Monitoring Conversational Web Services

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

Carlo Ghezzi
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Via Ponzio 34/5, I-20133, Milano, Italy
ghezzi@elet.polimi.it

ABSTRACT

The dynamic nature of web service compositions demands continuous monitoring of the quality of the provided service, as perceived by the client. We focus here on monitoring functionality of conversational services, whose behavior depends on the local state resulting from the client-service interaction. We propose a monitoring approach based on an algebraic specification language and we show how this can be integrated into a run-time monitoring architecture.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*

General Terms

Verification

Keywords

Monitoring, algebraic specifications, services, BPEL

1. INTRODUCTION

Web services support a dynamic architectural style where the binding among components may change at run-time. New services may be developed and published in registries, and then discovered by possible clients. Previously available services may disappear or become unavailable. This situation has been characterized elsewhere [5] by the term *open-world software*, to describe a setting where applications are composed out of parts that may change unpredictably and dynamically. It has been observed that open-world software introduces the requirement of continuous validation. Since a software architecture evolves dynamically, validation must extend from development time to run time.

IW-SOSWE'07 September 3, 2007, Dubrovnik, Croatia

Monitoring is a necessary component of a run-time validation facility. Monitoring consists of collecting data about a monitored service and checking that the data conform to the expected behavior of that service. Thus monitoring requires that the expected behavior is known and explicitly described, so that it can be checked.

We assume the expected behavior to be described as a formal specification of the monitored service. The specification should state what the monitored service ought to do on behalf of its clients, in terms of the service's visible behavior and of other qualities it should exhibit. The specification should therefore include not only the syntactic aspects (i.e., the signature of the exported operations), but also the semantic properties of the service.

Past work in our group [6] focused on monitoring services whose behavior is specified by a contract [23] defined via pre- and post-conditions. The pre-condition states what the client must ensure to invoke the service correctly. It is an obligation imposed on the client. The post-condition states what the service promises to provide. Hence it is an obligation imposed on the service. Pre- and post-conditions are adequate if we need to express the functional behavior of stateless services. They are inadequate for stateful services whose expected behavior depends on the history of previous interactions between a client and a given service instance. These services are called *conversational services*. An example is a service providing a "shopping cart", an abstraction that is supported by most existing e-commerce web applications. The shopping cart is chosen as running example in this paper.

Run-time validation of conversational services has been largely unexplored by current research on Web services. Conversational services raise new challenges both to the specification language through which their behavior is formally described and to the way the actual behavior may be checked against the specification. Existing approaches, in fact, cannot be applied in this new context, but need to be rethought. In this paper, we explore a specification style and a monitoring strategy that is based on algebraic specifications of conversational Web services.

This paper is organized as follows. Section 2 investigates the nature of stateless versus stateful services, introduces the specific features of conversational Web services, and discusses why and how they differ from conventional stateless services. Section 3 focuses on specification methods for conversational Web services and motivates the use of algebraic specifications. It also provides some background material on algebraic specifications and shows how they can be used for

specification in our running example. Section 4 discusses how algebraic specification may be symbolically executed by a run-time checker. Section 5 illustrates our monitoring framework for conversational services. Section 6 provides a view of related current work. Finally, Section 7 draws some conclusions.

2. CONVERSATIONAL WEB SERVICES

In their simplest form, services are *stateless*, i.e. they provide a functional abstraction: the same service provides identical results, if it is invoked twice with the same arguments. If several instances of the same service are active for different clients, they all provide the same functional abstraction. An example can be a service that converts temperatures values from Celsius degrees to Fahrenheit.

More complex services require some notion of state, i.e. they are stateful [25]. It can be useful, however, to distinguish among different notions of stateful services.

A *Global Shared Resource (GSR) service* is a service whose computation depends on a global resource, that is shared among all instances of the services. As an example, a weather forecast service provides the weather forecasts for every city in a region based on global data supplied by the local weather forecast station. Although the service can be invoked like a function, the values it provides may change over time, depending on the modifications that third-party services (such as the local weather forecast station in the example) perform on the global shared resource.

Conversational services are yet another kind of stateful services. In a conversational service an instance of the service keeps a local state that depends on the conversation with its client. Every instance has its own local state, not visible to the others. Such kinds of services correspond to the well-known notion of *data abstractions*. Using familiar terms of object-oriented programming, each service can be viewed as an abstract data type and each instance as an object that can be manipulated only through operations exposed in the service interface, which may modify the local state of the object. As an example of a conversational service, we consider the well-known shopping cart, available on most web sites providing e-commerce facilities. Each client has its own instance of the shopping cart, which contains the items selected by the client. Figure 1 shows the interface of a shopping cart service: the operation `insert(Item i)` (respectively, `delete(Item i)`) adds (respectively, removes) an item `i` from the cart; the operation `amount()` returns the total amount of the items in the cart.
In the sequel, we will focus on monitoring conversational services.

3. PRELIMINARIES

Our goal is to monitor the actual behavior of services with respect to their expected behavior. This assumes that a specification of the expected behavior is available and can be used to monitor the service on the client's behalf.

Previous work in our group focused on monitoring services specified via pre- and post-conditions [6]. Pre- and post-conditions are awkward to use in the case of conversational services, and more generally, stateful services.

In the case of a stateful service, the effects of an operation depend not only on its input arguments but also on the state in which invocation occurs. Because the state is hidden, it would be necessary to introduce some form of state abstraction, to enrich the expressiveness of pre- and post-conditions. This is what is accomplished for example in JML [20], a behavioral specification language defined for Java, which allows for using model fields to abstract the underlying state of an object [9].

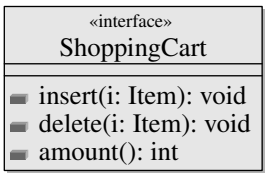

Figure 1: The `ShoppingCart` service interface

Algebraic specifications provide an alternate approach: the hidden state is implicitly taken care of by axioms that establish the equivalence of sequences of operations performed on the object. The next section provides a quick overview of algebraic specifications. Afterwards we show how they can be used to specify conversational services by formally introducing the shopping cart example.

3.1 Algebraic specifications

Algebraic specifications [15, 2, 7] are a well-know approach to formally specify software. This formalism is founded on the concept of an algebra, i.e. a set of data with operations on those data. An algebraic specification can be seen as the description of an algebra. It consists of a signature declaring the types of the algebraic operations and a set of logical formulae describing properties of these operations.

We illustrate this formalism using the shopping cart example reported in Figure 2. In the example, we do not use the specific syntax of any of the existing algebraic specification languages, but rather we use a neutral and simple to explain notation, similar to the one introduced in [24].
In the first line we declare other algebras that will be used in the specification, by means of the import clause; in this case we import IntSpec, supposed to contain the specification of integer numbers, and ItemSpec, which specifies the items that can be purchased. We assume that module ItemSpec exports sort Item, for which an operation price is defined. Module IntSpec can be assumed to be predefined. If not, sort int can also be defined by an algebraic module, based on Peano's axiomatization.

An algebra defines a type, and therefore the two terms can be used interchangeably when no confusion arises. The type definition is introduced by the keyword sort (line 3), followed by the signatures of the operations (lines 5–11). Operations can be classified in different categories: *constructors*, through which one can build any instance of a type, *observers*, which retrieve information from the instances of a type, and *transformers*, which denote operations whose specification can be defined in terms of *constructor* operations. Operation price is an observer of type Item, which is not specified for simplicity. The standard form of a signature is <operation-name> (<domain-types>) $\longrightarrow$ <range-types> where <domain-types> and <range-types> are comma-separated lists of types associated with the domain and the range of the operation.

The axioms section (lines 13–21) of the specification lists the logical formulae, also called *equations* or *axioms*, which

```
1  import IntSpec, ItemSpec;
2
3  sort cart;
4
5  constructors
6    create() ⟶ cart;
7    insert(cart, item) ⟶ cart;
8  observers
9    amount(cart)  ⟶ int;
10 transformers
11   delete(cart, item)  ⟶ cart;
12
13 axioms
14   forall c: cart, i, j: item
15
16   amount(create()) = 0;
17   amount(insert(c,i)) = amount(c)
         + price(i);
18   delete(create(),i) = create();
19   delete(insert(c,i),j) =
20     if (i == j) c
21     else insert(delete(c,j),i);
22 end
```

Figure 2: Algebraic specification of a shopping cart

define the semantics of the operations by stating equalities between terms.

Axioms at lines 16–17 define the semantics of the amount operation using a recursive style: the amount of an empty shopping cart (obtained as a result of the create operation) is zero, while the amount of the shopping cart resulting from the insertion of item i to an existing cart c is equal to the sum of the amount of cart c before the insertion of the new item plus the price of item i. A similar recursive style is used also to specify the semantics of the delete operation (lines 18-21).

4. EVALUATING ALGEBRAIC SPECIFICATIONS AT RUN-TIME

Algebraic specifications can be evaluated by symbolically executing them. Various approaches and tools have been proposed in the past; we are currently experimenting with Heureka [16] and CafeOBJ [14].

Evaluation is formally defined by a term rewriting system (TRS) [19, 12]. In the sequel we provide a glimpse of term rewriting and show how it can be used to execute algebraic specifications.

A TRS is a pair (Σ, R), where Σ is an *alphabet* (*signature*) and R is a set of *reduction rules* (*rewrite rules*). The alphabet Σ consists of:

1. a countably infinite set of *variables* $x, y, z, \ldots$
2. a non-empty set of operation names (*function symbols*) $F, G, \ldots$, each with its own arity.

The set of terms (or expressions) over Σ is $\mathrm{Ter}(\Sigma)$ and is defined inductively:

1. $x, y, z, \cdots \in \mathrm{Ter}(\Sigma)$,
2. if F is an n-ary function symbol and $t_1, t_2, \ldots, t_n \in \mathrm{Ter}(\Sigma)$, with $n \geq 0$, then $F(t_1, \ldots, t_n) \in \mathrm{Ter}(\Sigma)$.

A *substitution* σ is a mapping from $\mathrm{Ter}(\Sigma)$ to $\mathrm{Ter}(\Sigma)$ which satisfies $\sigma(F(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n))$ for every n-ary function symbol F. We also write t^σ instead of $\sigma(t)$.

In the shopping cart example, Σ is defined as $\mathcal{X} \cup \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2$, where $\mathcal{X}$ is the set of variables and $\mathcal{F}_i$ is the set of functions of arity i. Specifically, $\mathcal{X} = \{c, i, j\}$, $\mathcal{F}_0 = \{\mathsf{create}, 0\}$, $\mathcal{F}_1 = \{\mathsf{amount}\}$, $\mathcal{F}_2 = \{\mathsf{insert}, \mathsf{delete}, +\}$.

A *reduction rule* (or rewrite rule) is a pair (t, s) of terms $\in \mathrm{Ter}(\Sigma)$ and is written $t \to s$. A reduction rule $r : t \to s$ determines a set of *rewrites* $t^\sigma \to_r s^\sigma$ for all substitutions σ.

In operational semantics, the axioms of an algebraic specification can be regarded as rewrite rules of a TRS. Letting t, s, r three terms and b a boolean condition, we get:

- $t \to s$ is a rewrite rule for the equation t = s;
- the rewrite rule for the conditional equation t = if (b) s else r, is $t \to s$ if b is true, otherwise it is $t \to r$.

For simplicity, we omit to specify all the rewrite rules associated to the shopping cart example: we limit the presentation to the rules having an amount operation as their left term:

$$\mathsf{amount}(\mathsf{create}()) \to 0$$
$$\mathsf{amount}(\mathsf{insert}(c, i)) \to \mathsf{amount}(\mathrm{c}) + \mathsf{price}(i)$$

Other rules can be written in a similar way, starting from specification in Figure 2, by replacing in the axioms section, every occurrence of symbol '=' with the symbol '→'.

An interpreter works by applying a sequence of rewriting rules to a term, till the term reduces to a ground term, i.e. a constant.

Following our running example, consider the term deriving from the execution of this sequence of operations:
1) create a cart;
2) insert item I_1;
3) insert item I_2;
4) delete item I_1;
5) insert item I_3;
6) get the amount of the cart.
where I_i denotes, for brevity, the item whose price is i.
The resulting term and the sequence of rewrite rules that reduce it to a constant are shown in Figure 3, which is a simplified pretty-printing of a session with CafeOBJ. The reduction of the term took 0.040s on a machine with a 2.1GHz Intel Pentium M Processor, 1GB RAM, running CafeOBJ ver. 1.4.6p10 on Apple Mac OS X.

5. MONITORING ALGEBRAIC SPECIFICATIONS OF CONVERSATIONAL WEB SERVICES

In this section we illustrate how conversational web services can be monitored. We first enrich our running example in Section 5.1 by specifying a workflow that invokes an external shopping cart service. Section 5.2 then focuses on the architecture of the proposed solution for the monitor component. Section 5.3 ends by discussing the integration of the evaluator for algebraic specifications in the monitor.

5.1 Extended example

We assume that the shopping cart service introduced earlier has to be integrated into a workflow that supports an

```
CART> reduce amount(insert(I3, delete(I1, insert(I2, insert(I1, create())))))
1>[1] apply trial -- rule: ceq delete(X:Item,insert(Y:Item,C:Cart))  = insert(Y,delete(X,C)) if X =/= Y
1<[1] delete(I1,insert(I2,insert(I1,create()))) --> insert(I2, delete(I1, insert(I1, create())))
1>[2] apply trial-- rule: ceq delete(X:Item,insert(Y:Item,C:Cart)) = C if X == Y
1<[2] delete(I1,insert(I1,create())) --> create()
1>[3] rule: eq amount insert(X:Item,C:Cart) = X.price + amount(C)
1<[3] amount insert(I3,insert(I2,create())) --> 3 + amount(insert(2,create()))
1>[4] rule: eq amount insert(X:Item,C:Cart) = X.price + amount(C)
1<[4] amount(insert(I2,create())) --> 2 + amount(create())
1>[5] rule: eq amount(create()) = 0
1<[5] amount(create()) --> 0
1>[6] rule: eq [:BDEMOD] : M:Nat + N:Nat = #! (+ m n)
1<[6] 2 + 0 --> 2
1>[7] rule: eq [:BDEMOD] : M:Nat + N:Nat = #! (+ m n)
1<[7] 2 + 3 --> 5
CART>
```

Figure 3: Evaluation of a symbolic term

advanced shopping session. The workflow is defined in the de-facto standard BPEL language [1]. The workflow is instructed by an end-user to shop for a list of items, searching different on-line stores and seeking for the best price for each item. The stores can be sought among those for which the end-user owns a shopping card. The workflow is sketched by the flowchart in Figure 4.

An end-user starts the shopping session by invoking the startShopping operation and passing a message containing the list of the shopping cards owned by her and her credentials, received by the first activity of the workflow. After logging in the shopping cart service (loginCart operation), the workflow starts a loop, waiting for messages from the user (`pick` activity). The workflow may receive two kinds of messages: buyItem and stopShopping.

In case of receiving a message of the first type, the workflow loops over the stores, seeking for the best price for the item indicated in the message. The workflow queries a store (searchStore operation) and starts analyzing its response message. We assume that the response message contains information on the availability and the price of the item. If the item is not available (condition checked in the first `switch` activity), the workflow starts immediately a new iteration of the loop, by querying another store; otherwise, the workflow proceeds examining the price of the item (second `switch` activity), according to the following conditions:

- *no instance of the current item is already in the cart*: the item is added to the shopping cart (insert operation);
- *the price proposed by the just-queried store is higher than the one proposed by a previously-visited store*: the current item is discarded, nothing is done;
- *the price proposed by the just-queried store is lower than the one of the instance of the item currently stored in the cart*: the instance of the item present in the cart is removed (delete operation) and the new one is inserted (insert operation).

All branches of the second `switch` activity lead to another iteration of the loop.

If stopMessage is received, the shopping cart service is queried (amount operation) to get the total amount of the items in the cart. This value is then used in the check-out phase, delegated to a 3rd-party service (checkout operation), before terminating the workflow.

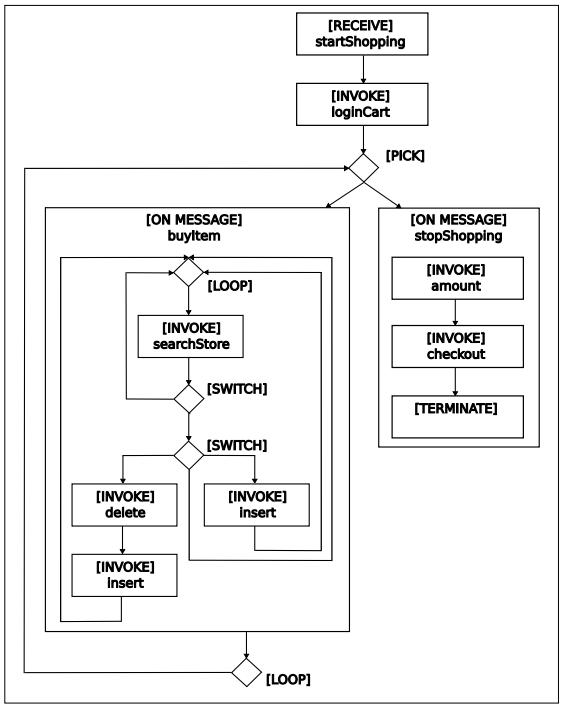

Figure 4: Advanced shopping session workflow

In the above description, we have omitted some details — such as local variables used to save the best price of an item and to refer to the most recently inserted item, and the `assign` activities related to them — to keep the example simple.

5.2 Monitoring Architecture

The architecture of our monitoring approach adopts an aspect-oriented programming approach [18]. Aspects are dynamically attached to the BPEL engine to monitor service compositions. By using AspectJ [17], we add monitoring

facilities to ActiveBPEL[1], an open source execution engine.

This solution allows the business logic to be kept separate from the monitoring logic, to achieve good code modularization. Monitoring is thus added to the business process both transparently, i.e. without modifying the structure of the process, and dynamically, i.e. by enabling/disabling, at run-time, the conformance checking of a service with respect to its specification, possibly depending on the performance requirements of the monitored process. This solution represents an alternative to proxy-based architectures, such as the one proposed in [6]. It removes the architectural bottleneck due to a central proxy. Furthermore, it eliminates the need to modify the original BPEL process to route its external service invocations to the proxy. Moreover, the overhead due to the monitoring instrumentation is kept very limited, as experimentation [11] reports only a 5% increment of the execution time of the workflow, with respect to the execution time on the non-instrumented BPEL engine.

Our solution extends the standard implementation of the ActiveBPEL engine with three main additional components, shown in Figure 5:

- *Main Interceptor*: it intercepts and modifies the execution of a process within the engine, at some pointcuts, using aspect-oriented programming;
- *Specifications registry*: it contains the specifications against which services are checked for conformance;
- *Monitor*: the actual conformance checker.

Main Interceptor plays a major role in our architecture: it supports the definition of monitor-related pointcuts and advices in AspectJ. We assume that pointcuts correspond to the activities in the BPEL process that interact with external services. We in fact assume the local workflow of the process to be correct and only the interactions with the external world to be potential causes of anomalies. Under this assumption, the `invoke` activity is the only one which triggers an interaction with the monitoring framework.
Indeed, we define a pointcut associated with the *Activity Execution* event, raised inside the BPEL execution engine: whenever a BPEL activity is about to be executed by the engine, if the activity is an `invoke`, the *Monitor* will be invoked, as will be described in the next section.

5.3 Integrating the evaluator into the monitoring framework

We assume that BPEL processes interact with conversational services that are described using an algebraic specification, as the one showed in Section 3.1. These specifications, together with a mapping of the operations of each conversational web service to operations of the corresponding algebraic sort, are supposed to be contained in a process deployment descriptor, in order to be processed by the execution engine when a new process is deployed.

The *Monitor* component shown in Figure 5 is actually a wrapper for an evaluator of algebraic specifications. The wrapper contains a *symbolic state generator* and an *interpreter*, as depicted in Figure 6.

The *symbolic state generator* keeps a machine-readable description of the state of each sort described by the algebraic specifications, for each process instance. The state is determined by the sequence of *constructor* and *transformer* operations performed on the corresponding external, conversational web service. This state is passed to the interpreter when an *observer* operation is invoked; the interpreter then returns a constant, resulting from the evaluation of the state, i.e. from the reduction of the term equivalent to the symbolic state representation.

The mapping between operations of a conversational web service and operations of the corresponding algebraic sort is stored in the *Specification registry* and is accessed by the *Main Interceptor*, whenever an `invoke` activity is about to be executed. Indeed, if the operation associated with an `invoke` activity is also associated with a monitored web service, the corresponding *constructor/transformer/observer* operation is invoked on the monitor. The invocation of a *constructor* or a *transformer* operation on the monitor lets the *symbolic state generator* update its internal representation of the state. On the other hand, invoking an *observer* operation triggers the monitor to make the interpreter evaluate the symbolic state. This evaluation returns a constant[2], which represents the expected value from the real invocation of the web service. The two values are then compared and, if a mismatch is found, an user-specified action (e.g. logging) is performed.

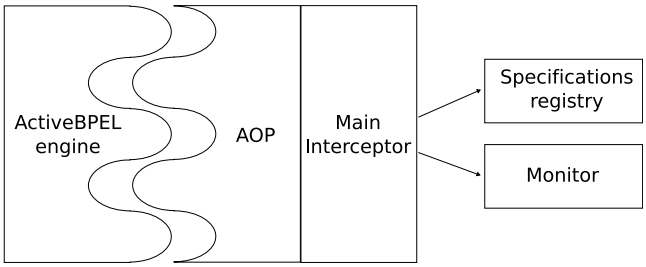

Figure 5: Monitoring embedded within the BPEL engine

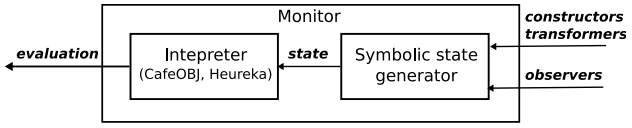

Figure 6: Architecture of the monitor component

6. RELATED WORK

The research described in this paper is a further step in the direction of the work of our group that focuses on the run-time support of service-oriented architectures, and in particular on run-time monitoring [6, 4]. The novelty of this work is the focus on conversational services.

To the best of our knowledge, this is the first research effort that uses algebraic specifications as the specification language for the run-time monitoring of conversational services. In the rest of this section we will survey existing approaches to monitoring functional and extra-functional properties of service compositions.

[1] `http://www.activebpel.org`

[2] We assume that provided specifications are complete, which guarantees the termination of the evaluation procedure.

Mahbub and Spanoudakis [22] propose a framework for the run-time verification of requirements of service-based software systems, implemented as BPEL compositions. Monitored properties are specified using event calculus and correspond to behavioral properties or assumptions on the behavior of the services used by the system. Monitoring is executed post-mortem, by exploiting techniques for integrity constraint checking in temporal deductive databases.

Robinson [27] presents a goal-driven methodology for monitoring requirements expressed using KAOS and temporal logic. Events to be monitored at run time are derived from goals, which represents requirements. Patterns of events observable at run time are then assigned to an agent for monitoring.

Pistore et al. [26] define a framework for planning the composition and monitoring the execution of BPEL Web services. The monitor component is synthesized by using planning techniques and observes interactions with the external services to detect whether the external partners behave inconsistently with the specified protocols.

Barbon et al. [3] propose (Run-time Monitoring Specification Language) to express temporal, boolean, time-related, and statistic properties of BPEL service compositions. These properties can then be monitored using a monitor engine run in parallel with the BPEL execution engine.

Li et al. [21] present a framework for monitoring the run-time interaction behavior of web services and validating the behavior against predefined interaction constraints. The monitor intercepts and analyses messages exchanged between a service and its clients, and validates the message sequence against the specified interaction constraints. Interaction constraints specify restrictions on both the occurrence of individual events and sequences of events.

Skene et al. [29] illustrate a model and an analysis technique for reasoning on the monitorability of systems, whose service-level agreements are expressed using SLAng [28].

The architecture of our run-time monitor has been inspired by Courbis and Finkelstein [10], who proposed an AOP-based solution to integrate monitoring, property checking, and possible reactions. The main difference with our solution is that they adopt a non-standard and proprietary workflow engine, while we refer to industry standards and publicly available tools.

Run-time monitoring can be applied also to more conventional, non service-oriented applications. In this domain, the work described by Nunes et al. in [24] proposes an approach for the run-time conformance checking of Java classes against algebraic specifications. The conformance checking problem is reduced to the run-time monitoring of contract-annotated classes, whose annotations are automatically derived from the specifications.

Orthogonal to our approach, that assumes the point of view of a BPEL process instance that monitors at run time its conversations with external services, are the ones based on model checking, like the work by Fu et al. [13], which performs a static analysis on the global sequence of messages exchanged by a set of composite web services.

Brenner et al. [8] discuss six different strategies of run-time testing of third-parties web services and how the identified strategies can be applied to the different kinds of web services. They propose a classification of web services, based on the state and on the per-client and pan-client properties, which is similar to the one we discussed in Section 2.

7. CONCLUSION

This paper introduces the notion of conversational services and proposes a technique to monitor their functional behavior. The expected behavior is specified formally via algebraic specifications, and the run-time behavior is then checked against them.

The check is performed by using an interpreter of the formal specification, which performs symbolic execution based on term rewriting. The results of the interpreter are compared with the values yielded by the monitored service. We also outlined the architecture of the tool we are developing, which integrates a workflow engine with a monitor of external service invocations.

We are currently experimenting with different interpreters of algebraic specifications. Our goal is to achieve a full integration of the interpreter into a comprehensive monitoring environment.

8. ACKNOWLEDGMENTS

This work has been partially supported by the IST EU project "PLASTIC" contract number 026955, and the Italian FIRB project "ART DECO".

9. REFERENCES

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services, Version 1.1, May 2003.

[2] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brueckner, editors. *Algebraic Foundations of Systems Specification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

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

[4] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Software*, 2007. submitted for publication.

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

[6] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL processes. In *ICSOC 2005: Proceedings of the 3rd International Conference on Service Oriented Computing*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.

[7] M. Bidoit and P. D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[8] D. Brenner, C. Atkinson, O. Hummel, and D. Stoll. Strategies for the run-time testing of third party web services. In *Proceedings of the 2007 IEEE International Conference on Service-Oriented Computing and Applications (IEEE SOCA 2007)*, pages 114–121, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[9] Y. Cheonand, G. Leavensand, M. Sitaramanand, and S. Edwards. Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience*, 35(6):583–599, 2005.

[10] C. Courbis and A. Finkelstein. Towards aspect weaving applications. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 69–77, New York, NY, USA, 2005. ACM Press.

[11] L. D'Ercole. Monitoraggio di processi di workflow mediante programmazione orientata agli aspetti. Master's thesis, Politecnico di Milano, 2006.

[12] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, 1990.

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

[14] K. Futatsugi and A. Nakagawa. An overview of CAFE specification environment. In *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM 1997)*, pages 170–182. IEEE Computer Society, 1997.

[15] J. A. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume IV: Data Structuring, pages 80–149. Prentice-Hall, 1978.

[16] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th international conference on Software engineering*, pages 449–558, Washington, DC, USA, 2004. IEEE Computer Society.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[19] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–117. Oxford University Press, 1992.

[20] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[21] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *ASWEC'06: Proceedings of the 17th Australian Software Engineering Conference*, pages 70–79. IEEE Computer Society, 2006.

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

[23] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[24] I. Nunes, A. Lopes, V. Vasconcelos, J. Abreu, and L. S. Reis. Checking the conformance of Java classes against algebraic specifications. In *8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260 of *Lecture Notes in Computer Science*, pages 494–513. Springer-Verlag, October 2006.

[25] M. Papazoglou and J.-J. Dubray. A survey of web service technologies. Technical Report DIT-04-058, Department of Information and Communication Technology, University of Trento, June 2004.

[26] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Artificial Intelligence: Methodology, Systems, and Applications, 11th International Conference, AIMSA 2004, Proceedings*, volume 3192 of *Lecture Notes in Computer Science*, pages 106–115. Springer, 2004.

[27] W. N. Robinson. Monitoring web service requirements. In *RE'03: Proceedings of the 11th IEEE International Conference on Requirements Engineering*, page 65, Washington, DC, USA, 2003. IEEE Computer Society.

[28] J. Skene, D. D. Lamanna, and W. Emmerich. Precise service level agreements. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 179–188, Washington, DC, USA, 2004. IEEE Computer Society.

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