SAVVY-WS at a glance: supporting verifiable dynamic service compositions*

Domenico Bianculli
University of Lugano - Faculty of Informatics
Lugano, Switzerland
domenico.bianculli@lu.unisi.ch

Carlo Ghezzi
Politecnico di Milano - DEEP-SE group - DEI
Milano, Italy
carlo.ghezzi@polimi.it

Abstract

Service-oriented architectures support the development of distributed and evolvable applications that live in an open world.

We focus on Web service compositions through which new added-value services are provisioned by integrating pre-existing services through a workflow. We assume that such pre-existing services can be selected and bound at run time to support continuous evolution and contextual adaptive policies.

We illustrate a methodology and a set of tools supporting both design-time and run-time verification of service compositions.

1. Introduction

Service-oriented architectures (SOAs) are emerging as a promising solution to the problem of developing decentralized, distributed, and evolvable applications that live in an open world [3]. In these architectures, *services* represent software components that provide specific functionality, exposed for possible use by many clients, who can then dynamically discover the services and access them through network infrastructures. This emerging scenario is highly dynamic, open, and decentralized; it is an instantiation of the *open-world software* scenario described in [3]: services may evolve dynamically and autonomously. New services may be developed and published in registries, and then discovered dynamically by possible clients. Previously available services may disappear or become unavailable.

The goal of SAVVY-WS (Service Analysis, Verification and Validation methodologY for Web Services) is to support the development and operation of added-value Web services built by composing third-party services through the workflow language BPEL [17]. In particular, it supports verifiable compositions, which are guaranteed to satisfy certain explicitly formulated global correctness properties, involving both functional and non-functional aspects. These properties are described in the ALBERT assertion language [2], which is briefly reviewed hereafter.

To support continuous evolution and contextual adaptation of composite services, we need dynamic binding, that is the ability to dynamically link service invocations from the workflow to specific service instances. This implies that at design time we should assume that external services orchestrated by the workflow are only known through their specifications, while their identity will only become known at run time.

SAVVY-WS supports design-time verification by a formal verification tool that can check whether a composite service delivers its expected functionality and meets the required quality of service, under the assumption that the external services used in the composition fulfill their specification.

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

Furthermore, during design-time verification, it is not possible to model the behavior of the network, which plays an important role in the provision of networked services. Although service providers' specifications could take into account, to some extent, the role of the network, it is virtually impossible to foresee all possible network conditions during design-time analysis.

To solve these problems, SAVVY-WS supports continuous verification by automatically generating run-time as-

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

sertions that are monitored to check for possible deviations from the correct behavior verified at design time. If a deviation is caught, suitable compensation policies and recovery actions should be activated.

SAVVY-WS is supported by several prototype tools that are currently being integrated in a comprehensive design and execution environment.

The paper is organized as follows. Section 2 informally introduces a simple motivating example of a composite service. Section 3 provides a brief introduction to the language we designed to express properties. Section 4 shows how SAVVY-WS supports the lifetime of the example composite service through design-time and run-time verification activities. Section 5 illustrates the related work. Section 6 concludes the paper.

2. Running example: *On Road Assistance*

This example is inspired by one of the scenarios developed in the context of the EU IST project SENSORIA[1]. We considered the *On Road Assistance* scenario, which takes place in an automotive domain, where a SOA interconnects (the devices running on) a car, service centers providing facilities like car repair, towing and car rental, and other actors.

The *On Road Assistance* process, is supposed to run on an embedded module in the car and is executed after a breakdown, when the car becomes not drivable.

The *Diagnostic System* sends a message with diagnostic data and the driver's profile (which contains credit card data, the allowed amount for a security deposit payment, and preferences for selecting assistance services) to the workflow, which starts by executing the `startAssistance` *receive* activity. Then, it starts a *flow* (named `flow1`) containing two parallel *sequence*s of activities.

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

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

[1] `http://www.sensoria-ist.eu`.

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

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

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

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

3. ALBERT

The ALBERT assertion language [2] is a temporal specification language for stating functional and non-functional properties of BPEL compositions. It is used in SAVVY-WS at design time according to an assume/guarantee specification and proof pattern. Certain ALBERT properties (*AAs–assumed assertions*) specify external services as seen by the workflow: they define the assumptions made on the external services that are composed. Other ALBERT properties (*GAs–guaranteed assertions*) define the properties the BPEL workflow ought to guarantee. GAs precisely state the proof obligations to be honored at design time. AAs precisely state the properties to be verified at run time, when external service invocations are bound to concrete services, to ensure that they behave as expected.

ALBERT formulae predicate over *internal* and *external* variables. The former represent data pertaining to the internal state of the BPEL process in execution. The latter represent data that are used in the verification, but are not part of the process' business logic and must be obtained externally (for example, by invoking other Web services, or

by accessing some global, persistent data representing historical information).

Given a finite set of variables V and a finite set of natural constants C, an ALBERT formula ϕ is defined according to the following grammar:

ϕ ::= χ | $\neg\phi$ | $\phi \wedge \phi$ | ((forall | exists) id in var ; ϕ) | *Becomes*(χ) | *Until*(ϕ,ϕ) | *Between*(ϕ,ϕ,K) | *Within*(ϕ,K)
χ ::= ψ relop ψ | $\neg\chi$ | $\chi \wedge \chi$ | *onEvent*(μ)
ψ ::= var | ψ arop ψ | const | *past*(ψ,*onEvent*(μ), n) | *count*(χ, K) | *count*(χ,*onEvent*(μ), K) | fun(ψ, K) | fun(ψ, *onEvent*(μ), K) | *elapsed*(*onEvent*(μ))
relop ::= $<$ | $\leq$ | $=$ | $\geq$ | $>$
arop ::= $+$ | $-$ | $\times$ | $\div$
fun ::= *sum* | *avg* | *min* | *max* | ...

where var $\in V$, const $\in C$, $n \in \mathbb{N}$, $K \in \mathbb{R}^+$ and *onEvent* is an event predicate. *Becomes*, *Until*, *Between* and *Within* are temporal predicates. *count*, *elapsed*, *past*, and all the functions derivable from the non-terminal fun are temporal functions of the language. Parameter μ identifies an event: the *start* or the *end* of an *invoke* or *receive* activity, the receipt of a message by a *pick* or an *event handler*, or the execution of any other BPEL activity. The above syntax only defines the language's core constructs. The usual logical derivations are used to define other connectives and temporal operators (e.g., $\vee$, *Always*, *Eventually*, ...).

As an example, a functional AA that should hold after the execution (as a post-condition) of an *invoke* activity `Act` on an external service S can be written as *onEvent*(`end_Act`) $\rightarrow$ *AA* where *AA* is a predicate on the values returned by activity `Act`.

It is also possible to express nonfunctional AAs, such as latency in a service response. An ALBERT formula that specifies that the duration of an *invoke* activity should not exceed 5 time units can be expressed as *onEvent*(`start_Act`) $\rightarrow$ *Within*(*onEvent*(`end_Act`),5). We leave the choice of the most suitable timing granularity to the verification engineer, who can then properly convert the informal system requirements to formal, real-time specifications [13].

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

ALBERT semantics is defined in [2] rather conventionally over a *timed state word*, an infinite sequence of states $s = s_1, s_2, \ldots$, where a state s_i is a triple (V_i, I_i, t_i). V_i is a set of $\langle \psi, value \rangle$ pairs, where ψ is an expression that appears in a formula, I_i is a location of the process and t_i is a time-stamp. States can therefore be considered as snapshots of the process.

4. SAVVY-WS

SAVVY-WS's goal is to support, by means of an integrated set of tools, the designers of composite services during the verification phase, which extends from design time to run time.

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

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

Therefore, the second step of the SAVVY-WS-aware development process is to annotate the BPEL process with AAs and GAs written in ALBERT. The BPEL process, annotated with ALBERT properties, is then translated by a tool, BPEL2BIR, into a representation suitable for static analysis. We use the Bogor model checker [8] to check that the GAs are satisfied, assuming that the external services behave as specified by the AAs.

The BPEL process is then put into operation by deploying it on a standard BPEL engine, which we have extended with Dynamo, our monitoring framework. At run time, Dynamo checks the BPEL process and the external services it interacts with, for possible deviations from the correct behavior verified at design time.

In the rest of this section, we show how ALBERT can be used as a specification language (Sect. 4.1), how verification is performed at design time via model checking (Sect. 4.2) and how continuous verification of service compositions can be achieved at run time (Sect. 4.3).

4.1. Specifying the *On Road Assistance* example

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

comment, when necessary. We assume that each time tick of the system represents one minute.

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

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

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

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

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

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

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

  $$\begin{aligned}&\mathit{onEvent}(\texttt{end\_findLocalServices}) \rightarrow \\ &\quad(\texttt{forall t in \$foundServices/[*] ;} \\ &\qquad(\texttt{forall s in \$foundServices/t/[*] ;} \\ &\qquad\texttt{s/distance} < 50))\end{aligned}$$

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

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

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

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

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

  $$\begin{aligned}&\mathit{onEvent}(\texttt{end\_OrderTowTruck}) \rightarrow \\ &\mathit{Within}(\mathit{onEvent}(\texttt{start\_TTPN}), 50)\end{aligned}$$

  where TTPN is the short form for TowTruckProgressNotice.

- AssistanceTimeliness: The tow truck that will be requested will be in proximity of the car within 60 minutes after the credit card is charged. This property must be guaranteed to the user by the *On Road Assistance* workflow. It is a GA, whose validity is (rather trivially) assured at design time by the AllButBankServicesResponseTime, the TowTruckServiceTimeliness and the TowTruckArrival AAs, and by the structure of the process. The property can be expressed as follows:

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

4.2. Model checking the *On Road Assistance* process

Our design-time verification phase is based on model checking. We developed BPEL2BIR, a tool that translates a BPEL process and its ALBERT properties into BIR (Bogor's input language).

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

```
enum TDiagnosticData {dd1, dd2}
enum TCustomerPreference {cp1, cp2}
enum TCreditCard {cc_c1, cc_c2}
record TStartMsg {
  TDiagnosticData diagData;
  TCreditCard ccData;
  int (1,10) deposit;
  TCustomerPreference cpData;
}
```

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

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

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

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

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

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

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

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

```
function launchAndWaitFlow1() {
  boolean temp0;
  loc loc0: do {
    flow1_sequence1_tid := start
        flow1_sequence1();
    flow1_sequence2_tid := start
        flow1_sequence2();
  } goto loc1;
  loc loc1: do {
    temp0 := threadTerminated(
        flow1_sequence1_tid)
             && threadTerminated(
                 flow1_sequence2_tid);
  } goto loc2;
  loc loc2: when temp0 do{} return;
            when !temp0 do{} goto loc1;
}
```

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

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

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

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

The next example shows how AAs can be used to define time constraints for modeling either the execution time of or the time elapsed between BPEL activities. The adopted technique is based on previous work on model checking temporal metric specifications [7]. We insert a code block that randomly generates the duration of the activity within a certain interval, bounded by the value specified in an AA. For *flow* activities, the time consumed by the *flow* is the maximum time spent along all paths. By focusing on `flow2` of our example and using properties AllButBankServiceResponseTime and TowTruckArrival, we get the following code:

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

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

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

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

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

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

Before emitting the actual BIR code, BPEL2BIR performs a static analysis on the flow graph of the BIR program to detect data variables (i.e., the ones associated with inbound messages activities like *receive* and *invoke*) that are not used in the computation of the process. If such variables exist, we perform an optimization that removes them and the corresponding generative code blocks from the BIR model, to reduce the size of the model itself.

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

4.3. Monitoring the *On Road Assistance* process

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

The monitoring framework is based on the ActiveBPEL[2] open-source BPEL server implementation, which has been extended with monitoring capabilities by using aspect-oriented programming (AOP). The advice code that is weaved into the engine is represented by the Data Manager. When the engine initiates a new process instance, the Data Manager loads all that process' ALBERT formulae from a Formulae Repository, and uses them to configure and activate both the Active Pool and the Data Analyzer. The former is responsible for maintaining (bounded) historical sequences of process states, while the latter is the actual component responsible for the analysis.

The Data Manager's main task stops the process every time a new state needs to be collected for monitoring. Besides referring to internal variables, ALBERT formulae may also refer to external data, which do not belong to the business logic itself; special-purpose Data Collectors are used to retrieve these data from external sources.

[2] `http://www.activevos.com/`.

After a process state is time-stamped, it is labeled with the location in the process from which the data were collected, and sent to the Active Pool, which stores it. Every time the Active Pool receives a new state it updates its sequences to only include the minimum amount of states required to verify all the formulae. The sequences are then used by the Data Analyzer to check the formulae.

The evaluation of ALBERT formulae that contain only references to the present state and/or to the past history (i.e., formulae that do not contain *Until*, *Between*, or *Within* operators) is straightforward. On the other hand, the evaluation of formulae that contain *Until*, *Between*, or *Within* operators depends on the values the variables will assume in future states. From a theoretical point of view, this could be expressed by referring to the well-known correspondence between Linear Temporal Logic and Alternating Automata [23]. From an implementation point of view, the Data Analyzer relies on additional evaluation threads for evaluating each subformula containing one of the three aforementioned temporal operators. In the rest of this section, we focus on the Data Analyzer, by describing how it evaluates the properties of our running example.

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

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

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

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

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

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

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

5. Related work

The work presented in [20] is similar to SAVVY-WS, since it also proposes a lifelong verification framework for service compositions. The approach is based on the Event Calculus of Kowalski and Sergot [14], which is used to model and reason about the set of events generated by the execution of a business process. At design time the control flow of a process is checked for livelocks and deadlocks, while at run time it is checked if the sequence of generated events matches a certain desired behavior. The main difference with SAVVY-WS is the lack of support for data-aware properties.

While SAVVY-WS focuses on lifelong verification of service compositions, the approach described in [6] focuses only on testing at design time; however, as SAVVY-WS, it considers both functional and non-functional properties. The core of the approach is a model-based stub generator, which uses state machines and SLAs (Service Level Agreements) expressed in WS-Agreement [18] to generate, respectively, functional and non-functional stubs.

Many other approaches investigated by current research tackle isolated aspects related to the main issue of engineering dependable service compositions. Design-time verification is addressed, for example, in [11], where the interaction between BPEL processes is modeled as a conversation and then verified using the SPIN model checker. In [10], design specifications (in the form of Message Sequence Charts) and implementations (in the form of BPEL processes) are

translated into the Finite State Process notation and checked with the Labelled Transition System Analyzer. Other approaches focus on run-time verification of service compositions, considering either the behavior, as in [1, 15], or the non-functional aspects [16, 19, 21], or both [9].

Design- and run-time verification activities are related to the language that is used to specify the properties that are to be checked. Besides approaches based on assertion languages like WSCoL [5] or behavioral description languages like the OpenModel Modeling language [12], there also proposals of languages for defining SLAs, such as WS-Agreement and SLAng [22], and policies, such as WS-Policy [24].

6. Conclusion and future work

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

Further developments of SAVVY-WS include improvements to the expressiveness of the ALBERT language, and enhancements to the verification techniques. In this direction, we will refine the way we handle timeliness-related properties, both at design time, by extending Bogor (starting from the work described in [4]), and at run time, by experimenting with different monitoring strategies.

References

[1] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06 Proceedings*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[2] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, 2007.

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

[4] L. Baresi, G. Gerosa, C. Ghezzi, and L. Mottola. Playing with time in publish-subscribe using a domain-specific model checker. In *SAVCBS '07 Proceedings*, pages 55–62. ACM, 2007.

[5] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC '05 Proceedings*, volume 3826 of *LNCS*, pages 269–282. Springer, 2005.

[6] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *Testcom/Fates 2008 Proceedings*, volume 5047 of *LNCS*, pages 266–282. Springer, 2008.

[7] D. Bianculli, P. Spoletini, A. Morzenti, M. Pradella, and P. San Pietro. Model checking temporal metric specification with Trio2Promela. In *FSEN 2007 Proceedings*, volume 4767 of *LNCS*, pages 388–395. Springer, 2007.

[8] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *CAV 2005 Proceedings*, volume 3576 of *LNCS*, pages 148–152. Springer, 2005.

[9] A. Erradi, P. Maheshwari, and V. Tosic. WS-Policy based monitoring of composite web services. In *ECOWS '07 Proceedings*, pages 99–108. IEEE Computer Society, 2007.

[10] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based Verification of Web Service Compositions. In *ASE 2003 Proceedings*, pages 152–163. IEEE Computer Society, 2003.

[11] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *WWW '04 Proceedings*, pages 621–630. ACM Press, 2004.

[12] R. J. Hall and A. Zisman. Behavioral models as service descriptions. In *ICSOC '04 Proceedings*, pages 163–172. ACM, 2004.

[13] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *ICSE '05 Proceedings*, pages 372–381. ACM, 2005.

[14] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Gen. Comput.*, 4(1):67–95, 1986.

[15] K. Mahbub and G. Spanoudakis. A framework for requirements monitoring of service based systems. In *ICSOC '04 Proceedings*, pages 84–93. ACM Press, 2004.

[16] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In *WWW'08 Proceedings*, pages 815–824. ACM, 2008.

[17] OASIS. Web Service Business Process Execution Language Version 2.0 Specification, 2007.

[18] Open Grid Forum. Web Services Agreement Specification (WS-Agreement). `http://www.ogf.org/documents/GFD.107.pdf`, 2007.

[19] F. Raimondi, J. Skene, , and W. Emmerich. Efficient monitoring of web service SLAs. In *SIGSOFT 2008 - FSE 16 Proceedings*. ACM, 2008. to appear.

[20] M. Rouached, O. Perrin, and C. Godart. Towards formal verification of web service composition. In *BPM 2006 Proceedings*, volume 4102 of *LNCS*, pages 257–273. Springer, 2006.

[21] A. Sahai, V. Machiraju, M. Sayal, L. J. Jin, and F. Casati. Automated SLA monitoring for web services. In *DSOM '02 Proceedings*, volume 2506 of *LNCS*, pages 28–41. Springer, 2002.

[22] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *ICSE '04 Proceedings*, pages 179–188. IEEE Computer Society, 2004.

[23] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher order workshop Proceedings*, volume 1043 of *LNCS*, pages 238–266. Springer, 1996.

[24] W3C Web Services Policy Working Group. WS-Policy 1.5. `http://www.w3.org/2002/ws/policy/`, 2007.