

5.2 Specifying the *Car Rental Agency* Process

Hereafter we provide some properties of the *Car Rental Agency* process. As we did in the previous section, each property is first stated informally and then in ALBERT, followed by an additional clarifying comment, when necessary.

- ParkingInOut
  Between two events signaling that a car exits the car parking, an event signaling the entrance for the same car must occur. This AA can be expressed[4] as a conjunction of formulae, each of which follows the pattern:

  $$\mathit{onEvent}(\texttt{carExitX}) \rightarrow \mathit{Until}(\neg \mathit{onEvent}(\texttt{carExitX}), \mathit{onEvent}(\texttt{carEnterX}))$$

  where `X` ranges over the identifiers of the cars available in the local rental branch. Moreover, this formula can be combined, using a logical AND, with a similar constraint that refers to the `carEnterX` message.
- CISUpdate
  If a car is marked as available in the *Car Information Service*, and the same car is not marked as unavailable until a `lookupCar` operation for that car is invoked, then the `lookupCar` operation should not return a negative answer. This AA on the behavior of the *Car Information Service* can be expressed[5] as a conjunction of formulae, each of which follows the pattern:

  $$\begin{aligned}&(\mathit{onEvent}(\texttt{end\_markAvailableX}) \wedge \\ &\qquad\qquad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{end\_markUnavailableX}), \\ &\qquad\qquad\qquad \mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X})) \\ &\quad \rightarrow \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_lookupCar}) \wedge \texttt{\$carInfo/id=X} \wedge \\ &\quad \mathit{Eventually}((\mathit{onEvent}(\texttt{end\_lookupCar}) \wedge \texttt{\$queryResult/res!="no"})))\end{aligned}$$

  where `X` ranges over the identifiers of the cars available in the local rental branch, `carInfo` is the input variable of the `lookupCar` operation, whose output variable is `queryResult`.

[4] The semantics of the *Until* operator described in Appendix A guarantees that its first argument will not be evaluated in the current state.

[5] A more complete specification should also include that two `lookupCar` operations for the same car could not happen at the same time. However, this is guaranteed by the structure of the workflow.

– RentCar
  If a car enters in the car parking, and the same car does not exit until a customer requests it for renting, then this request should not return a negative answer. This is a GA, whose validity is (rather trivially) assured at design time by the ParkingInOut and CISUpdate AAs, and by the structure of the process. The property can be expressed as a conjunction of formulae, each of which follows the pattern:

$$\begin{aligned}&(\mathit{onEvent}(\texttt{carEnterX}) \wedge \\ &\qquad \mathit{Until}(\neg\, \mathit{onEvent}(\texttt{carExitX}), \\ &\qquad\qquad \mathit{onEvent}(\texttt{start\_findCar}) \wedge \texttt{\$carInfo/id=X})) \\ &\rightarrow \mathit{Eventually}(\mathit{onEvent}(\texttt{start\_findCar}) \wedge \texttt{\$carInfo/id=X} \wedge \\ &\mathit{Eventually}((\mathit{onEvent}(\texttt{start\_findCarCB}) \wedge \texttt{\$queryResult/res!="no"})))\end{aligned}$$

  where X ranges over the identifiers of the cars available in the local rental branch, carInfo is the input variable of the findCar message, queryResult is the variable returned to the *User Interaction Service* by the findCarCB *reply* activity.

6 Design-time Verification

Our design-time verification phase is based on model checking. We developed Bpel2Bir, a tool that translates a BPEL process and its ALBERT properties into a model suitable for the verification with the Bogor model checker [17]. In the rest of this section, we illustrate, with the help of some code snippets, how the two running examples and their ALBERT properties are translated into BIR (Bogor's input language).

6.1 Example 1: Model Checking the *On Road Assistance* Process

A BPEL process is mapped onto a BIR **system** composed of threads that model the main control flow of the process and its *flow* activities.

Data types are defined by using an intuitive mapping between WSDL messages/XML Schema types and BIR primitive/record types. In this mapping, XML schema simple types (e.g., xsd:int, xsd:boolean) correspond to their equivalent ones in BIR (e.g., **int** and **boolean**). Moreover, the mapping also supports some XML schema facets, such as restrictions on values over integer domains (e.g., minInclusive) and enumeration, which is translated into an enumeration type. For example, the message that is sent by the *Diagnostic System* to the process, contains diagnostic data and the driver's profile (which includes credit card data, the allowed amount for the security deposit payment and preferences for selecting assistance services). This complex type can be modeled as follows, using a combination of record types in BIR:

```
enum TDiagnosticData {dd1, dd2}
enum TCustomerPreference {cp1, cp2}
enum TCreditCard {cc_c1, cc_c2}

record TStartMsg {
  TDiagnosticData diagData;
  TCreditCard ccData;
  int (1,10) deposit;
  TCustomerPreference cpData;
}
```

where we assume, based on the WSDL specification associated with the BPEL process, that the amount for the security deposit payment is an integer value between 1 and 10 and that dd1, dd2, cp1, cp2, cc_c1 and cc_c2 are enumeration values.

The input variables of *receive* activities and the output variables of *invoke* activities, whose values result from interactions with external services, can be modeled using non-deterministic assignments. For example, the `startAssistance` *receive* activity can be modeled as follows:

```
TStartMsg startMsg;
// other code
startMsg := new TStartMsg;
choose
  when <true> do startMsg.diagData:=TDiagnosticData.dd1;
  when <true> do startMsg.diagData:=TDiagnosticData.dd2;
end
// same pattern for generating credit card data
// and customer's preferences
choose
  when <true> do startMsg.deposit :=1;
  when <true> do startMsg.deposit :=2;
         ...
  when <true> do startMsg.deposit :=9;
  when <true> do startMsg.deposit :=10;
end
```

Activities nested within a *flow* are translated into separated threads. In our example (see Fig. 3), `flow1` contains two *sequence* activities; `flow2` contains a *sequence* and an *invoke* activity. For each of these activities, we declare a corresponding global **tid** (thread id) variable:

```
tid flow1_sequence1_tid;
tid flow1_sequence2_tid;
tid flow2_sequence1_tid;
tid flow2_invoke1_tid;
```

For each activity in the *flow* we declare a thread, named after the corresponding **tid** variable. This thread contains the code that models the execution of the corresponding activity. For example, the thread corresponding to the *sequence* that includes `requestCardCharge` and `requestCCCallBack` activities, has the following structure:

```
thread flow1_sequence1() {
// code modeling requestCardCharge
// code modeling requestCCCallBack
exit;
}
```

Finally, the actual execution of a *flow* is translated into the invocation of a helper function launchAndWaitFlow$_i$, which creates and starts a thread for each activity in the flow, and returns to the caller only when all the launched threads terminate. This function has the following form (in the case of `flow1`):

```
function launchAndWaitFlow1() {
  boolean temp0;
  loc loc0: do {
    flow1_sequence1_tid := start flow1_sequence1();
    flow1_sequence2_tid := start flow1_sequence2();
  } goto loc1;

  loc loc1: do {
    temp0 := threadTerminated(flow1_sequence1_tid)
             && threadTerminated(flow1_sequence2_tid);
  } goto loc2;

  loc loc2: when temp0 do{} return;
            when !temp0 do{} goto loc1;
}
```

The `assignPL` activity is not translated since it only updates the partner link references of the process and thus it does not change the state of the process.

Once the basic model of the BPEL process has been created, it can be then enriched by exploiting *assumed assertions*. AAs can provide a better abstraction of the values deriving from the interaction with external services and they can also express constraints on the timeliness of the activities involving external services.

For example, property TowTruckServiceTimeliness represents a constraint on the value of variable `bestServices`. This means that we can restrict the range of the values that can be non-deterministically assigned to that variable, when modeling the output variable of the `selectServices` activity. This is shown in the following code snippet:

```
choose
  when <true> do bestServices.towing.ETA :=1;
  when <true> do bestServices.towing.ETA :=2;
          ...
  when <true> do bestServices.towing.ETA :=49;
  when <true> do bestServices.towing.ETA :=50;
end
```

The next example shows how AAs can be used to define time constraints for modeling either the execution time of, or the time elapsed between BPEL activities. The adopted technique is based on previous work on model checking temporal metric specifications [22]. We insert a code block that randomly generates the duration of the activity within a certain interval, bounded by the value specified in an AA. For *flow* activities, the time consumed by the *flow* is the maximum time spent along all paths. By focusing on `flow2` (see Fig. 3) of our example and using properties AllButBankServiceResponseTime and TowTruckArrival, we get the following code:

```
int (0,52) flow2_sequence1_clock;
int (0,2) flow2_invoke1_clock;
//other code
thread flow2_sequence1() {
  // code modeling orderTowTruck
  choose
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 1;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 2;
        end
  //code modeling TowTruckProgressNotice
  choose
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 1;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 2;
                 ...
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 49;
    when <true> do flow2_sequence1_clock :=
        flow2_sequence1_clock + 50;
  end
}

thread flow2_invoke1() {
  // code modeling orderRentalCar
  choose
    when <true> do flow2_invoke1_clock :=
        flow2_invoke1_clock + 1;
    when <true> do flow2_invoke1_clock :=
        flow2_invoke1_clock + 2;
  end
}
//other code
active thread MAIN {
  //other code
  launchAndWaitFlow2();
  if flow2_sequence1_clock >= flow2_invoke1_clock do
```

```
    assistanceTimeliness_clock :=
      assistanceTimeliness_clock + flow2_sequence1_clock;
  else do
    assistanceTimeliness_clock :=
      assistanceTimeliness_clock + flow2_invoke1_clock;
  end
  //other code
}
```

The first two lines of the previous code snippet represent the declarations of local counters associated with the activities included in the *flow* (in this case a *sequence* and an *invoke*). The domain of these variables is bounded by the duration of each activity, as expressed in an AA; for structured activities (e.g., a *sequence*), we take as upper-bound the sum of the durations of all nested activities.

Each of these counters is then non-deterministically incremented in the body of the thread that simulates the execution of an activity. After the end of the execution of the *flow*, we take the maximum time spent along all paths and assign it to a global counter, associated with the process (`starTowTruckProgress-Notice_clock` in our example).

The last step before performing the verification of the model is represented by translating into BIR the GA we want to verify. In our example, we want to prove that the time elapsed between the end of activity `requestCCCallBack` and the start of activity `TowTruckProgressNotice` is less than 60 time units (minutes). To achieve this, we declare a (global) clock that keeps track of the elapsed time; this is the global variable `assistanceTimeliness_clock` introduced above. Moreover, we need a boolean flag that will be set to true right after the end of activity `requestCCCallBack`, to enable access to the global counter. The AssistanceTimeliness property can then be translated into a simple BIR assertion:

```
assert(assistanceTimeliness_clock <= 60);
```

Before emitting the actual BIR code, Bpel2Bir performs a static analysis on the flow graph of the BIR program to detect data variables (i.e., the ones associated with inbound messages activities like *receive* and *invoke*) that are not used in the computation of the process. If such variables exist, we perform an optimization that removes them and the corresponding generative code blocks from the BIR model, to reduce the size of the model itself.

The verification of the (optimized) model of the process has been performed on a Intel Core 2 Duo 2.1 GHz processor running Apple Mac OS X 10.5.3 and Bogor ver. 1.2. The verification of property AssistanceTimeliness took 175s; the model had 708002 states and 2178206 transitions.

6.2 Example 2: model checking the *Car Rental Agency* Process

The basic structure of the *Car Rental Agency* process contains a *while* loop, with a *pick* activity that waits, at each iteration, for one of the messages described in Sect. 4.2. This structure is modeled in the following BIR code snippet:

```
active thread MAIN() {
  boolean operating; operating := true;
  while operating do choose
      when <true> do //code modeling findCar
        //code modeling lookupCar
      when <true> do //code modeling carEnterX
        //code modeling markCarAvailableX
      when <true> do //code modeling carExitX
        //code modeling markCarUnavailableX
      when <true> do //code modeling stopRental
        operating := false;
      end
  end
}
```

We translated the *pick* activity into a **choose** statement, which models the occurrence of one of the events waited for by the *pick* activity. In this way, we automatically model the mutual exclusion for the occurrence of the events and the non-determinism in selecting among events that occurred simultaneously. Variable `operating` is a boolean flag that keeps the process receiving messages from external services; it is set to false when a `stopRental` message arrives, making the *while* activity and then the process terminate. We do not translate the `findCarCB` *reply* activity, since it represents an outgoing communication with an external service, which does not modify the state of the process.

To model the `carInfo` variable, which is associated with the arrival of message `findCar` and the `lookupCar` operation, we declare the `TCarInfoID` **enum** type[6] and the corresponding variable in the BIR model, as shown in the following code snippet:

```
enum TCarInfoID {c1, c2, c3}
TCarInfoID carInfo;
```

This variable is assigned a value by means of a **choose** statement, when the `findCar` message is selected by the outer **choose** statement, which models the enclosing *pick* activity. Since there are two nested **choose** statements, it is possible to optimize the generated code by flattening and producing only one **choose** statement, with a number of alternatives equal to the combination of the incoming messages and their input variables. The resulting code follows this structure:

```
choose
  when <...> do
    carInfo := TCarInfoID.c1;
    //other code modeling findCar C1
  when <...> do
    carInfo := TCarInfoID.c2;
    //other code modeling findCar C2
    //other code modeling the other alternatives
```

[6] To keep the example simple, we assume that there are only three cars available for renting, to which the three identifiers declared in the enumeration correspond.

```
  when <...> do
    //other code modeling carExit C3
end
```

The AAs defined for this process can help improve and enrich the BIR model. For example, property ParkingInOut adds some constraints on when the arrival of a `carExitX` (or a `carEnterX`) message can be "simulated" by the **choose** statement. The intuitive meaning of property ParkingInOut (a `carExitX` message cannot be received if the last message received was another `carExitX`) is translated into a guard for the **when** statement. The guard consist of a boolean variable named after the message name (e.g., `carEnter_c1`, `carExit_c1`). The boolean flag is then assigned a proper value when the corresponding event occurs: e.g., `carExit_c1` is assigned the true value when the alternative of the **choose** statement equivalent to the corresponding event is selected. The following code snippet clarifies how these boolean variables are dealt with:

```
boolean carEnter_c1;
boolean carExit_c1;
...
when <!carEnter_c1> do  //code modeling carEnterC1
  carEnter_c1 := true;
  carExit_c1  := false;
  //other code
```

As the reader may notice, variable `carEnter_c1` is true whenever `carExit_c1` is false, and vice versa. However, they are kept distinct as the translator cannot deduce this relation by simply analyzing the logical predicates of a formula. A further optimization step includes additional input from the user, when the relation between two variables/predicates is provided to the translator.

Property CISUpdate makes the translator emit the definition of similar boolean flags corresponding to the execution of activities `markAvailableX`, `markUnavailableX` and `lookupCar`. Moreover, it also defines how to generate the return value corresponding to the invocation of the `lookupCar` operation. The following code snippet exemplifies this behavior:

```
when <true> do  //code modeling findCar c3
  carInfo := TCarInfoID.c3;
  // code modeling lookupCar c3
  if markAvailable_c3 && !markUnavailable_c3 do
    queryResult_res_DiffNo := true;
  else do
    queryResult_res_DiffNo := false;
  end
```

where `queryResult_res_DiffNo` is the boolean variable corresponding to the predicate `$queryResult/res!="no"`.

Last, the RentCar GA can be translated into an **assert** expression, guarded by a logical predicate corresponding to the antecedent of the formula, as shown in the following code snippet:

```
if carEnter_c3 && !carExit_c3 do
  assert(queryResult_res_DiffNo == true);
end
```

Since the `findCarCB`, as said before, is not modeled, this assertion is placed right after the code modeling the arrival of the `findCar` message.

The verification of the model of the process in Figure 4 has been performed using the same configuration detailed in the previous section. It took 24ms to verify property RentCar; the model had 85 states and 106 transitions. By comparing the order of magnitude of the experimental data of the two examples, the reader will observe how the use of explicit time bounds, as in the *On Road Assistance* example, may increase the complexity of a model.

7 Run-time Monitoring

In SAVVY-WS, service compositions are validated at run time by monitoring AAs and GAs via Dynamo, our dynamic monitoring framework.

Monitoring rules specify the directives for the monitoring framework; each of them comprises two main parts: a set of *Monitoring Parameters* and a *Monitoring Property* expressed in ALBERT. *Monitoring Parameters* allow our approach to be flexible and adjustable with respect to the context of execution. They are meta-level information used at run time to decide whether a rule should be monitored or not. We provide three parameters:

– `priority`, which defines a simple "notion" of importance among rules, ranging over five levels of priority. When a rule is about to be evaluated, its priority is compared with a threshold value, set by the owner of the process; the rule is taken into account if its priority is less than or equal to the threshold value. By dynamically changing the threshold value we can dynamically set the intensity of probing.
– `validity`, which defines time constraints on *when* a rule should be considered. Constraints can be specified in the form of either a time window or a periodicity. The former defines time-frames within which monitoring is performed; when outside of this frame, any new monitoring activities are ignored. The latter specify how often a rule should be monitored; accepted values are durations and dates, e.g., "`3D`", meaning every 3 days, or "`10/05`" meaning every May 10th.
– `trusted providers`, which defines a list of service providers who do not need to be monitored.

Figure 5 presents the technologies we used in the implementation, as well as how the various components interact among themselves. We have chosen to adopt ActiveBPEL [23], an open-source BPEL server implementation, as our Execution Engine, and to extend it with monitoring capabilities by using aspect-oriented programming (AOP) [24]. The Data Manager represents the advice code that is weaved into the engine. When the engine initiates a new process instance, the Data Manager loads all that process' ALBERT formulae from the Formulae

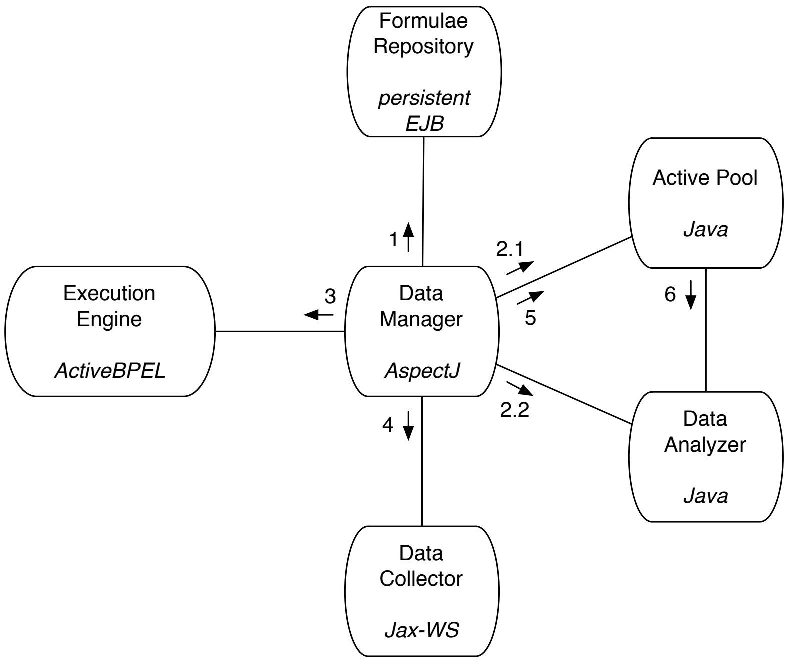


Fig. 5. Monitoring framework

Repository (step 1), and uses them to configure and activate both the Active Pool and the Data Analyzer (steps 2.1 and 2.2). The former is responsible for maintaining (bounded) historical sequences of process states, while the latter is the actual component responsible for the analysis.

The Data Manager's main task stops the process every time a new state needs to be collected for monitoring. This is facilitated by the fact that it has free access to the internals of the executing process. Once all the needed ALBERT internal variables are collected (step 3), the process is allowed to continue its execution. Notice that ALBERT formulae may also refer to external data, which do not belong to the business logic itself. In this case the data collected from the process need to be completed with data retrieved from external sources (e.g., context information), and this is achieved through special-purpose Data Collectors (step 4). Once collected, the internal and the external data make up a single process state. At this point the state is time-stamped, labeled with the location in the process from which the data were collected, and sent to the Active Pool (step 5), which stores it. Every time the Active Pool receives a new state it updates its sequences to only include the minimum amount of states required to verify all the formulae. The sequences are then used by the Data Analyzer to check the formulae (step 6).

The evaluation of ALBERT formulae that contain only references to the present state and/or to the past history (i.e., formulae that do not contain

Until, *Between*, or *Within* operators) is straightforward. On the other hand, the evaluation of formulae that contain *Until*, *Between*, or *Within* operators depends on the values the variables will assume in future states. From a theoretical point of view, this could be expressed by referring to the well-known correspondence between Linear Temporal Logic and Alternating Automata [25]. From an implementation point of view, the Data Analyzer relies on additional evaluation threads for evaluating each subformula containing one of the three aforementioned temporal operators.

Run-time monitoring inevitably introduces a performance overhead. Indeed we need to temporarily stop the executing process at each BPEL activity to collect the internal variables that constitute a new state. Meanwhile, if any external variables are needed they are collected after the process resumes its execution. Therefore, the overhead is due to two main factors: the time it takes the AOP advice to stop the process and activate internal data collection, and the collection time itself. Exhaustive tests have allowed us to quantify the former in less than 30 milliseconds. This is the time it takes the advice code to obtain the list of internal variables it needs to collect. The actual collection time, on the other hand, depends on the number of internal variables we need to collect. Once again our tests have shown that, on average, it takes 2 milliseconds per internal variable. This is due to the fact that the AOP advice code has direct access to the process' state in memory, and that ActiveBPEL provides an API method for doing just that.

More details on how the different components of this architecture work for monitoring ALBERT properties are given in [12]. Instead, in the next two sections, we will focus on the Data Analyzer, by describing how it evaluates the properties of our running examples.

7.1 Example 1: Monitoring the *On Road Assistance* Process

The first property we consider is BankResponseTime. When the `requestCard-Charge` activity is executed, the Data Manager detects, by accessing the Formulae Repository, that a property is associated with the end of the execution of the activity. Right after the activity completes, the Data Analyzer starts evaluating the consequent of the formula.

Since the root operator of the consequent is a logical OR, the Data Analyzer evaluates the left operand first, i.e., the first conjunction. The left conjunct is a reference to an external variable: the Data Analyzer asks the Data Collector to invoke the operation `getConnection` on the Web service `VCG` and then it checks the value of the `cost` part of the return message. If the value is equal to `'low'`, the Data Analyzer evaluates the other operand of the logical AND, that is the *Within* subformula.

The evaluation of such a formula cannot be completed in the current state, thus the Data Analyzer spawns a new thread to evaluate the formula in future states of the process execution. This thread checks for the truth value of its formula argument, i.e., for the occurrence of the event (notified by the Active Pool) corresponding to the start of the execution of activity `requestCCCallBack`, while keeping track of the progress of a timer, bounded by the second argument of the *Within* formula. If the formula associated with the *Within* operator becomes true before the timer reaches its upper bound, the thread returns true, otherwise it returns false.

Since the evaluation of logical AND and OR operators is short-circuited, if the evaluation of the external variable returned by the Data Collector returns false, the second operand (i.e., the *Within* formula) is not evaluated, making the Data Analyzer start evaluating the other operand of the logical OR, following a similar pattern (accessing the external variable, spawning a thread for checking the *Within* formula, checking the value returned by this thread). Similarly, if the first operand of the logical OR evaluates to true, the second operand is not evaluated.

Property AllButBankServiceResponseTime can be monitored in a similar way, but without the need for accessing external variables through the Data Collector. When one of the activities bounded to the `Act` placeholder is started, the Data Analyzer spawns a new thread, waiting for the end of the corresponding activity, within the time bound.

AvailableServicesDistance and TowTruckServiceTimeliness are two examples of properties that can be evaluated immediately. As a matter of fact, as soon as the execution of the activity listed in the antecedent of the formula finishes, the Data Analyzer retrieves the current state of the process from the Active Pool, and it evaluates the variables referenced in the formula.

Finally, the monitoring of properties TowTruckArrival and AssistanceTimeliness, follows the evaluation patterns seen above. Both formulae include a *Within* subformula, which requires an additional thread for the evaluation.

7.2 Example 2: Monitoring the *Car Rental Agency* Process

Monitoring of property ParkingInOut is triggered by the arrival of a `carExitX` message, intercepted during the execution of a *pick* activity. Right after the arrival of the message, the Data Analyzer evaluates the consequent of the formula, whose operator is an *Until*. Such a formula cannot be evaluated in the current state, thus the Data Analyzer spawns a new evaluation thread. This thread receives notifications from the Active Pool about new process states being available. When a notification arrives, the thread evaluates the second subformula of the *Until* operator, i.e., it waits for the occurrence of the event `carEnterX`. If this subformula evaluates to false, the thread evaluates (in a similar way) the first subformula of the *Until* operator. If this subformula is also false, the evaluation thread terminates by returning false. Otherwise, the thread continues to evaluate the *Until* formula in future states.

In property CISUpdate, the evaluation of the antecedent of the formula requires to spawn a new thread, since it contains an *Until* subformula. First, the Data Analyzer checks for the end of the execution of activity `markAvailableX` and then waits for the thread evaluating the *Until* subformula to terminate. This thread evaluates the formula in a similar way as described above, in the case of the ParkingInOut formula. Once the evaluating thread terminates, the overall antecedent of the formula, i.e., the logical AND, is evaluated. The consequent of the formula is thus evaluated only when the logical AND in the antecedent is true; since it contains the *Eventually* operator, its evaluation requires a new thread to be spawned. This thread checks for the occurrence of the event corresponding to the start of activity `lookupCar`; then, it spawns a new thread —since there is a second, nested, *Eventually* operator— that then waits for the end of the execution of activity `lookupCar` and checks for the value of variable `queryResult`.

Property RentCar is evaluated in a similar way, since the formula follows the same pattern of the previous one.

8 Related Work

The work presented in [19] is similar to SAVVY-WS, since it also proposes a lifelong validation framework for service compositions. The approach is based on the Event Calculus of Kowalski and Sergot [26], which is used to model and reason about the set of events generated by the execution of a business process. At design time the control flow of a process is checked for livelocks and deadlocks, while at run time it is checked if the sequence of generated events matches a certain desired behavior. The main difference with SAVVY-WS is the lack of support for data-aware properties.

Many other approaches investigated by current research tackle isolated aspects related to the main issue of engineering dependable service compositions. Design-time validation is addressed, for example, in [27], where the interaction between BPEL processes is modeled as a conversation and then verified using the SPIN model checker. In [28], design specifications (in the form of Message Sequence Charts) and implementations (in the form of BPEL processes) are translated into the Finite State Process notation and checked with the Labelled Transition System Analyzer. Besides finite state automata and process algebras, Petri Nets represent another computational model for static verification of service compositions. They are used to model both BPEL [29] and BPMN [30] processes; in both cases, the verification focuses on detecting unreachable activities and deadlocks.

Other approaches focus on run-time validation of service compositions, considering either the behavior, as in [31, 20], or the non-functional aspects [32–34], or both [35].

Design- and run-time validation activities are related to the language that is used to specify the properties that are to be validated. Besides more traditional approaches based on assertion languages like WSCoL [36], or languages for defining service-level agreements (SLAs), such as WSLA [37], WS-Agreement [38] and SLAng [39], a third trend is based on languages for defining policies, such as WS-Policy [40]. An extension of WS-Policy, called WS-Policy4MASC, is defined in [35] to support monitor and adaptation of composite web services. Even though it is not specifically bound to a validation framework, the StPowla approach [41] aims at supporting policy-driven business modeling for SOAs. StPowla is a workflow-based approach that attaches to each task of a workflow modeling a business process, a policy that expresses functional and non-functional requirements and business constraints on the execution of the task.

9 Conclusion

The paper provided a tutorial introduction to the SAVVY-WS methodology, which supports the development and operation of Web service compositions by means of a lifelong validation process. SAVVY-WS's goal is to enable the development of flexible SOAs, where the bindings to external services may change dynamically, but still control that the composition fulfills the expected functional and non-functional properties. This allows the flexibility of dynamic change to be constrained by correctness properties that are checked during design of the architecture and then monitored at run time to ensure their continuous validity.

SAVVY-WS has been implemented and validated in the case of Web services compositions implemented in the BPEL workflow language. The approach, however, has a more general scope.

It can be generalized to different composition languages and to other implementations of SOAs, which do not use Web service technologies. We have described our long-term research vision in [42]: leveraging the experience gained while working on SAVVY-WS, we want to develop SAVVY, a complete methodology for lifelong validation of dynamically evolvable software service compositions. The ultimate goal of SAVVY is to integrate specification, analysis and verification techniques, in a technology-independent manner, supported by a rich set of tools.

Acknowledgements. Part of this work has been supported by the EU project "PLASTIC" (contract number IST 026995) and by the EU project "S-Cube", funded within FP7/2007-2013 under Objective 1.2 "Services and Software Architectures, Infrastructures and Engineering".

References

1. Baresi, L., Di Nitto, E., Ghezzi, C.: Towards Open-World Software. IEEE Computer **39** (2006) 36–43
2. ICSOC: International Conference on Service-Oriented Computing series. `http://www.icsoc.org` (2003–2008)
3. SeCSE Project: Description of Work. `http://secse.eng.it/` (2004)
4. PLASTIC Project: Description of Work. `http://www.ist-plastic.org` (2005)
5. S-CUBE: S-CUBE network. `http://www.s-cube-network.eu/` (2008)
6. NESSI: Networked European Software and Services Initiative. `http://www.nessi-europe.com` (2005)
7. Papazoglou, M.: Web Services: Principles and Technology. Prentice Hall (2008)
8. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web services: Concepts, Architectures, and Applications . Springer (2003)

9. The SeCSE Team: Designing and deploying service-centric systems: The SeCSE way. In: Proceedings of Service-Oriented Computing: a look at the Inside (SOC@Inside'07), workshop co-located with ICSOC 2007. (2007)
10. ART DECO Project: Description of Work. `http://artdeco.elet.polimi.it/Artdeco` (2005)
11. DISCoRSO project: Project vision. `http://www.discorso.eng.it` (2006)
12. Baresi, L., Bianculli, D., Ghezzi, C., Guinea, S., Spoletini, P.: Validation of web service compositions. IET Softw. **1**(6) (2007) 219–232
13. Ghezzi, C., Inverardi, P., Montangero, C.: Dynamically evolvable dependable software: from oxymoron to reality. In Degano, P., De Nicola, R., Meseguer, J., eds.: Concurrency, Graphs and Models: Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday. Volume 5065 of Lecture Notes in Computer Science. Springer (2008) 330–353
14. Bianculli, D., Ghezzi, C.: SAVVY-WS at a glance: supporting verifiable dynamic service compositions. In: Proceedings of the 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS 2008), IEEE Computer Society Press (2008) to appear.
15. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1 (2003)
16. OMG: Business process modeling notation, v.1.1. `http://www.omg.org/spec/BPMN/1.1/PDF` (2008) OMG Available Specification.
17. Dwyer, M.B., Hatcliff, J., Hoosier, M., Robby: Building your own software model checker using the Bogor extensible model checking framework. In: Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005). Volume 3576 of Lecture Notes in Computer Science., Springer (2005) 148–152
18. Wirsing, M., Carizzoni, G., Gilmore, S., Gonczy, L., Koch, N., Mayer, P., Palasciano, C.: SENSORIA: Software engineering for service-oriented overlay computers. `http://www.sensoria-ist.eu/files/whitePaper.pdf` (2007)
19. Rouached, M., Perrin, O., Godart, C.: Towards formal verification of web service composition. In: Proceedings of the 4th International Conference on Business Process Management (BPM 2006). Volume 4102 of Lecture Notes in Computer Science., Springer (2006) 257–273
20. Mahbub, K., Spanoudakis, G.: A framework for requirements monitoring of service based systems. In: Proceedings of the 2nd international conference on Service-Oriented computing (ICSOC '04), ACM Press (2004) 84–93
21. Konrad, S., Cheng, B.H.C.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software engineering (ICSE '05), New York, NY, USA, ACM (2005) 372–381
22. Bianculli, D., Spoletini, P., Morzenti, A., Pradella, M., San Pietro, P.: Model checking temporal metric specification with Trio2Promela. In: Proceedings of International Symposium on Fundamentals of Software Engineering (FSEN 2007), Teheran, Iran. Volume 4767 of Lecture Notes in Computer Science., Springer (2007) 388–395
23. Active Endpoints: ActiveBPEL Engine Architecture. `http://www.activebpel.org/docs/architecture.html` (2006)
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241 of Lecture Notes in Computer Science., Springer (1997) 220–242

25. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata. Volume 1043 of Lecture Notes in Computer Science., Springer (1996) 238–266
26. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Gen. Comput. **4**(1) (1986) 67–95
27. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th International Conference on World Wide Web (WWW '04), ACM Press (2004) 621–630
28. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-based Verification of Web Service Compositions. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003) , IEEE Computer Society (2003) 152–163
29. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Sci. Comput. Program. **67**(2-3) (2007) 162–198
30. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and analysis of BPMN process models using Petri Nets. `http://eprints.qut.edu.au/archive/00007115/` (2007)
31. Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-time monitoring of instances and classes of web service compositions. In: Proceedings of the International Conference on Web Services (ICWS '06), Washington, DC, USA, IEEE Computer Society (2006) 63–71
32. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web (WWW'08), ACM (2008) 815–824
33. Raimondi, F., Skene, J., , Emmerich, W.: Efficient monitoring of web service SLAs. In: Proceedings of the 16th International Symposium on the Foundations of Software Engineering (SIGSOFT 2008 - FSE 16), ACM (2008) to appear.
34. Sahai, A., Machiraju, V., Sayal, M., Jin, L.J., Casati, F.: Automated SLA monitoring for web services. In: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '02). Volume 2506 of Lecture Notes in Computer Science., Springer (2002) 28–41
35. Erradi, A., Maheshwari, P., Tosic, V.: WS-Policy based monitoring of composite web services. In: Proceedings of the 5th European Conference on Web Services (ECOWS '07), IEEE Computer Society (2007) 99–108
36. Baresi, L., Guinea, S.: Towards dynamic monitoring of WS-BPEL processes. In: Proceedings of the 3rd International Conference on Service-Oriented Computing (ICSOC '05). Volume 3826 of Lecture Notes in Computer Science., Springer (2005) 269–282
37. Keller, A., Ludwig, H.: The WSLA framework: specifying and monitoring service level agreement for web services. Journal of Network and System Management **11**(1) (2003)
38. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement). `http://www.ogf.org/documents/GFD.107.pdf` (2007)
39. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proceedings of the 26th International Conference on Software Engineering (ICSE '04), IEEE Computer Society (2004) 179–188
40. W3C Web Services Policy Working Group: WS-Policy 1.5. `http://www.w3.org/2002/ws/policy/` (2007)

41. Gorton, S., Montangero, C., Reiff-Marganiec, S., Semini, L.: StPowla: SOA, Policies and Workflows. In: Proceedings of the 3rd International Workshop on Engineering Service-Oriented Applications: Analysis, Design, and Composition (WESOA 2007). (2007)
42. Bianculli, D., Ghezzi, C.: Towards a methodology for lifelong validation of service compositions. In: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments (SDSOA 2008), co-located with ICSE 2008, Leipzig, Germany, ACM (2008) 7–12

A ALBERT Formal Semantics

The formal semantics of the ALBERT language [12] is defined over a *timed state word*, an infinite sequence of states $s = s_1,\ s_2,\ \ldots,$ where a state s_i is a triple $(V_i,\ I_i,\ t_i)$. V_i is a set of $\langle\psi, \mathit{value}\rangle$ pairs with ψ being an expression that appears in a formula, I_i is a location of the process[7] and t_i is a time-stamp. States can therefore be considered snapshots of the process.
The arithmetic (arop) and mathematical (fun) expressions behave as expected. Function $\mathit{past}(\psi, \mathit{onEvent}(\mu),\ n)$ returns the value of ψ, calculated in the nth state in the past in which $\mathit{onEvent}(\mu)$ was true. Function $\mathit{count}(\chi,\ K)$ returns the number of states, in the last K time instances, in which χ was true, while its overloaded version $\mathit{count}(\chi, \mathit{onEvent}(\mu),\ K)$ behaves similarly but only considers states in which $\mathit{onEvent}(\mu)$ was also true. Finally, function $\mathit{elapsed}(\mathit{onEvent}(\mu))$ returns the time elapsed from the last state in which $\mathit{onEvent}(\mu)$ was true.
For all timed words s, for all $i \in \mathbb{N}$, the *satisfaction relation* $\models$ is defined as follows:

– $s,\ i \models \psi$ relop ψ' iff $\mathit{eval}(\psi, s_i)$ relop $\mathit{eval}(\psi', s_i)$
– $s,\ i \models \neg\phi$ iff $s,\ i \not\models \phi$
– $s,\ i \models \phi \wedge \xi$ iff $s,\ i \models \phi$ and $s,\ i \models \xi$
– $s,\ i \models \mathit{onEvent}(\mu)$ iff
 - if μ is a start event, $\mu \in I_{i+1}$,
 - otherwise, $\mu \in I_i$
– $s,\ i \models \mathit{Becomes}(\chi)$ iff $i > 0$ and $s,\ i \models \chi$ and $s,\ i-1 \not\models \chi$
– $s,\ i \models \mathit{Until}(\phi, \xi)$ iff $\exists j > i \mid s,\ j \models \xi$ and $\forall k$, if $i < k < j$ then $s,\ k \models \phi$
– $s,\ i \models \mathit{Between}(\phi, \xi, K)$ iff $\exists j \geq i \mid s,\ j \models \phi$ and $\forall l$ if $i \leq l < j$ then $s,\ l \not\models \phi$ and $\exists h\ \mid t_h \leq t_j + K,\ t_{h+1} > t_j + K$ and $s,\ h \models \xi$
– $s,\ i \models \mathit{Within}(\phi, K)$ iff $\exists j \geq i \mid t_j - t_i \leq K$ and $s,\ j \models \phi$

where function *eval* takes an ALBERT expression ψ and a state in the timed state word s_i and returns the value of ψ in s_i.

[7] A location is defined as a set of labels of BPEL activities; in the case of a *flow* activity, it contains, for each parallel branch of the *flow*, the last activity executed in that branch.