Run-Time Monitoring of the Execution of Plans for Web Service Composition[*]

Fabio Barbon and **Paolo Traverso**
ITC-IRST
Via Sommarive 18, Povo
38050 Trento, Italy
{barbonfab,traverso}@itc.it

Marco Pistore and Michele Trainotti
DIT, University of Trento
Via Sommarive 14, Povo
38050 Trento, Italy
{pistore,trainotti}@dit.unitn.it

Abstract

While a lot of work has been done on the problem of the automated generation of plans that compose web services, the problem of monitoring their execution has still to be investigated. However, the run-time monitoring of web service executions is a compelling requirement, since it makes it possible to detect misbehaviors of external component services that are provided by third parties, and thus not fully under control. In this paper we propose a technique for the automatic generation of monitors as Java programs that check at run-time the execution of plans composing web services and detect violations to interaction protocols and service level agreements. The former correspond to unexpected changes in the planning domain, while the latter represent violations of assumptions that have been used to generate the composition plan.

Introduction

A lot of work has been done on the problem of planning for the automated composition of web services, addressing the problem of generating plans that, when executed, interact with available component services and compose them by satisfying some composition goal, see, e.g., (Blythe, Deelman, & Gil 2003; Wu *et al.* 2003; Dermott 1998; Sheshagiri, desJardins, & Finin 2003; McIlraith & Son 2002; Pistore, Traverso, & Bertoli 2005). In this context, the successful execution of the composition plan depends on the behavior of the external services, which may change in a very unpredictable and autonomous way. It is therefore of critical importance the monitoring of the execution of composition plans to detect violations to the agreed interaction protocols and service level agreements.

In this paper, we focus on web services described in BPEL (Andrews *et al.* 2003), a well-known standard language for web services. In this setting (see also (Pistore, Traverso, & Bertoli 2005; Pistore *et al.* 2005)), the planning domain is constructed from a set of *abstract BPEL specifications* available on the Web. An abstract (or interface) BPEL specification describes the interaction protocol with an external services that can be used as a component of the web service composition. Given such a planning domain, and given a composition goal, which describes the requirements for the desired composed service, the automated composition of web services can be formulated as the planning problem of generating plans that satisfy the composition goal by interacting with and composing the available component web services. Also the plans can be encoded as BPEL processes, but in this case we use *executable BPEL specifications*, which are executable programs that implement a business process by exchanging messages and orchestrating the external services.

The contribution of this paper is twofold. First, we propose an architecture for the execution and monitoring of BPEL processes. It is composed of two engines: a service execution engine and a monitor execution engine. The *service execution engine* is one of the available BPEL engines and corresponds to the plan execution environment. We have designed and implemented a *monitor execution engine*, which runs monitors implemented as Java programs, and we have integrated it with the BPEL engine, so that monitors are executed in parallel to the BPEL processes. The monitor engine can execute different kinds of monitors. *Domain Monitors* check at run-time whether the planning domain is the one that has been used by the planner at planning time. Since the planning domain is constructed from the abstract BPEL specifications of external services available on the Web, monitoring the planning domain corresponds to monitoring if some partner is not respecting the agreed flow of interaction. *Assumption Monitors* check instead the execution of the plan w.r.t. some assumption that has been used to generate the plan. Assumption monitors can be used to detect a violation of a functional requirement by one partner/process, as well as a global misbehavior where more than one process contribute to violate a required condition. We also allow for *Goal Monitors* that check whether composition (sub)goals are satisfied at run-time.

The second contribution of the paper is an efficient technique for the automatic generation of both domain and assumption/goal monitors, thus reducing the effort in their design and implementation. Domain monitors are generated by translating the available abstract BPEL processes of the component services into state transition systems, and by aggregating the states that cannot be observed by the moni-

[*]This work is partially funded by the MIUR-FIRB project RBAU01P5SS and by the MIUR-PRIN 2004 project "Advanced Artificial Intelligence Systems for Web Services".

tor into belief states (Bonet & Geffner 2000), i.e., sets of states that are indistinguishable for the monitor. The result is a transition system that evolves from belief states to belief states when the process receives/sends input/output messages, and can determine whether correct sequences of messages are exchanged. Assumption/goal monitors are generated from abstract BPEL processes and from an assumption/goal to be monitored at run-time, that in this paper we formalize in temporal logic. Both kinds of transition systems can be easily translated into Java code that is then executed by the monitor run-time environment.

Run-Time Execution and Monitoring

Figure 1 depicts the design-time and run-time environments in our framework. The *Design-Time Environment* has two main components, a Planner and a Monitor Generator. The *Planner* is used to automatically generate concrete BPEL processes according to the approach described in (Pistore, Traverso, & Bertoli 2005; Pistore *et al.* 2005). It takes in input the component services and a composition goal. The component services are abstract BPEL specifications that are available on the web, and they can be seen as the planing domain. The composition goal specifies requirements on the composed service. The planner can also take advantage of assumptions about the behavior of component services, in order to prune the search for a plan, e.g., according to the approach described in (Albore & Bertoli 2004).

The second component of the Design-Time Environment is the *Monitor Generator*, which is composed by a core component, the *Domain Monitor Generator* and by the *Assumption Monitor Generator*, which is built upon the generator of domain monitors. The algorithms run by these modules are described in the next section.

The *Run-Time Execution/Monitoring Environment* runs in parallel concrete BPEL processes (the plans generated at design time) and Java monitors (generated by the monitor generator). In our approach, monitors observe BPEL process behaviors by intercepting the input/output messages that are received/sent by the processes, and signal some misbehavior or, more in general, some situation or event of interest. In Figure 1, the light components on the left-hand side constitute the BPEL process execution environment, that in our framework corresponds to the plan execution environment. The monitor run-time environment is instead composed of the darker components on the right-hand side.

For the *BPEL process execution environment*, we have chosen a standard engine for executing BPEL processes, namely Active BPEL (http://www.activebpel.org), which is available as open source and implements a modular architecture that is easy to extend. From a high level point of view, the Active BPEL run-time environment can be seen as composed of four parts (see the light components of Figure 1). A *Process Inventory* contains all the BPEL processes deployed on the engine. A set of *Process Instances* consists of the instances of BPEL processes that are currently in execution. The *BPEL Engine* is the most complex part of the run-time environment, and consists of different modules which are responsible of the different aspects of the execution of the BPEL processes. In particular, the Process Manager creates

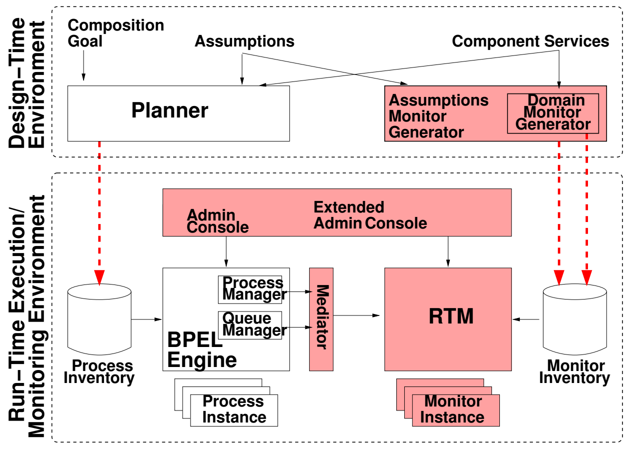

Figure 1: Design-Time and Run-Time Execution and Monitoring Environments

- init(): init method, executed when an instance of the monitor is created
- evolve(BpelMsg message): handles a message, updating the state of the monitor instance
- terminate(): handles the notification of a process termination event
- isValid(): returns true if the monitor instance is in a valid state (i.e., no misbehavior has been detected)
- getErrorString(): returns an error string if the monitor instance is in an invalid state
- getProcessName(): returns the name of the BPEL process associated to the monitor
- getPropertyName(): returns the (short) property name of the monitor
- getPropertyDescription(): returns the description of the property checked by the monitor

Figure 2: Methods of a Monitor Java Class

and terminates process instances, and the Queue Manager is responsible of dispatching incoming and outgoing messages. The *Admin Console* provides web pages for checking and controlling the status of the engine and of the process instances.

We extended Active BPEL with a *Run-Time Monitoring Environment*, which is composed of four parts (see Figure 1). The *Monitor Inventory* and the *Monitor Instances* are the counterparts of the corresponding components of the BPEL engine: the former contains all the monitor classes deployed in the engine, while the later is the set of instances of these classes that are currently in execution. Each monitor class is associated to a specific BPEL process, while each monitor instance is associated to a specific process instance. Each monitor class is a Java class that implements the methods described in Figure 2. The *Run-Time Monitor (RTM)* is responsible to support the life-cycle (creation and termination) and the evolution of the monitor instances. The *Media-*

tor allows the RTM to interact with the Queue Manager and the Process Logger of the BPEL engine and to intercept input/output messages as well as other relevant events such as the creation and termination of process instances. The *Extended Admin Console* is an extension of the Active BPEL Admin Console that presents, along with other information on the BPEL processes, the information on the status of the corresponding monitors.

The monitor life-cycle is influenced by three relevant events: the process instance creation, the input/output of messages, and the termination of the process instance. When the RTM receives a message for the Mediator, it tries to find a match with the already instantiated monitors. If a match is found, the message is dispatched to all the matching monitor instances through method "evolve". If no match is found, then a new process instance has been created in the BPEL engine, and hence a set of monitor instances specific for that process instance is created by the RTM and initialized through the method "init". For each message, the Mediator provides also information on the process instance receiving/sending the message, as well as on the BPEL process corresponding to the instance. The information on the BPEL process is used to select the relevant set of monitors to be instantiated for that process. The process termination is dispatched, through the invocation of method "terminate", to all the monitor instances associated to the process instance.

Automatic Generation of Monitors

We implement monitors as pieces of Java code that are synthesized automatically from the abstract BPEL specifications of the external partners. Before performing the actual monitor generation, we automatically translate abstract BPEL specifications into *state transition system*s (STS).

Definition 1 (State transition system (STS)) *A* state transition system $\mathcal{D}$ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{X}, \mathcal{T} \rangle$ *where:*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{X}$ *is the finite set of messages;*
- $\mathcal{T} \subseteq \mathcal{S} \times (\mathcal{X} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation.*

A state s *is said to be* final *if there is no transition starting from* s *(i.e.* $\forall x \in \mathcal{X}, \forall s' \in \mathcal{S}.(s, x, s') \notin \mathcal{T}$*).*

STSs describe dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of either sending/receiving messages, or performing a special transition τ, called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the process can evolve without producing any message in output, and independently from the reception of messages in input. A *transition relation* describes how the state can evolve on the basis of messages, or of the internal action τ. The translation from BPEL to STS has been described in (Pistore, Traverso, & Bertoli 2005; Pistore *et al.* 2005).

Domain monitors. Monitors can only observe messages that are exchanged among processes. As a consequence, they cannot know exactly the internal state reached by the evolution of a monitored external service. Non-observable behaviors of a service, are modeled by τ-transitions, i.e. transitions from state to state that don't have any associated input/output. From the point of view of the monitor, this kind of evolutions of external services cannot be observed, and states involved in such transitions are indistinguishable. Such sets of states are called *belief states*, or simply *beliefs* (Bonet & Geffner 2000).

The generation of a domain monitor for an external abstract BPEL process is based on the idea of beliefs and belief evolutions. The domain monitor generation algorithm incrementally generates the set of beliefs by grouping together indistinguishable states of the STS. These beliefs are linked together with (non τ) transitions that correspond to messages sent to, or received from, the external service. The Java code implementing the domain monitor can be generated from this belief-level description of an external service. More precisely, the belief states are exploited to trace the current status of the evolution of the monitored BPEL process, and the transitions among beliefs are used to update let the status of the monitor whenever a message is received. At run-time, whenever a message is received that does not match the existing transitions among beliefs, an error is issued by the monitor.

Assumption monitors. The algorithm for the generation of assumption monitors takes in input the abstract BPEL process of one of the external services plus an assumption to be monitored. We express run-time properties in KPLTL, a variant of Linear Time Logic *LTL* (Emerson 1990) that is restricted to past temporal operators and interprets basic proposition in a special way. More precisely, basic propositions are of the type $\mathbf{K}\,p$, where p is a property on the states of the STS Σ corresponding to the abstract BPEL of the external process to be monitored. Intuitively, $\mathbf{K}\,p$ holds in a belief B if we know that p will be true while the external service is within B.

Definition 2 (KPLTL Properties) *Let* $\mathcal{P}rop$ *be a property set and* $p \in \mathcal{P}rop$*.* KPLTL *properties are inductively defined as follows:*

$$\phi ::= \mathrm{true} \mid \boldsymbol{K} p \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{Y}\,\phi \mid \mathrm{O}\,\phi \mid \phi\,\mathrm{S}\,\phi \mid \mathrm{H}\,\phi.$$

We already explained the meaning of $\mathbf{K}\,p$. Intuitively, the temporal operators above can be read as:

- $\mathrm{Y}\,\phi$ means "ϕ was true in the previous belief state";
- $\mathrm{O}\,\phi$ means "ϕ was true (at least) once in the past";
- $\mathrm{H}\,\phi$ means "ϕ was true always in the past";
- $\phi_1 \mathrm{S}\, \phi_2$ means "ϕ_1 has been true since ϕ_2".

A KPLTL property P to be monitored can be translated into a Java class that traces the evolution of the validity of the property. The generation of the Java code is based on standard techniques for evaluating LTL formulas (Emerson 1990). The evolution of the assumption monitor is based on the evolution of the basic propositions in the KPLTL formula. The evolution of these propositions can be computed by tracing the evolution of the status of the corresponding external service, in a similar way to what is done for domain monitors.

Goal monitors. Goal monitors usually correspond to choreographic properties, i.e., they check properties that depend on the behavior of possibly many different components. This case is similar to that of the assumption monitor, but the KPLTL property contains basic propositions belonging to different external partners of the BPEL process to be monitored. The construction of the monitor in this case follows the same lines of that described above for assumption monitors, with the only difference that we have to trace the belief level evolution of more than one external BPEL process.

Conclusions and Related Work

In this paper we address the problem of monitoring the execution of plans composing web services described in the standard BPEL language. We implement a run-time environment for monitoring plan executions that is integrated with a BPEL engine, and we propose a technique for automatically generating different kinds of monitors. Domain monitors can check at run-time whether a partner does not respect the published protocol by detecting changes in the planning domain, while assumption monitors can check whether a partner violates some basic assumptions that have been agreed and that can be used to generate the composition plan. Finally, goal monitors can check whether (part of) the composition goal is satisfied at run-time. As far as we know, this is the first proposal that addresses the problem of run-time checking the execution of web services described as BPEL processes in a framework for automated planning.

Considering the problem of monitoring BPEL processes, an obvious alternative to our approach would be to code manually monitors in BPEL. However, this approach has several drawbacks. A main disadvantage is that it does not allow for implementing monitors that capture misbehaviors caused by BPEL execution engines. For instance, BPEL engines can exchange the order of messages. A process may receive a message that it is not able to accept at the moment, which can be followed by another message that can instead be consumed. The first message can be consumed later on by the process, or may never be consumed. This mechanism is known as “message overpass”. Our monitors can capture message overpasses, while BPEL monitors cannot, since they are subject to the same execution conditions of the processes they monitor. BPELJ is a language that allows the programmer to embed monitors as Java code into BPEL processes. However, both BPEL and BPELJ monitors do not allow for a clear separation of the business logic from the monitor functionality. Moreover, manually programming monitors in BPEL or BPELJ is time consuming, error prone, and requires programming effort.

In (Baresi, Ghezzi, & Guinea 2004), monitors are specified as assertions that annotate the BPEL code. Annotated BPEL processes are then automatically translated to “monitored processes”, i.e., BPEL processes that interleave the business processes with the monitor functionalities. This approach allows for monitoring time-outs, runtime errors, as well as functional properties. The advantage is that monitors are themselves services implemented in BPEL, and can run on standard BPEL engines. However, they cannot capture misbehaviors of the BPEL engine and do not clearly separate the business logic from the monitor functionality. A difference with our approach is also in the expressiveness of the monitor specifications. We automatically generate monitors that check properties expressed in temporal logic. These monitors can easily check properties that depend on the whole history of the execution path, and that would be very hard to express as assertions.

In the future, we are going to apply and extend our techniques to the case of semantic web services, and to enrich the run-time environment with techniques for the analysis of data generated by monitors. We also plan to extend our approach to generate and execute monitors that can check the internal state variables of their own service, and to provide techniques for the automated service failure-handling, repairing and adaptation triggered by information provided by monitors.

References

Albore, A., and Bertoli, P. 2004. Generating Safe Assumption-Based Plans for Partially Observable, Nondeterministic Domains. In *Proc. AAAI’04*.

Andrews, T.; Curbera, F.; Dolakia, H.; Goland, J.; Klein, J.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; and Weeravarana, S. 2003. Business Process Execution Language for Web Services (version 1.1).

Baresi, L.; Ghezzi, C.; and Guinea, S. 2004. Smart Monitors for Composed Services. In *Proc. ICSOC’04*.

Blythe, J.; Deelman, E.; and Gil, Y. 2003. Planning for Workflow Construction and Maintenance on the Grid. In *Proc. of ICAPS’03 Workshop on Planning for Web Services*.

Bonet, B., and Geffner, H. 2000. Planning with Incomplete Information as Heuristic Search in Belief Space. In *Proc. AIPS’00*.

Dermott, D. M. 1998. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University. CVC Report 98-003.

Emerson, E. A. 1990. Temporal and modal logic. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier.

McIlraith, S., and Son, S. 2002. Adapting Golog for composition of semantic web Services. In *Proc. KR’02*.

Pistore, M.; Marconi, A.; Bertoli, P.; and Traverso, P. 2005. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI’05*.

Pistore, M.; Traverso, P.; and Bertoli, P. 2005. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS’05*.

Sheshagiri, M.; desJardins, M.; and Finin, T. 2003. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS’03*.

Wu, D.; Parsia, B.; Sirin, E.; Hendler, J.; and Nau, D. 2003. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC’03*.