Validation of Web Service Compositions [*]

Luciano Baresi[†] Domenico Bianculli[‡] Carlo Ghezzi[†] Sam Guinea[†] Paola Spoletini[†]

[†] Dipartimento di Elettronica e Informazione
Politecnico di Milano
via Ponzio 34/5 - I-20133, Milano, Italy
{baresi, ghezzi, guinea, spoleti}@elet.polimi.it

[‡] Faculty of Informatics
University of Lugano
via G. Buffi 13 - CH-6900, Lugano, Switzerland
domenico.bianculli@lu.unisi.ch

March 17, 2008

Abstract

Web services support software architectures that can evolve dynamically. In particular, here we focus on architectures where services are composed (orchestrated) through a workflow described in the BPEL language. We assume that the resulting composite service refers to external services through assertions that specify their expected functional and non-functional properties. Based on these assertions, the composite service may be verified at design time by checking that it ensures certain relevant properties. Because of the dynamic nature of Web services and the multiple stakeholders involved in their provision, however, the external services may evolve dynamically, and even unexpectedly. They may become inconsistent with respect to the assertions against which the workflow was verified during development. As a consequence, validation of the composition must extend to run time. We introduce an assertion language, called ALBERT, which can be used to specify both functional and non-functional properties. We also describe an environment which supports design-time verification of ALBERT assertions for BPEL workflows via model checking. At run time, the assertions can be turned into checks that a software monitor performs on the composite system to verify that it continues to guarantee its required properties. A *TeleAssistance* application is provided as a running example to illustrate our validation framework.

1 Introduction

Service oriented architectures (SoAs) recently emerged as a useful architectural paradigm in new and innovative computing domains, like ambient intelligence, context-aware applications, and pervasive computing [1]. Many current technologies can be associated with SoAs, such as Jini [2], OSGi [3], etc. However, due to substantial investments by important industrial players, such as

[*]This paper is a postprint of a paper submitted to and accepted for publication in IET Software and is subject to Institution of Engineering and Technology Copyright. The copy of record is available at **IET Digital Library**.

BEA, IBM, Microsoft and Oracle, important open-source communities, such as Apache, and a very active research community, it is common to identify SoAs with the Web-based implementations that go under the term “Web services”.

The research we describe here focuses on Web services. In particular, it deals with service compositions built using BPEL (Business Process Execution Language) [4]. BPEL *workflows* (also called *processes*) may define new, composite services by coordinating (“orchestrating”) external services, which are typically not under their jurisdiction. This leads to a distributed ownership of the composite system, which is ultimately responsible for its overall functionalities and quality of service (QoS). External partner services, which affect both functionalities and QoS of composite services, may in fact evolve independently, and even unexpectedly, even after the system is deployed.

When a composite service is designed, certain assumptions must be made on the external services which will be orchestrated. Not only must the syntax of their interface be considered, but also their semantics. In particular, the designer must decide which properties must be fulfilled by the external services and, in turn, based on these assumptions, which properties the composite service will guarantee to its own users.

In the dynamic world of service-oriented architectures, however, what is guaranteed at development time, unfortunately, may not be true at run time. The actual services to which the workflow is bound may change dynamically[1], possibly in an unexpected way that may cause the implemented composition to diverge from the assumptions made at design time. Traditional approaches, which restrict validation to being a design time activity, are no longer valid in this dynamic setting. Besides performing design-time validation, it is also necessary to perform continuous run-time validation to ensure that the required properties are maintained by the operational system. However, it is virtually impossible to predict all the evolutions and changes that might occur in the services we use, and the same is true for the environment. This leads us to consider monitoring as a defensive means. Since it is currently unrealistic to believe that external services will provide a formal and machine-readable specification of their functionality and quality of service, all we can rely on are our process-side expectations of their functionality and quality of service. Therefore, monitoring allow us to take notice of infringements of such expectations.

In this article we propose a framework for validating BPEL processes that covers both design-time and run-time validation. Properties are expressed in ALBERT (Assertion Language for BPEL pRocess inTeractions). ALBERT assertions can be used for two purposes. First, they can formally specify the properties that partner services are required to fulfill. Such properties formalize the assumptions on the external services made at development time by the software developer while designing the workflow. These are called *assumed assertions (AAs)*. Second, ALBERT assertions can be used to state properties that the workflow should satisfy, assuming that external services operate as specified. Such assertions, which characterize the behavior the composite service should guarantee, are called *guaranteed assertions (GAs)*. The use of assume-guarantee reasoning is a known technique in verification. It is used to support “divide and conquer” and compositional reasoning [6]. *AA*s and *GA*s can state both functional and non-functional properties. Because of this use of assertions, ALBERT promotes “design by contract”, as advocated by [7].

Our validation environment supports the software designer at design time by verifying that *GAs* hold for a given workflow, assuming that *AAs* hold. This verification is achieved via model checking. We also provide a run-time monitoring facility that checks whether the external services satisfy the *AAs* and whether *GAs* also hold.

[1]In BPEL, only the implementation behind partner services can change, but there are many proposals [5] to complement BPEL with dynamic binding capabilities.

When designing our framework, we adhered to the following design principles:

- **Use of standard technology.** We decided to use standard technology (e.g. BPEL, XML, XPath, etc.) to favor adoption of the proposed approach.
- **Separation of concerns.** The process designer should concentrate separately on the business logic implemented by the workflow and on the validation properties expressed in ALBERT. The two are kept in two separate documents. This is also true for the enactment of the workflow. The adoption of an aspect-oriented approach in the implementation allows the monitoring logic to be kept separate from the business logic.
- **Defensive design.** Because external services can evolve dynamically, the assumptions (*AAs*) made on the environment when the abstract workflow is statically validated may be violated at run time. ALBERT can be used to declaratively state the properties that must hold and then to support the monitoring of these properties at run time.

The main contribution of this article is the description of a complete and coherent framework for validating BPEL process compositions. It consolidates our previous work which investigated different aspects of service-oriented architectures, and in particular static and dynamic analysis of Web services. The ALBERT language and the overall validation framework represent the novel contributions of the article. To the best of our knowledge, although many existing research efforts deal with different aspects of our approach, none provides a complete and coherent coverage of validation of Web service compositions and the specification of both functional and non-functional properties.

The article is organized as follows. Section 2 gives an overview of BPEL, to make the reader familiar with the language constructs which are then used throughout the paper. Section 3 introduces a running example, which will be used to illustrate the concepts and tools we provide. Section 4 introduces the specification language ALBERT. Section 5 describes the overall validation methodology and how it fits into a complete development process. Section 6 describes our model checking approach, while Section 7 describes how we achieve run-time monitoring. Section 8 surveys the state of the art. Finally, Section 9 draws some conclusions and outlines future work directions.

2 Overview of BPEL

BPEL [4], Business Process Execution Language (for Web Services) is a high-level XML-based language for the definition and execution of business processes by means of Web service-based workflows. The definition of a process contains a set of global variables and the workflow logic expressed as a composition of *activities* (Figure 1 shows the graphical notation we use in the rest of the paper); 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*), for signaling faults (*throw*), for pausing (*wait*), and for stopping the execution of the process (*terminate*). The *sequence, while,* and *switch* constructs provide standard control structures to order activities, define loops and branches.The *pick* construct is peculiar to the domain of concurrent and distributed systems, and waits either for the first out of several incoming messages to occur or for a time-out alarm to go off, to execute the activities associated with such event.

The *flow* construct supports the concurrent execution of activities. Synchronization among the activities of a *flow* may be expressed using the *link* construct; a *link* can have a guard, which is called

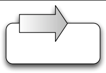
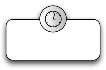
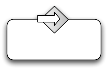
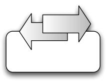
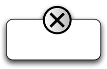
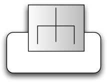
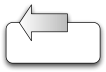
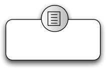
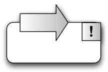
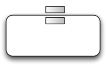
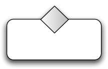
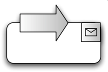
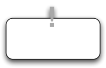
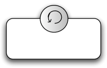
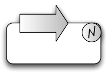
Activity	Shape	Activity	Shape	Activity	Shape
receive		*wait*		*pick*	
invoke		*terminate*		*flow*	
reply		*sequence*		*fault handler*	
assign		*switch*		*event handler*	
throw		*while*		*compensation handler*	

Figure 1: Graphical notation for BPEL.

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* (DPE), 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*.

3 Running example

TA (*TeleAssistance*) is a small company in the business of remote assistance to medical patients. Its server runs the BPEL process shown in Figure 2 to assist its clients. The process starts as soon as a *Patient* (PA) enables the home device supplied by *TA*, which sends a message to the process' *receive* activity `startAssistance`. Then, it enters an infinite loop: every iteration is a *pick* activity that suspends the execution and waits for one of the following three messages:

- `vitalParamsMsg`. The home device (e.g. a glucometer) sends the patient's vital parameters (which are saved in the variable `vitalParams`, which has a field named `glucose`, containing the measured glucose level). This message enables the corresponding execution path, in which the vital parameters are sent to the service *Medical Laboratory* (*LAB*), by invoking operation `analyzeData`. The *LAB* is in charge of analyzing the data and replies by sending a result value stored in a variable `analysisResult`. This variable contains a field `suggestion`

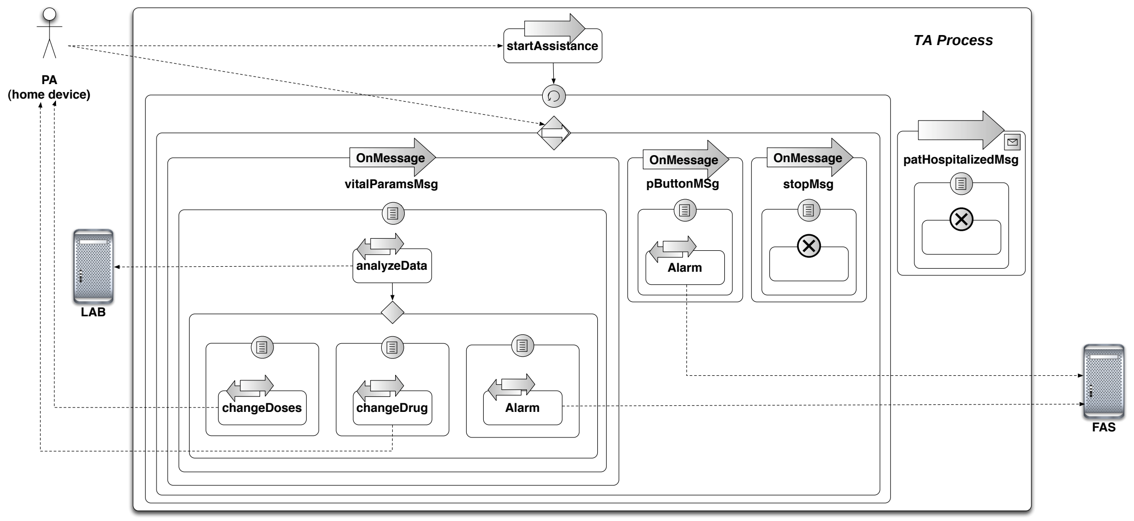


Figure 2: The TeleAssistance service.

whose value can be: `‘changeDrug’`, `‘changeDoses’` or `‘sendAlarm’`. In the last case, the *TA* process invokes service *First-Aid Squad* (*FAS*). This service coordinates a group of doctors, nurses, and paramedics who assist patients day-by-day and visit them at home in case of emergency. To alert the squad, the *TA* process invokes the operation `alarm` on the *FAS*, by passing the id of the patient and the severity level of the alarm (`‘mild’` in this case).

- `pButtonMsg`. The patient can press a panic button, causing an alarm message to be sent to the *TA* process. If the patient feels sick, s/he can issue a request for immediate assistance. The *TA* process alerts the *FAS* with an alarm, whose severity level is `‘high’`.
- `stopMsg`. The patient may decide to cancel the *TA* service.

We assume that the *TA* process uses a variable `alarmNotif` to send an alarm notification to the *FAS*. This variable contains a field `level`, which can be set to `‘mild’` or `‘high’`, and a field `pID` which represents the patient identification code.

If the patient needs special-purpose assistance, the *FAS* transfers the patient to the closest hospital. Upon arrival, the *FAS* notifies *TA* that the patient has been hospitalized by sending a message (`patHospitalizedMsg`), which is received through an appropriate *event handler*, saving the parameter in variable `patHospitalized`. The current instance of the *TA* process then terminates.

4 ALBERT

ALBERT is an assertion language for BPEL processes. It is reminiscent of assertion languages that were designed for specific programming languages such as ANNA [8], an annotation language for Ada, and JML [9], the Java Modeling Language. ALBERT supports the definition of assumed (*AA*) and guaranteed (*GA*) assertions, that state both functional and non-functional properties of BPEL processes.

For example, in our *TA* process we may define a number of assertions (*AAs*) that state the assumptions made on the partner services, upon which we base the design of the process. One of such properties (referred to as VitalParams) specifies that "The glucose value sent by the patient's remote device to the process is between 40 mg/dL and 300 mg/dL". Another *AA* (referred to as FASConfirmHospitalization) specifies that "If the *FAS* is invoked three times over a week, with a `'high'` severity level alarm for a certain patient, it must notify the *TA*, within one day, that the patient has been hospitalized."

The language can also specify assertions (*GAs*) that must be satisfied by the workflow, assuming that external services behave as specified by *AAs*. One such property (referred to as FASInvokeMildAlarm) specifies that "After receiving a message from the *LAB*, indicating that an alarm must be issued to the *FAS*, the *TA* process must invoke the *FAS* service within 4 hours, passing a `'mild'` alarm notification".

Another example of a *GA* (referred to as MDCheckUp) specifies that: "If a certain patient sends the `pButtonMsg` three times during a time span of a week, the *FAS* must hospitalize the patient within one day". Intuitively, the truth of this assertion is assured by the *AA* FASConfirmHospitalization, combined with the structure of the workflow, shown in Figure 2.

4.1 Variables

Variables used in ALBERT assertions can be of two kinds: internal and external. *Internal variables* consist of elementary data (i.e., a number, a string, or a boolean) that are extracted from BPEL variables. For example, we can refer to a patient's glucose level as: `($vitalParams/glucose)`. In this expression, `$vitalParams` indicates the BPEL variable from which we are extracting an elementary value, while `/glucose` is the XPath expression used to extract the desired value.

ALBERT also provides means to predicate on variables whose values originate outside the process. This is useful when the correctness of a property can only be established by referring to contextual data provided by external data sources, such as the time and/or place of execution. These are called *external variables*. For example, an external variable can be used to define the following *AA* property: "The new drug sent by the *LAB* service through variable `analysisResult` must be amongst those in the list of drugs approved by the FDA (Food and Drug Administration)". To check if the drug is valid, one needs to refer to an external variable that is provided by querying the *FDA online registry*, an external service which is not part of the workflow. In our example the external variable could be defined as: `FDA::inList(ins)/result`, where `FDA::inList` is the invoked remote method, `ins` is the input message for the method (in our example it contains the name of the drug), and `/result` is the XPath expression used to extract the desired datum.

4.2 Constructs

In this section we provide an introduction to the main constructs of the ALBERT language. The language defines formulae which specify *invariant* assertions for the workflow. Formulae are defined by the following syntax:

$$\begin{aligned}
&\phi ::= \psi \text{ relop } \psi \;\mid\; \neg\phi \;\mid\; \phi \wedge \phi \;\mid\; (\text{ op id } \texttt{in} \text{ var } ;\ \phi\ ) \;\mid\; \mathit{onEvent}(\mu) \;\mid\\
&\mathit{Becomes}(\phi) \;\mid\; \mathit{Until}(\phi,\phi) \;\mid\; \mathit{Between}(\phi,\phi,K) \;\mid\; \mathit{Within}(\phi,K)\\
&\psi ::= \text{var} \;\mid\; \psi \text{ arop } \psi \;\mid\; \text{const} \;\mid\; \mathit{past}(\psi,\mathit{onEvent}(\mu),\ n) \;\mid\; \mathit{count}(\phi,\ K) \;\mid\\
&\mathit{count}(\phi,\mathit{onEvent}(\mu),K) \;\mid\; \text{fun}(\psi,K) \;\mid\; \text{fun}(\psi,\ \mathit{onEvent}(\mu),K) \;\mid\; \mathit{elapsed}(\mathit{onEvent}(\mu))\\
&\text{relop} ::= < \;\mid\; \leq \;\mid\; = \;\mid\; \geq \;\mid\; >\\
&\text{op} ::= \texttt{forall} \;\mid\; \texttt{exists}\\
&\text{arop} ::= + \;\mid\; - \;\mid\; \times \;\mid\; \div\\
&\text{fun} ::= \mathit{sum} \;\mid\; \mathit{avg} \;\mid\; \mathit{min} \;\mid\; \mathit{max} \;\mid\; \ldots
\end{aligned}$$

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 the start or the end of an *invoke* or *receive* activity, the reception 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 defines the core of the language. To improve its expressiveness, other constructs are also provided, including the obvious logical connectives, which can be trivially derived from $\neg$ and $\wedge$, and the temporal operators *Always* and *Eventually*, which can be derived from *Until*. Although, in principal, universal and existential quantifiers should not be part of the core of the language, because they predicate over finite sets of data values, we introduce for notational convenience and use them extensively in the rest of the paper.

Here we present the semantics of the language informally, assuming that the workflow process does not contain a *flow* activity. This is not a limitation of the language; it allows us to simplify our presentation. The formal semantics for the complete core language is reported in Appendix A.

The informal meaning of ALBERT formulae can be explained by referring to sequences of (time-stamped) states of the BPEL process. A *state* is a triple (V, i, t), where V is a set of $\langle$*variable*, *value*$\rangle$ pairs, i is a label of a BPEL instruction, and t indicates a time instant in the domain of positive real numbers. V is the set of $\langle$variable, value$\rangle$ pairs that hold *after* executing the BPEL activity i, and t is the instant at which the execution of the statement is completed. Two states $s_j = (V_j, i_j, t_j)$ and $s_{j+1} = (V_{j+1}, i_{j+1}, t_{j+1})$ are adjacent in the sequence if i_{j+1} is the activity that follows i_j in the control flow and the execution of i_{j+1} on the variables in V_j terminates at time t_{j+1}, yielding V_{j+1}.

Boolean, relational, and arithmetic operators have the conventional meaning; the same is true for quantifiers. Because ALBERT assertions are implicitly assumed to be invariant for the BPEL process, they express properties that must hold in all states. To express the fact that they must hold when the execution reaches a given point of the workflow, we need to use the predicate *onEvent*, which is true when the corresponding event occurs. For example, property VitalParams can be expressed as:

$$\begin{array}{l} \textit{onEvent}(\texttt{vitalParamsMsg}) \rightarrow \\ (\texttt{\$vitalParams/glucose} \geq 40 \wedge \texttt{\$vitalParams/glucose} \leq 300) \end{array}$$

More precisely, in the case of *assign*, *pick*, *event handler*, and the end of *invoke* or *receive* activities, it is true in a state whose label identifies the corresponding activity. In the case of the start of an *invoke* or *receive* activity, it is true in a state if the label of the next state in the sequence identifies the corresponding activity. In the case of a *while* or a *switch* activity, it is true in the state where the condition is evaluated.

Temporal predicate *Becomes* is evaluated on two adjacent elements of the sequence of states. The formula is true when its argument is true in the current state, and false in the previous. The temporal predicate *Until*(ϕ, ξ) is true in a given state if ξ is true in the current state, or eventually in a future state, and ϕ holds in all the states from the current (included) until that state (excluded). The temporal predicate *Between*(ϕ, ξ, K) is true in a given state $s_j = (V_j, i_j, t_j)$ if ϕ is true, for the first time, in a state $s_k = (V_k, i_k, t_k)$ such that $t_k \geq t_j$, and ξ is true in a further subsequent state $s_w = (V_w, i_w, t_w)$ such that $t_w - t_k \leq K$, and for the successor state $s_{w+1} = (V_{w+1}, i_{w+1}, t_{w+1})$, $t_{w+1} - t_j > K$. The temporal predicate *Within*(ϕ, K) is evaluated on a finite sequence of states, built from (and including) the current state $s_j = (V_j, i_j, t_j)$ until we reach a subsequent state $s_k = (V_k, i_k, t_k)$ for which $t_k - t_j \leq K$, and for the successor state $s_{k+1} = (V_{k+1}, i_{k+1}, t_{k+1})$, $t_{k+1} - t_j > K$. The predicate is true if ϕ is true at least in one of these states.

Function $past(\psi, onEvent(\mu), n)$ is computed on a historical sequence of states, built backwards from (and excluding) the current state. The sequence must contain n states in which $onEvent(\mu)$ is true. The function returns the value of ψ in the state of the sequence with the smallest t. The function is undefined if such a sequence cannot be found. Function $count(\phi, K)$ function is also computed on a finite historical sequence of states, built backwards from (and including) the current state $s_j = (V_j, i_j, t_j)$ until we reach a state $s_k = (V_k, i_k, t_k)$ for which $t_j - t_k \leq K$, and for the predecessor state $s_{k-1} = (V_{k-1}, i_{k-1}, t_{k-1})$, $t_j - t_{k-1} > K$. The function returns the number of elements in this sequence in which ϕ holds. The overloaded version of the function ($count(\phi, onEvent(\mu), K)$ only considers states in which $onEvent(\mu)$ is true[2].

The placeholder fun stands for any function (e.g., average, sum, minimum, maximum ...) that can be applied to sets of numerical values. As for *count*, there are two overloaded cases. $\mathsf{fun}(\psi, K)$ is computed on a finite historical sequence of states, built backwards from (and including) the current state $s_j = (V_j, i_j, t_j)$ until we reach a state $s_k = (V_k, i_k, t_k)$ for which $t_j - t_k \leq K$, and for the predecessor state $s_{k-1} = (V_{k-1}, i_{k-1}, t_{k-1})$, $t_j - t_{k-1} > K$. The function returns the value resulting from the application of function fun to all values of the expression ψ in all states of the sequence. The overloaded version, which only considers states in which $onEvent(\mu)$ is true, as before, does not add expressive power to the language. Function $elapsed(onEvent(\mu))$ is computed on a finite historical sequence of states, built backwards from (and including) the current state $s_j = (V_j, i_j, t_j)$ until we reach the first state $s_k = (V_k, i_k, t_k)$ in which $onEvent(\mu)$ is true. The function returns $t_j - t_k$. The function is undefined if such a sequence cannot be found.

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. These are common templates for *AAs*:

- $onEvent(\mu) \rightarrow \phi$
- $past(\psi', onEvent(\mu), n) = \psi \rightarrow \phi$
- $Becomes(count(\phi', onEvent(\mu), K) = \psi) \rightarrow \phi$
- $Becomes(\mathsf{fun}(\psi', onEvent(\mu), K) = \psi) \rightarrow \phi$

where ϕ and ϕ' are ALBERT formulae, μ identifies the start or the end of an *invoke* or *receive* activity, or the reception of a message by a *pick* or an *event handler*. ψ and ψ' are ALBERT expressions, n is a natural number, and K is a positive real number.

These templates limit the states on which ϕ needs to be true to satisfy the formula. More precisely, the first template checks the truth value of ϕ only in the states in which $onEvent(\mu)$ is true; i.e., in the states preceding or following an interaction with an external service. In the second template, the fact that a property is checked depends on past interactions with the outside world. More precisely, we only check ϕ if, in association with a past interaction with an external service ($onEvent(\mu)$), ψ' was equal to ψ. In the third template, the property is checked if past interactions with the outside world have led to a certain number of specific events. More precisely, we are interested in checking ϕ in a state s, if, in s, it becomes true that, in the last K time instants, ϕ' is true ψ times. Notice that, when counting, we only consider states that are related to interactions with external services ($onEvent(\mu)$), meaning that we count the number of times in which, in relation to these interactions, ϕ' is true. In the last template, the property is checked if past interactions

[2]The overloaded version does not add expressive power. Indeed, $count(\phi, onEvent(\mu), K)$ is equivalent to $count(\phi \wedge onEvent(\mu), K)$. The overloaded version is kept to simplify the specification of formulae.

with the outside world have led to a certain value of an aggregate function. In more detail, we are interested in checking ϕ in a state s, if, in s, it becomes true that fun, calculated over the values of ψ', obtained from states associated with interactions with external services ($onEvent(\mu)$) over the last K time instants, is ψ. In all four cases, the decision to check ϕ depends on interactions with external services.

4.3 Examples

Referring to the example of Figure 2, a non-functional assertion (hereafter referred to as LabServiceTime) could be "After sending the patient's data to the lab, the lab should reply within 1 hour" . This is an *AA* on the response time of the external *LAB* service that can be expressed as:

$$onEvent(\texttt{start\_analyzeData}) \rightarrow Within(onEvent(\texttt{end\_analyzeData}), 60)$$

Another non-functional assertion (hereafter referred to as AverageLabServiceTime) could be "The average response time of all the invocations of operation `analyzeData` on service *LAB* completed in the past ten hours should be less than 45 minutes". This *AA* can be expressed as:

$$avg(elapsed(onEvent(\texttt{start\_analyzeData})), onEvent(\texttt{end\_analyzeData}), 600) \leq 45$$

Property FASInvokeMildAlarm can be expressed as:

$$onEvent(\texttt{end\_analyzeData}) \wedge \texttt{\$analysisResult/suggestion} = \texttt{`sendAlarm'}$$
$$\rightarrow \quad Within(\texttt{\$alarmNotif/level} \quad = \quad \texttt{`mild'} \quad \wedge \quad onEvent(\texttt{start\_alarm})), 240)$$

In this case we are defining a *GA* which states that upon ending the execution of the activity `analyzeData`, if field `suggestion` of the output variable is `sendAlarm`, then the process must guarantee that an `alarmNotif` is sent within 4 hours, with the severity level equal to `'mild'`.
Property MDCheckUp can be expressed as:

$$\forall x(Becomes(count(\texttt{\$alarmNotif/level=`high'} \wedge x = \texttt{\$alarmNotif/pID},$$
$$onEvent(\texttt{pButtonMsg}), 10080) = 3)$$
$$\rightarrow \quad Within(onEvent(\texttt{patHospitalizedMsg}) \quad \wedge \quad \texttt{\$patHospitalized/pId=x}, 1440))$$

In the example, we count the number of times the `pButtonMsg` is received within a week; if it is received three times, the *TA* should receive the confirmation of hospitalization within 24 hours, by processing a `patHospitalizedMsg` event.

5 ALBERT-aware Development Process

Figure 3 describes how our validation framework fits into a complete development process. The first step consists in designing the service composition, represented as a BPEL process; this task is usually accomplished by a BPEL designer, i.e. an expert in business processes modeling.

The BPEL document produced by the design phase is passed to the Verification and Validation (V&V) engineer, who annotates it with ALBERT assertions. These assertions may be:

- *AAs*, which represent the assumptions made on the external services, i.e. process-side expectations of the quality of service and functionality the external services will provide;

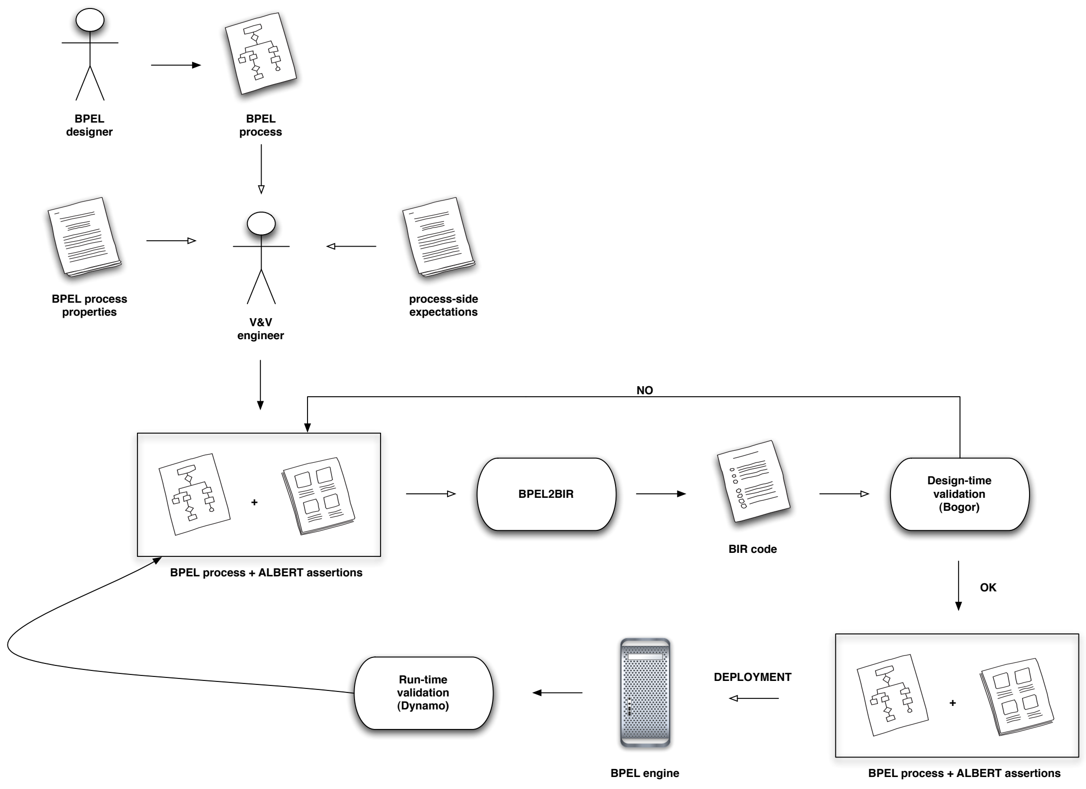

BPEL designer
BPEL process
BPEL process properties
V&V engineer
process-side expectations
NO
+
BPEL2BIR
BIR code
Design-time validation (Bogor)
BPEL process + ALBERT assertions
OK
Run-time validation (Dynamo)
DEPLOYMENT
BPEL engine
BPEL process + ALBERT assertions

Figure 3: ALBERT-aware development process.

- *GAs*, which state the properties that the workflow should satisfy, if the *AAs* hold.

The BPEL process, annotated with ALBERT assertions, is then provided to BPEL2BIR, which outputs a model suitable for verification through a model checker (Bogor, in this case). The design and design-time validation phases are continuously repeated until the *GAs* are satisfied.

Then, the BPEL process is deployed to an execution engine running Dynamo, our monitoring framework. Dynamo checks during run time if the assertions (both the *GAs* and the *AAs*) are satisfied.

If the assertions are violated, the process can be verified again, by relaxing the assumptions, or be redesigned and re-deployed.

The next two sections present the technical details of the two validation techniques we propose, including a qualitative evaluation of performance issues.

Both of the analysis techniques involve relevant complexity issues, that could affect the overall performance of the proposed methodology. However the technical details we present in the following sections allow us to deal with them efficiently, as we shall see from a qualitative stand point.

6 Design-time validation

In this section we describe our approach to design-time validation based on model checking. Model checking [6] is a completely automatic technique in which the state space of the model representing the system under verification is exhaustively analyzed. As a consequence, the model has to be finite in the number of states and can only contain variables that have finite sets of values. Since the state space must be completely explored, model checking suffers the well-known *state explosion problem*, which can be mitigated by the introduction of carefully crafted abstractions.

Several abstraction mechanisms have been proposed for programs written in common programming languages such as C (see, for example [10, 11]). Hereafter we discuss how ALBERT and the assume-guarantee design methodology it enforces can support the definition of abstractions that can help in the verification of Web service compositions.

According to our approach, we analyze workflow-based service compositions by abstracting the external environment (i.e., external partner services) through interfaces to external services, viewed as black boxes that fulfill certain desired properties, formalized through *AAs*. Often, these abstract interfaces are all one can know about the environment. Sometimes, however, some of the external partner services are shared within a restricted cooperating community. In such a case, it may be possible to inspect the black box. The design-and-conquer methodology supported by ALBERT allows the implementation of these partner services to be also analyzed, to check that what their users require as *AAs* are guaranteed by the implementations as *GAs*. Indeed, this is the essence of an assume-guarantee approach to design-time verification. ALBERT allows the approach to deal not only with functional requirements, but also with QoS agreements that bind service requesters and service providers, such as response time constraints.

To support analysis, we developed BPEL2BIR (BPEL and ALBERT to BIR), a model checking framework based on Bogor [12]. Our validation framework and the ALBERT language are theoretically independent of the selected model checking and monitoring technologies. In particular, Bogor was chosen because most BPEL constructs can be easily mapped onto BIR (Bogor's input language). Furthermore, Bogor's modular architecture supports the introduction of different model checking algorithms and customizations for particular domains. This will be exploited in the future evolution of this work, as outlined in Section 9.

To develop BPEL2BIR, we had to solve three main problems. First, we build a model from the BPEL workflow by modeling the interactions with the external world through the generation of random values for the relevant variables. Second, we exploit the abstractions that can be derived from *AAs*, which allow for a better representation of the interactions with the external world. Then, we translate *GAs* into BIR properties that must be verified. The model-checking problem is described in the following by presenting how we encode BPEL activities, *AAs*, and *GAs* into BIR constructs.

The approach to model check Web service compositions presented in this article differs from other proposals that appeared in the literature because:

- It is supported by an assume-guarantee verification methodology that fosters design-and-conquer.
- It covers the whole set of BPEL constructs, including those dealing with time.
- ALBERT allows one to describe a rich set of functional and non-functional properties.

Preliminary experimental results on the use of BPEL2BIR have shown that Bogor can provide a better support for model checking than other similar tools [13].

6.1 Bogor

Bogor [12] is a model checking framework developed at Kansas State University. The input language for Bogor, called BIR (Bogor Intermediate Representation) provides constructs found in modern programming languages, such as dynamic threads and object creation, exception handling, virtual functions, recursive functions, and garbage collection. A low-level version of the intermediate representation, named low-level BIR, is a language for the description of transition systems based on guarded commands with explicit locations, explicit guards, sequences of statements comprising transition actions, and explicit transitions.

The BIR data model contains primitive types (`boolean`, `int`, `enum`) and non-primitive types (`null`, `record`, `array`, `lock`). For record types, sub-typing declarations and virtual methods are also supported. The language is statically strongly typed. The memory model forbids pointer arithmetic and achieves object reclamation through garbage collection. To overcome the state explosion problem, Bogor implements some well known optimization and reduction strategies, such as data and thread symmetry [14], collapse compression [15], and partial order reduction [16].

Bogor is *extensible*, both in terms of input language and model checking algorithms. It has an open, modular architecture that allows the development of extensions via new algorithms and optimizations, to improve core tasks, such as state encoding and state exploration. Several extensions have been implemented; among these, the ones related to partial-order reduction, state-encoding, search strategies, support for different property languages (regular expressions, LTL and CTL, JML), and support for several domains (multi-threading and Swing, event-based Java programs, CORBA-based avionics systems). For example, in our group we designed Bogor extensions to validate event-based service architectures based on the Publish/Subscribe paradigm [17].

6.2 Modeling BPEL in Bogor

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 using an intuitive mapping between WSDL message/XML Schema types and BIR primitive/record types. Basic activities are trivially mapped onto their equivalent in BIR. Instead, a *receive* (resp., *invoke*) activity is translated as an assignment to its input (resp., output) variable, since the behavior of external services is not modeled. If no assumptions are made on the external partner services, the assignment is performed with a non-deterministically generated value, ranging over all its domain. Otherwise, if *AAs* are provided to constraint the expected behavior of external services, they generate ad-hoc abstractions as discussed next. Activities contained within a *flow* are translated into threads, preserving both transition and join conditions.

For each scope, *fault handlers* are translated as `try/catch` statements, with `catch(var)` clauses matching exception variables corresponding to faults. *Event handlers* are modeled using a dedicated thread, which non-deterministically consumes the events produced by a helper function.

A *pick* activity waits for the occurrence of one out of several messages delivered by external services. In this case, it is translated by invoking a function that models the occurrence of one of the messages being awaited; the message is then treated as if it had been received through a *receive*. Optionally, a *pick* activity can specify a timeout and a *wait* activity can specify a suspension. They both indicate how long (*for*) or *until* the process needs to wait.

To deal with time constraints in BIR, we need to include in the model an execution time for all BPEL activities represented in BIR and introduce a time counter for each sequential path of activities generated during execution of the BPEL workflow.

For each activity, we insert a code block in BIR that randomly generates the duration of the activity within a certain interval. If there is an *AA* that constrains the duration of the activity, we use it to generate the BIR code as discussed in the next section. For a *flow* activity, the time consumed by the flow is the maximum time spent along all paths. Notice that if two activities in different paths are *linked*, we synchronize the time on this link by assigning to its time counter the maximum value of the counters on the incoming links.

6.3 Assumed Assertions

In the previous section, we generated random values to model both the values generated through interactions with an external service and their duration. Now we show how *AAs* can provide a better abstraction of external environment by reducing the range of the generated values.

As an example, in the *TeleAssistance* service, the *AA* associated with a receipt of message `VitalParamsMsg` (property VitalParams) states that the glucose value sent by the remote device must be between 40 and 300 mg/dL. This constraint is used to reduce the size of the domain from which the values are generated by the model checker for variable `$vitalParams/glucose`.

In some cases the formula representing an *AA* may refer to the value of an expression in the past, such as $past(\psi, onEvent(\mu), n)$. In such a case, it is necessary to log the historical values of the variables appearing in ψ when a location, in which $onEvent(\mu)$ is true, is reached during the execution. Logged data must be retrieved to evaluate *past*. Quantified ALBERT formulae are also used in the optimized generation of input values. The universal quantification is translated into a generation, of values of all the quantified variables, performed according to the quantified formula, while the existential quantifier is translated into a non-deterministic choice of the variable, whose value is then generated according to the quantified formula.

AAs can express constraints on the duration of activities executed by the invocation of external services. As an example, consider assertion LabServiceTime. In such a case, the time bound expressed by the *Within* operator can be assumed as an upper-bound for the duration of the activity whose `start` and `end` events are referred in the formula.

6.4 Guaranteed Assertions

GAs specify invariant properties of the workflow. Thus each assertion must be evaluated after the execution of each activity of the BPEL process. The model checker executes the check by instantiating an evaluator for each assertion after executing the BIR code block corresponding to each activity in the workflow.

The evaluation of each assertion can be completed in a single step if it only refers to the present or past states of the computation. If, instead, it contains future temporal operators (such as *Within*), the instance of the evaluator carries on in all subsequent states until it terminates by producing the truth value of the assertion.

ALBERT assertions can refer to present and past values of variables and past sequences of events. To support their evaluation, the BIR `system` that models the BPEL process, should also include bookkeeping actions that collect the historical values and other auxiliary variables needed for verification.

Let us describe in more details how an evaluator for an ALBERT assertion works. The evaluation of arithmetic and relational operators is straightforward. Existential (universal) quantifiers over formulae are interpreted as disjunctions (respectively, conjunctions) of formulae. The evaluation of function $past(\psi, onEvent(\mu), n)$ requires accessing the historical values of expression ψ. The values of the variables referred by ψ are stored in an array of n elements, which is kept updated by the bookkeeping actions we mentioned above.

To evaluate predicate $onEvent(\mu)$, we rely on a boolean auxiliary variable, which is set to true by the BIR `system` exactly when μ happens, and to false immediately afterwards. The evaluation of the predicate return the value this variable. The evaluation of predicate $Becomes(\phi)$ returns true if ϕ is true in the current state and false in the previous. This is implemented by using an auxiliary variable that contains the previous value of ϕ. Function $elapsed(onEvent(\mu))$ is computed by using a counter variable managed as auxiliary variable by the BIR `system`. This variable is set to 0 whenever *onEvent*(μ) is true in the current state and is incremented by the duration of the last executed instruction, in each state in which *onEvent*(μ) does not hold. When the function is evaluated the value of this variable is returned. The evaluation of function *count* and of the other functions derivable from `fun` requires an auxiliary array variable that keeps track of the process state in the last K time units. All such function can be evaluated by using the values stored in the array; the size of this array is finite and limited by the number of activities performed in the last K units.

Let us now describe how future temporal predicates can be evaluated. The evaluation of *Until*(ϕ, ξ) returns false if ϕ is false in the current state and true if ξ is true in the current state; otherwise an evaluator for the formula is started and remains active in all future states until either ϕ is false (in which case the evaluator returns false) or ξ is true (in which case it terminates returning true). The evaluation of *Within*(ϕ, K) requires an evaluator to be started if ϕ is not true in the current state; otherwise it returns true. The evaluator remains active until either ϕ becomes true (in which case it returns true) or K time units elapsed (in which case it returns false). To evaluate predicate *Between*(ϕ, ξ, K), an evaluator is started to check whether ϕ occurs. If this is the case it remains active for K time units and when this time interval elapses, it checks if ξ is true. If it is the case it returns true, otherwise false.

6.5 Performance

The performance of a model checker is influenced by the dimension of its two inputs: the model, which in the context of this work is a BPEL process extended with *AAs*, and a formula, which in our case is a *GA*. While the formula is generally small, the model can be huge and sometimes potentially infinite.

Model checking requires that the systems under analysis be finite, hence the main issue to consider when modeling is the involved data's size. In a BPEL process, variables can vary on a potentially infinite domain, that needs to be made finite using abstraction techniques. In our approach we represent the model by randomly generating the data over their domain: hence there is a trade-off between abstraction and precision of data generation. Indeed, the data representation is dually crucial; if we select coarse-grained domains for the variables, the model itself looses generality and significance; on the other hand, a fine-grained domain leads to an exponential blow-up during verification.

AAs may help reduce the range of data exchanged with the environment, by considerably affecting the space required for verification. For instance, by annotating our *TeleAssistance* process with the `VitalParams` assertion, we can reduce the size of the `$vitalParams/glucose` variable's domain, and as a consequence, the number of states visited during verification: in our experimentation, this value decreased from 422 to 282 states.

Another main issue that affects performance is the metric ALBERT induces over time, to support which we enrich the model with time annotations. The temporal domain we consider is discrete, but not finite. Hence, even though we explicitly consider time, we do not use a global temporal variable to represent it, since this would make verification infeasible, but local timers.

The timers' granularity is set using the state of the process with respect to the *GA* that must be verified, and to the time guards in the model. More precisely, whenever the passing of time does not affect the model's behaviour it is abstracted and considered as a single time slot.

7 Run-time Validation

When performing model checking, we distinguish between *AAs* and *GAs*. The former define assumptions on the outside world, while the latter define properties that, when statically verified, allow us to state the "internal" correctness of the workflow process. When moving from development time to run time, the workflow interacts with real external services. Such services, as we observed, are owned, run, and evolved by independent authorities. Compliance of their behavior with the properties assumed by the workflow at development time is not automatically ensured, and must be checked by monitoring. Moreover, to ensure system robustness, we may also be interested in monitoring the guaranteed assertions.

Our validation framework proposes a rule-based monitoring approach where *AAs* and *GAs* are checked at run time by Dynamo, which is our monitoring infrastructure for *dynamic monitoring*.

7.1 Monitoring Rules

Monitoring rules specify the directives for the monitoring framework, and are made up of two main parts: (1) a set of optional meta-level information called Monitoring Parameters, and (2) a Monitoring Property expressed in the ALBERT language.

Monitoring Parameters allow our approach to be flexible and adjustable with respect to the context of execution. Each rule is associated with a set of optional monitoring parameters. These are meta-level information used at run time to decide whether the rule should be taken into account or not. By default, if no parameters are given, monitoring is performed. Supported parameters are `priority`, `validity`, and `trusted providers`. A `priority` associated with a monitoring rule defines a simple notion of "importance". In our model we support five levels of priority: very low, low, medium, high, and very high. When the rule is about to be evaluated, its priority is compared with a threshold value[3]; 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. A `validity` parameter defines time constraints on *when* a supervision rule should be considered. The supervision designer can define two different kinds of constraints: time windows and periodicity. The former define time-frames within which monitoring is performed. When outside of this frame, any new monitoring activities are ignored. Rule checking, however, when started within a valid time-frame, is always completed. Monitoring periodicity, on the other hand, can be specified by using the `every` keyword. Accepted values are durations and dates, e.g., "every 3D", meaning every 3 days, or every "01/01", meaning every January 1st. Finally, `trusted providers` is a list of service providers for which supervision is not necessary. This is useful because, in abstract process definitions, the actual service to which the process binds could be chosen at development time or at run time. If a rule refers to at least one non-trusted provider, it is checked.

7.2 Dynamo

Figure 4 presents the Dynamo execution and monitoring framework, by illustrating the dependencies existing between the various components and the technologies used in the implementation.

The Configuration Manager is a storage component for all the ALBERT properties that have been defined. The ActiveBPEL engine is a modified version of ActiveBPEL [18] in which we embed monitoring. This is achieved by following an aspect oriented programming approach [19]. The engine is a Java program in which we weave the cross-cutting monitoring features via AspectJ [20]. ActiveBPEL works by creating an internal tree representation of the process being executed. In this

[3]The threshold value is set by the owner of the process.

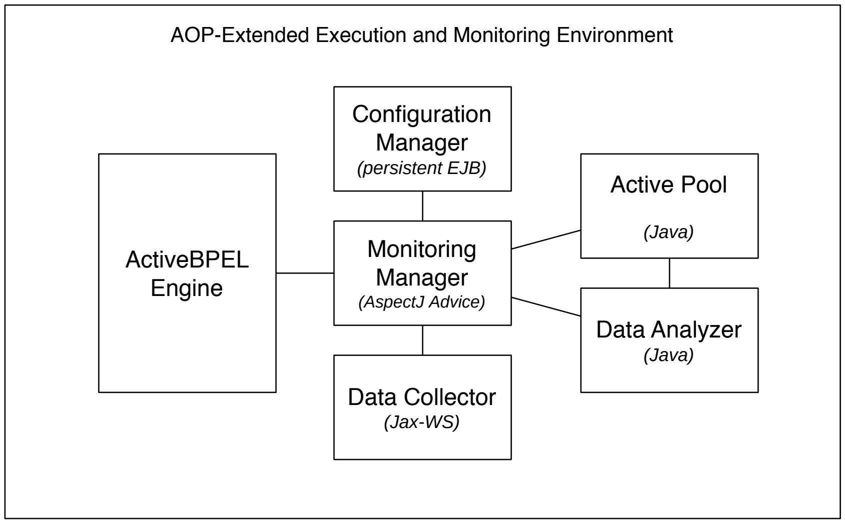


Figure 4: A run time view of the monitoring framework.

tree, each node represents a single BPEL activity in the process definition, and is an appropriate extension of the `AEActivityDefinition` class. Each node contains the information necessary to perform the particular activity it is associated with. At run time, the tree is visited and the definition classes are used by the engine to instantiate appropriate `AEActivityImpl` classes, all of which implement a common interface. Amongst other things, this interface provides an `execute` method where the activity's primary action is performed. For example, a *scope* activity will set up its internal variables, while an *invoke* activity will perform the appropriate external invocation. To perform monitoring, we intercept the process after the `execute` method is called for the various BPEL activities. These are the points where the Monitoring Manager (implemented as an AspectJ advice) is activated. Its main responsibility is data collection, both from within the process and from the outside world (in the case of external variables, through the Data Collector). The Monitoring Manager collects all the values of the variables defining the state of the process, and time-stamps and stores them in the Active Pool, together with the label of the BPEL activity after which they were collected. In practice, a complete process state is built and saved in the Active Pool.

The Active Pool is responsible for keeping track of historical sequences of process states. These states are needed by the Data Analyzer to perform analysis, and can be obtained in two ways. One way is to use an API, provided by the Active Pool, which can return the last state or a historical sequence of states. Another way is to subscribe to new states being generated (via the Publish/Subscribe paradigm), to receive them as they become available.

ALBERT formulae are evaluated by the Data Analyzer, which retrieves the data from the Active Pool, as described in the next section.

7.3 Evaluating ALBERT properties

This section describes how Dynamo evaluates ALBERT formulae in an intuitive manner. The Active Pool and the Data Analyzer perform optimizations that are ignored here for simplicity.

The evaluation of ALBERT formulae that do not contain references to the present state and/or to the past history (i.e., formulae that do not contain *Until*, *Between*, or *Within* operators) can be performed by the Data Analyzer by retrieving the relevant values from the Active Pool. We show that, given a formula ϕ that describes an assertion to be monitored at run time, the size of the state history to be kept by the Active Pool is limited. For the sake of simplicity, we first assume the formulae do not contain nested past operators.

When considering $past(\psi, onEvent(\mu), n)$ functions, the Active Pool needs to keep n history states. In the case of $Becomes(\phi)$, the Active Pool needs to keep one history state. Assertion ϕ evaluated in such state must be false, and it must be true when evaluated in the current state. When considering $count(\phi, K)$ and $\mathsf{fun}(\psi, K)$ functions, K represents a historical time window, going back from the current time. Thus, the Active Pool needs to keep all the history states that were collected within this time frame, whose number is of course bound. Therefore, the number of states in the Active Pool will be the maximum among the maximum of the n_i states needed for the $past_i$ functions, 1 (if there is a *Becomes* predicate), and the maximum number of states needed for the various *count* and fun time windows.
To illustrate a case where the formula contains nested operators that refer to the past, we analyze the following expression: $past(past(\psi, onEvent(\mu), a), onEvent(\mu'), b)$ where a and b are two Natural numbers. The size of the history sequence to be kept to evaluate the expression is $a + b$. In general, nested operators that refer to the past lead to a bounded growth of the history sequence.

Function *elapsed* needs special attention. For each event μ appearing in the argument of $onEvent(\mu)$, the Active Pool keeps one temporal distance variable. The variable is initially undefined, and is set to zero the first time $onEvent(\mu)$ becomes true. The time-stamp is updated upon receiving new states to store the amount of time passed from the last time $onEvent(\mu)$ was true.

The evaluation of formulae that contain *Until*, *Between*, or *Within* predicates is more complex. From a theoretical viewpoint, it could be explained by referring to the well known correspondence between Linear Temporal Logic and Alternating Automata [21].

Conceptually, the evaluation of these formulae cannot be completed in the current state. Their value, in fact, depends on the values the variables will assume in future states. For this reason, as soon as the Data Analyzer needs to evaluate one such subformula, it spawns a new evaluation thread for that subformula. The thread terminates in some future time instant. If the formula comprises sub-formulae, its evaluation can only be completed when all threads spawned by the evaluation terminate.

Whenever a new state is stored in the Active Pool, the Data Analyzer is notified. Consequently, each of the threads that was spawned for the evaluation of temporal subformulae is evaluated, according to the following rules:

- If the subformula is of the type $Until(\phi, \xi)$, ξ is evaluated by the thread in the state notified by the Active Pool and, if it is false, ϕ is evaluated. If ϕ is also false, the evaluation thread terminates by returning false. Otherwise, the thread continues to evaluate the subformula in future states.

- If the subformula is of the type $Between(\phi, \xi, K)$ a timer is associated with its evaluation thread. The timer is initialized to 0 the first time ϕ evaluates to true in a state notified by the Active Pool. As the timer reaches K, the thread terminates by returning the value of ξ in the current state.

- If the subformula is of the type $Within(\phi, K)$, the thread checks the truth of ϕ. If it is true, the thread terminates by returning true. If it is false, a timer (initialized to 0) is associated with the thread. If ϕ becomes true before the timer reaches K, the thread terminates by returning true; otherwise, it terminates by returning false.

Consider for example property MDCheckUp, which is recalled here for convenience:

$$\forall x(\mathit{Becomes}(\mathit{count}(\texttt{\$alarmNotif/level='high'} \wedge x = \texttt{\$alarmNotif/pID}, \mathit{onEvent}(\texttt{pButtonMsg}), 10080) = 3) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{patHospitalizedMsg}) \wedge \texttt{\$patHospitalized/pId=x}, 1440))$$

The formula includes elements of different nature: two state variables `$alarmNotif/level` and `$alarmNotif/pID`, one *pick* message `pButtonMsg`, and one event `patHospitalizedMsg` and its associated variable `$patHospitalized/pId`.

The formula is universally quantified over the patients, hence it is rewritten as a conjunction of formulae of the same structure, where variable `x` is substituted each time with a different patient. More precisely, if the set of patients is $P = \{p_1, \ldots, p_N\}$, the formula is internally considered as:

$$\bigwedge_{i=1}^{N}(\mathit{Becomes}(\mathit{count}(\texttt{\$alarmNotif/level='high'} \wedge p_i = \texttt{\$alarmNotif/pID}, \mathit{onEvent}(\texttt{pButtonMsg}), 10080) = 3) \rightarrow \mathit{Within}(\mathit{onEvent}(\texttt{patHospitalizedMsg}) \wedge \texttt{\$patHospitalized/pId=}p_i, 1440))$$

Each component of the conjunction is monitored by the Data Analyzer as follows, taking in account that, since there is an external conjunction, all the subformulae have to be true.

Operator *Becomes* is evaluated for a single patient by checking the value of its argument in the previous and in the current state. To evaluate such an argument in either state, function *count* must be evaluated, and this requires examining the historical sequence of states that occurred in the previous time-span, starting from the time-stamp of the state we are considering (either the previous or the current). The evaluation of the consequent of the formula is evaluated only when the *Becomes* is true and spawns a thread, which terminates, at the latest, after 1440 time units from the current time value. This is the latest time at which the value of the overall formula becomes known.

7.4 Performance

Many different aspects must be considered when studying the time Dynamo takes to verify an ALBERT expression. However, before starting our analysis, we must recall that monitoring is achieved mainly asynchronously. As we shall see, in a realm in which long ongoing processes are the norm, this will turn out not to be a real issue.

Monitoring can be broken down into various steps. In the first step the process execution is intercepted to build a new process state and to save it to the Active Pool. This is the only step taken synchronously, as the Monitoring Manager performs data collection from the process and from external services. Tests performed on an AMD Athlon(tm) XP 2600+ (1.93GHz) with 512MB of RAM, running Windows XP, show that our system takes less than 2ms to intercept the execution and to commence data collection. Since the Monitoring Manager lives in the same application space of the executing process, internal variables are collected extremely rapidly leveraging ActiveBPEL's own APIs (i.e. in a time quantifiable in milliseconds). External variables, on the other hand, are a completely different matter. The time needed to obtain data from an external service mostly depends on issues we are not responsible for, such as network-related issues or middleware serialization and de-serialization. The more the designer decides to include external variables in his properties, the more this obviously becomes an issue. In general, however, literature has taught us to expect the use of external variables to be limited with respect to internal variables. Nevertheless, many interesting properties can only be expressed with the help of external data, leaving the decision of how much external variables to use up to the designer.

Once data collection has been completed, the data are time-stamped and sent to the Active Pool, while the process is free to proceed. From this point on all monitoring activities are performed asynchronously, meaning they have no further impact on performance.

On the other hand, the time needed to actually verify a property depends solely on the property itself. If a property does not contain any *Until*, *Between*, or *Within* predicates, it can be evaluated immediately, and in long running business processes this may very well occur before the next significant business step is taken. Moreover, if any one of these predicates appears, the analysis time will depend on the evolution of the process execution (i.e., the appearance of new process states in the Active Pool), and on the timers mentioned in Section 7.3. Regardless of the property, the result will be known in an amount of time that depends linearly on and is bounded by the time constants used in these predicates. When the process terminates, all properties still under analysis are immediately evaluated considering that internal variables can no longer evolve. Properties involving external variables remain pending, for a bounded amount of time, waiting for the external data to become available.

8 Related Work

In this section we discuss some related approaches. We first review the existing work on the application of model checking to Web services. Then, we move to run-time monitoring. Besides introducing the different proposals, we also describe how our approach is different from the others.

8.1 Model Checking

Research on model checking Web service compositions is quite recent, but it has attracted considerable attention.

WSAT [22] is a framework for analyzing the interactions among composite Web services modeled as conversations. BPEL specifications of Web services are translated into an intermediate representation, an XPath-guarded automaton augmented with unbounded queues for incoming messages. This model is then translated into Promela and LTL properties, which can also be derived from XPath expressions, are then checked with the SPIN model checker [23].

The Verbus verification framework [24] is based on an intermediate formalism, which decouples the approach from any particular process definition language or verification tool. The support for BPEL is incomplete: *compensation* and *event handlers* are not considered. The current version of the prototype performs reachability analysis and supports the verification of properties like invariants, goals, activity pre- and post-conditions, as well as generic properties defined in temporal logic.

Nakajima [25] proposes a method to extract the behavioral specification from a BPEL process and to analyze it by using the SPIN model checker. A finite-state automaton extended with variable annotations (definitions and updates) is used as an intermediate representation. This approach provides only partial BPEL support , which does not deal with *fault/event/compensation handlers*. The tool checks for deadlocks and verifies user-defined LTL properties.

Table 1 summarizes the results of comparing our approach with the three previous SPIN-based verification frameworks in terms of the support they provide to the BPEL language. BPEL2BIR is

Table 1: Comparison of BPEL constructs support among model checking approaches.

BPEL constructs	Bpel2Bir	WSAT	Verbus	Nakajima
basic + structured activities[a]	yes	yes	yes	yes
fault handler	yes	yes	yes	no
event handler	yes	no	no	no
compensation handler	yes	no	no	no

[a]Activities described in §11 and §12 of [4].

the only approach that supports all the constructs of the language; all the others have some limitations in dealing with handlers.

Other authors use different computational models for verifying BPEL processes. Schlingloff et al. [26] uses Petri Nets to define the semantics of BPEL. Validation is performed by using the LoLA [27] model checking tool. Process algebras are used in [28] and [29]. Foster et al. [28] verify web service compositions against properties created from design specifications and implementation models. Specifications, in the form of Message Sequence Charts, and implementations, in the form of BPEL processes, are translated into the Finite State Process notation, which is the input language for the LTSA (Labelled Transition System Analyzer) model checker. Koshkina and van Breugel uses a process algebra, the BPE-calculus, to abstract the BPEL control flow. This calculus is used as input for a process algebra compiler to produce a front-end for the concurrency workbench (CWB) [30], which performs equivalence checking, preorder checking and model checking.

8.2 Monitoring

Several works define specification languages for functional and non-functional properties (usually expressed in the form of an SLA, Service Level Agreement) and propose an associated monitoring architecture. Sahai et al. [31] describe an automated and distributed SLA monitoring engine. The monitor acquires data from instrumented processes and —by analyzing the execution of activities and message passing— then verifies the SLAs. Keller and Ludwig [32] propose a framework to define and monitor SLAs, focusing on QoS properties such as performance and costs. The language defines a type system for various SLA artifacts such as parties, obligations, parameters, metrics and functions. The monitoring component is composed of two services. The first (the *measurement service*) measures parameters defined in the SLA, by probing client invocations or by retrieving metrics from internal resources. The second (the *condition evaluation service*) tests measured values against the thresholds defined in the SLA and, in case of a violation, triggers corrective management actions. Skene et al. [33] propose the SLAng language for SLAs; in [34] they extend their work by providing a model and an analysis technique for reasoning about the monitorability of SLAs.

All these approaches focus on formally defining high-level contracts among parties (typically, between a service consumer and a service provider), hence they do not allow to specify properties that should hold on specific events occurring during the execution of the service, such as the completion of a certain activity.

Robinson [35] uses temporal logic and KAOS to express requirements, such as timeliness constraints. These requirements are then analyzed to identify conditions under which they can be violated. If such conditions correspond to a pattern of events observable at run time, each of them is assigned to an agent for monitoring. At run time, an event adaptor translates SOAP messages into events and forwards them to the corresponding monitoring agent.

Mahbub and Spanoudakis [36] propose a framework for the run-time verification of requirements of service-based software systems. Requirements can be behavioral properties of a service

Table 2: Comparison of monitoring approaches.

Approach	Language		Abstraction		Properties		Directives			Timeliness		
	Logic	HL/VHL	Domain	Implementation	Safety	Temporal	Process	Activity	Event	Post-Mortem	Synchronous	Asynchronous
Sahai et al. [31]		x	x			x	x					x
Keller and Ludwig [32]		x	x			x	x					x
Skene et al. [33, 34]		x	x			x	x					x
Robinson [35]	x		x		x	x	x			x		
Mahbub and Spanoudakis [36]	x		x		x	x	x				x	x
Barbon et al. [37]	x	x		x		x	x	x	x			x
ALBERT	x	x		x	x	x	x	x	x			x

composition, or assumptions on the behavior of the different services composing the system. The first can be automatically extracted from the composite process, expressed in BPEL; the latter are specified by system providers using the event calculus. System events are collected at run time and stored in an event database; defined properties are checked by means of an algorithm based on integrity constraint checking in temporal deductive databases.

Barbon et al. [37] describe an approach to monitor BPEL compositions. Monitors can be attached to a single instance or to the whole class of process instances; they can check temporal, boolean, time-related and statistics properties, expressed in a run-time monitoring specification language. Business and monitoring logics are kept separated by executing in parallel the monitor engine and the BPEL execution engine; code implementing monitors is automatically generated from high level specifications.

A comparison of our approach against the others mentioned above is presented in Table 2. The classification of the approaches follows the taxonomy presented by Delgado et al. in [38], with some modifications/extensions of the metrics to adapt them to the service-oriented context. *Language* indicates the type of specification used by the approach (*logic* or *HL/VHL*), *abstraction* indicates the abstraction level at which properties are defined (*domain* or *implementation*), *properties* is used to indicate the kind of properties definable by the language (*safety* or *temporal*), *directives* indicates the level at which a property can be evaluated (*process*, *activity*, *event*), *timeliness* indicates when the monitoring activity is performed (*post-mortem*, *synchronous* or *asynchronous*).

The comparison shows that ALBERT is one of the few logic-based specification languages to fully support BPEL. This means that we can define assertions that predicate both on the whole process and on the single activities or events.

9 Conclusions

Service oriented architectures provide unprecedented degrees of dynamism and flexibility to software systems. Independently developed and deployed services are made available dynamically and then composed by third parties to provide new useful features. Web services achieve these goals at the Internet level, supporting dynamic federations of business services. The intrinsically dynamic nature of these systems and the multiple stakeholders involved in their construction and composition, however, challenge our ability to provide dependable solutions. In particular, the traditional boundary between development time, during which applications are carefully validated, and run time, during which systems are operated in the real world, disappears.

Given such a premise, our goals are to provide designers with a coherent validation framework for composite services described in BPEL. We believe that designers can benefit from a language — ALBERT — built from the ground-up for specifying functional and non-functional properties, both for design-time and run-time validation. This is why we have also set out to provide appropriate model-checking tools and a monitoring-aware execution environment. In conclusion, such a framework allows designers to produce more dependable solutions and to promptly discover whether their systems deviate from an expected and desirable quality of service.

Our future work will focus on exploiting the results of run-time monitoring, by providing mechanisms and strategies to react to the detection of undesirable behaviors. Our goal will be to incorporate self-managing features in service oriented architectures to allow service compositions to reorganize themselves to dynamically optimize the overall quality of service. Moreover, regarding the design-time validation, we plan to exploit Bogor's extensibility for further development. The CEGAR (Counterexample Guided Abstraction Refinement) [11] loop and predicate abstraction [39] state space reduction techniques — which proved to be highly beneficial when applied to software model checking — may be implemented as Bogor plugins to improve verification efficiency.

10 Acknowledgments

Part of this work has been supported by the IST EU project *SeCSE* - contract number 511680, the IST EU project *PLASTIC* - contract number 026955, the Italian FIRB project *ART DECO*, and the FAR project *DISCORSO*.

References

[1] Baresi L, Di Nitto E, Ghezzi C. Towards Open-World Software: Issues and Challenges. IEEE Computer. 2006 October;39:36–43.

[2] Jini Network Technology [homepage on the Internet]. Sun Corporation; 2007. Available from: `http://sun.com/jini`.

[3] OSGi [homepage on the Internet]. OSGi Alliance; 2007. Available from: `http://www.osgi.org`.

[4] Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, et al. Business Process Execution Language for Web Services, Version 1.1; 2003. BPEL4WS specification.

[5] Colombo M, Di Nitto E, Mauri M. SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rule. In: Service-Oriented Computing - ICSOC 2006, 4th International Conference, Proceedings. vol. 4294 of Lecture Notes in Computer Science. Springer; 2006. p. 191–202.

[6] Clarke EM, Grumberg O, Peled DA. Model checking. Cambridge, MA, USA: MIT Press; 1999.

[7] Meyer B. Applying "Design by Contract". Computer. 1992 Oct;25(10):40–51.

[8] Luckham DC, von Henke FW, Krieg-Brueckner B, Owe O. ANNA: a language for annotating Ada programs. New York, NY, USA: Springer-Verlag; 1987.

[9] Leavens GT, Baker AL, Ruby C. JML: A Notation for Detailed Design. In: Kilov H, Rumpe B, Simmonds I, editors. Behavioral Specifications of Businesses and Systems. Boston: Kluwer Academic Publishers; 1999. p. 175–188.

[10] Henzinger TA, Jhala R, Majumdar R, Sutre G. Lazy abstraction. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, NY, USA: ACM Press; 2002. p. 58–70.

[11] Clarke EM, Grumberg O, Jha S, Lu Y, Veith H. Counterexample-Guided Abstraction Refinement. In: CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification. vol. 1855 of Lecture Notes in Computer Science. Springer; 2000. p. 154–169.

[12] Robby, Dwyer MB, Hatcliff J. Bogor: an extensible and highly-modular software model checking framework. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. New York, NY, USA: ACM Press; 2003. p. 267–276.

[13] Bianculli D, Ghezzi C, Spoletini P. A model checking approach to verify BPEL4WS workflows. In: Proceedings of the 2007 IEEE International Conference on Service-oriented Computing and Applications. IEEE Computer Society Press; 2007. p. 13–20.

[14] Emerson EA, Sistla AP. Symmetry and model checking. Formal Methods in System Design. 1996;9(1-2):105–131.

[15] Robby, Dwyer MB, Hatcliff J, Iosif R. Space-Reduction Strategies for Model Checking Dynamic Systems. In: Proceedings of the 2003 Workshop on Software Model Checking. vol. 89 of Electronic Notes in Theoretical Computer Science. Elsevier; 2003. p. 499–517.

[16] Godefroid P. Using Partial Orders to Improve Automatic Verification Methods. In: CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification. vol. 531 of Lectures Notes in Computer Science. London, UK: Springer; 1991. p. 176–185.

[17] Baresi L, Ghezzi C, Mottola L. Towards Fine-grained Automated Verification of Publish-Subscribe Architectures. In: Proceedings of the 26th International Conference on Formal Methods for Networked and Distributed Systems (FORTE06). vol. 4229 of Lectures Notes in Computer Science. Springer; 2006. p. 131–135.

[18] ActiveBPEL Engine Architecture [homepage on the Internet]. Active Endpoints; 2006. Available from: `http://www.activebpel.org/docs/architecture.html`.

[19] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM, et al. Aspect-Oriented Programming. In: ECOOP'97 - Object-Oriented Programming, 11th European Conference, Proceedings. vol. 1241 of Lecture Notes in Computer Science. Springer; 1997. p. 220–242.

[20] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An Overview of AspectJ. In: ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Proceedings. vol. 2072 of Lecture Notes in Computer Science. Springer; 2001. p. 327–353.

[21] Kupferman O, Vardi M. Weak Alternating Automata Are Not That Weak. In: Fifth Israle Symposium on Theory of Computing and Systems, ISTCS'97, Proceedings. IEEE Computer Society Press; 1997. p. 147–158.

[22] Fu X, Bultan T, Su J. Analysis of interacting BPEL web services. In: WWW '04: Proceedings of the 13th international conference on World Wide Web. New York, NY, USA: ACM Press; 2004. p. 621–630.

[23] Holzmann GJ. The Model Checker SPIN. IEEE Transactions on Software Engineering. 1997;23(5):279–295.

[24] Arias-Fisteus J, Fernández LS, Kloos CD. Formal Verification of BPEL4WS Business Collaborations. In: E-Commerce and Web Technologies, 5th International Conference, EC-Web 2004, Proceedings. vol. 3182 of Lecture Notes in Computer Science. Springer; 2004. p. 76–85.

[25] Nakajima S. Model-Checking Behavioral Specification of BPEL Applications. Electronic Notes in Theoretical Computer Science. 2006;151(2):89–105.

[26] Schlingloff BH, Martens A, Schmidt K. Modeling and Model Checking Web Services. Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems. 2005 March;126:3–26.

[27] Schmidt K. LoLA: A Low Level Analyser. In: Application and Theory of Petri Nets, 21st International Conference (ICATPN 2000). vol. 1825 of Lecture Notes in Computer Science. Springer; 2000. p. 465–474.

[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. p. 152–163.

[29] Koshkina M, van Breugel F. Verification of Business Processes for Web Services. 4700 Keele Street, Toronto, M3J 1P3, Canada: York University - Department of Computer Science; 2003. CS-2003-11.

[30] Cleaveland R, Parrow J, Steffen B. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. ACM Transactions on Programming Languages and Systems. 1993 January;15(1):36–72.

[31] Sahai A, Machiraju V, Sayal M, Jin LJ, Casati F. Automated SLA Monitoring for Web Services. In: Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management - Management Technologies for E-Commerce and E-Business Applications. vol. 2506 of Lecture Notes in Computer Science. Springer; 2002. p. 28–41.

[32] Keller A, Ludwig H. Defining and Monitoring Service-level Agreements for Dynamic e-business. In: Proceedings of the 16th Conference on Systems Administration. Berkeley, CA, USA: USENIX Association; 2002. p. 189–204.

[33] Skene J, Lamanna DD, Emmerich W. Precise Service Level Agreements. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2004. p. 179–188.

[34] Skene J, Skene A, Crampton J, Emmerich W. The monitorability of service-level agreements for application-service provision. In: WOSP '07: Proceedings of the 6th international workshop on Software and performance. New York, NY, USA: ACM Press; 2007. p. 3–14.

[35] Robinson WN. Monitoring Web Service Requirements. In: RE'03: Proceedings of the 11th IEEE International Conference on Requirements Engineering. Washington, DC, USA: IEEE Computer Society; 2003. p. 65.

[36] Mahbub K, Spanoudakis G. A framework for requirements monitoring of service based systems. In: ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing. New York, NY, USA: ACM Press; 2004. p. 84–93.

[37] Barbon F, Traverso P, Pistore M, Trainotti M. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In: ICWS '06: Proceedings of the 2006 IEEE International Conference on Web Services. Washington, DC, USA: IEEE Computer Society; 2006. p. 63–71.

[38] Delgado N, Gates AQ, Roach S. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. IEEE Transactions on Software Engineering. 2004;30(12):859–872.

[39] Graf S, Saidi H. Construction of Abstract State Graphs with PVS. In: CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification. vol. 1254 of Lecture Notes in Computer Science. Springer; 1997. p. 72–83.

A Formal semantics of ALBERT

In this appendix we provide the formal definition of the semantics of the core of ALBERT. We start by formalizing the notions of *state* and *sequence* of states.

A *state* is defined as a triple $(V,\ I,\ t)$, where V is a set of $\langle variable, value\rangle$ pairs, I is a location of the process and t a time-stamp. A state completely describes the system in the particular time instant indicated by the time-stamp. States can be considered snapshots of the process. A location is defined as a set of labels of BPEL activities; in the case of a *flow* activity, it contains, for each branch of the *flow*, the last instruction executed in that branch. The set is needed to deal with *flow* activities; it is a singleton if the workflow processes do not contain *flow* activities. Sequences of states are often called timed state words, defined as follows:

Definition A *timed state word* is an infinite sequence $s = s_1,\ s_2,\ \ldots$ such that $s_i = (V_i,\ I_i,\ t_i)$.

As a consequence of how states are defined, timed state words are strictly monotonic. In fact, between subsequent states there is always at least one time consuming interaction with the outside world or one internal activity execution (e.g., an *assign* activity).

The formal semantics of ALBERT operators can be defined as follows. Given a timed state word $s = s_1,\ s_2,\ \ldots\ s_i,\ \ldots$, we introduce a helper function $\mathsf{numState}(i,K)$ where i is an index of a state in the word and $K > 0$ is a real value denoting a time interval. The function returns the number of states in the word, encountered in the time window of size K by moving backwards in past from the i-th state:

- $\mathsf{numState}(i,K) = 1$ iff $t_i - t_{i-1} > K$
- $\mathsf{numState}(i,K) = 1 + \mathsf{numState}(i-1, K-(t_i - t_{i-1}))$ iff $t_i - t_{i-1} \leq K$

An overloaded version is $\mathsf{numState}(K, onEvent(\mu), s_i)$ which operates exactly as before, with the only difference that it counts only the states in which $onEvent(\mu)$ is true.

We introduce now the function $\mathsf{eval}(\psi,\ s_i)$, which takes as parameters an ALBERT expression ψ and the state s_i in a word s and returns the value of ψ in s_i:

- $\mathsf{eval}(\mathsf{const},\ s_i) = \mathsf{const}$;
- $\mathsf{eval}(\mathsf{var},\ s_i) = \mathsf{value}$ iff $(\mathsf{var},\ \mathsf{value}) \in V_i$;
- $\mathsf{eval}(\psi_1\ \mathsf{arop}\ \psi_2,\ s_i) = \mathsf{eval}(\psi_1,\ s_i)\ \mathsf{arop}\ \mathsf{eval}(\psi_2,\ s_i)$;
- $\mathsf{eval}(\mathit{past}(\psi, \mathit{onEvent}(\mu), n),\ s_i) = \mathsf{value}$ iff $\exists j < i \mid\ \mathsf{eval}(\psi,\ s_j) = \mathsf{value}$ and $w,\ j \models \mathit{onEvent}(\mu)$ (the satisfiability relation $\models$ is defined below) and $\exists$ exactly $n-1$ disjoint values $h_1, \ldots, h_m \mid \forall m \in \{1, \ldots, n-1\},\ j < h_m < i$ and $w,\ h_m \models \mathit{onEvent}(\mu)$. If such a value of j cannot be found, $\mathsf{eval}(\mathit{past}(\psi, \mathit{onEvent}(\mu), n),\ s_i)$ is undefined;
- $\mathsf{eval}(\mathit{elapsed}(\mathit{onEvent}(\mu)),\ s_i) = \mathsf{value}$ iff $\exists j \leq i \mid\ w,\ j \models \mathit{onEvent}(\mu)$ and $\neg\exists h \mid j < h < i$ and $w, h \models \mathit{onEvent}(\mu)$ and $t_i - t_j = \mathsf{value}$;
- Let j be such that $j \leq i$, $t_i - t_j \leq K$ and $t_i - t_{j-1} > K$. Then $\mathsf{eval}(\mathit{sum}(\psi, K),\ s_i)$ is defined as follows:
  - if $i = j$ then $\mathsf{eval}(\psi,\ s_i)$,
  - if $i \neq j$ then $\mathsf{eval}(\psi,\ s_i) + \mathsf{eval}(\mathit{sum}(\psi, K - (t_i - t_{i-1})),\ s_{i-1})$.
- $\mathsf{eval}(\mathit{avg}(\psi, K),\ s_i) = \frac{\mathsf{eval}(\mathit{sum}(\psi,K))}{\mathsf{numState}(i,K)}$
- Let j be such that $j \leq i$, $t_i - t_j \leq K$ and $t_i - t_{j-1} > K$. Then $\mathsf{eval}(\mathit{max}(\psi, K),\ s_i)$ is defined as follows:
  - if $i = j$ then $\mathsf{eval}(\psi,\ s_i)$,
  - if $i > j$ then
    * if $\mathsf{eval}(\psi,\ s_i) < \mathsf{eval}(\mathit{max}(\psi, K - (t_i - t_{i-1})),\ s_{i-1})$ then $\mathsf{eval}(\mathit{max}(\psi, K - (t_i - t_{i-1})),\ s_{i-1})$
    * if $\mathsf{eval}(\psi,\ s_i) \geq \mathsf{eval}(\mathit{max}(\psi, K - (t_i - t_{i-1}))$ then $\mathsf{eval}(\psi,\ s_i)$
- Let j be such that $j \leq i$, $t_i - t_j \leq K$ and $t_i - t_{j-1} > K$. Then $\mathsf{eval}(\mathit{count}(\phi,\ K), s_i)$ is defined as follows:
  - if $i = j$ then
    * if $w,\ i \models \phi$ then 1,
    * if $w,\ i \not\models \phi$ then 0;
  - if $i > j$ then
    * if $w,\ i \models \phi$ then $1 + \mathsf{eval}(\mathit{count}(\phi, K - (t_i - t_{i-1})),\ s_{i-1})$,
    * if $w,\ i \not\models \phi$ then $\mathsf{eval}(\mathit{count}(\phi, K - (t_i - t_{i-1})),\ s_{i-1})$.

Function $\mathsf{eval}(\mathit{min}(\psi, K),\ s_i)$ can be computed similarly to $\mathsf{eval}(\mathit{max}(\psi, K),\ s_i)$. The overloaded version $\mathsf{fun}(\psi, \mathit{onEvent}(\mu), K)$ of functions fun is performed similarly to the evaluation of the original versions, but only considering the states in which $\mathit{onEvent}(\mu)$ holds.

For all timed word w, for all $i \in \mathbb{N}$, the *satisfaction relation* $\models$ is defined as:

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

Notice that even if the definition of the satisfaction relation recalls the function eval and viceversa, the two definitions are not cyclic.