9

Run-Time Monitoring in Service-Oriented Architectures

Carlo Ghezzi and Sam Guinea

Dipartimento di Elettronica e Informazione—Politecnico di Milano via Ponzio 34/5 – 20133 Milano (Italy) ghezzi|guinea@elet.polimi.it

Abstract. Modern software architectures are increasingly dynamic. Among them, Service-Oriented Architectures (SOAs) are becoming a dominant paradigm. SOAs allow components to be exported as services for external use. Service descriptions (which include functional and non-functional properties) are published by service providers and are later discovered by potential users. Service discovery is based on matching the published service descriptions with the required service specifications provided by the user. Once an external service is discovered, it may be bound and invoked remotely. New services may also be created by composing existing services.

To achieve full flexibility, the binding between a service request and a service provision may be set dynamically at run-time. Dynamic binding and decentralized management of external services by independent authorities, however, challenge our ability to perform verification and validation (V&V). Traditional V&V is a pre-deployment activity. In the new setting it extends to run-time and requires continuous monitoring of functional and non-functional attributes.

This chapter investigates continuous monitoring of SOAs, with particular emphasis on web services. It provides a classification scheme that can help understanding the different monitoring approaches a system designer can choose. It also analyzes the running example and discusses some of the functional and non-functional aspects one might be interested in monitoring in its context. The chapter then presents a short survey of the most important ongoing research in this field and concludes by discussing future research directions.

9.1 Introduction

Traditionally, software systems had a pre-defined, static, monolithic, and centralized architecture. This was largely due to the technology available at the time and to the need of making the resulting system more easily manageable and controllable. All the parts that composed a large application were under the control of a single organization, which was ultimately responsible for their design, development, verification, and deployment procedures. Software architectures have been constantly evolving toward increasing degrees of dynamism and decentralization, from statically bound compositions to dynamically composed federations of already deployed and running components.

To describe this evolution, and the requirements that drove it, it is important to provide an informal definition of some terms and concepts that will be used throughout this chapter. The term *component* denotes an identifiable piece of code implementing some useful function, which may become part of a larger system. The term *service* denotes a component that is deployed and run separately. *Composition* is the way a whole system is made up, by binding components together. The arrangement and relation between the bound components defines the system's *architecture*.

The evolution of software architectures has been dictated by the need for applications to evolve continuously as the environment in which they are embedded evolves. Continuous and rapid changes are requested from the real world, and reactions to change requests must be extremely fast. The traditional strategy to respond to change requests—which implies switching to off-line mode and re-designing, re-implementing, and re-deploying (parts of) the application—does not work in this new context. Rather, changes should be made dynamically at run-time.

Requirements for these new features arise in a large variety of application fields, from in-the-large web-based information systems to in-the-small embedded applications supporting ambient intelligence. Information systems need to evolve continuously to support dynamic federations of business organizations interacting through the web. In this setting, each organization exposes internal functions as new services that other members of the federation may use, and new bindings may be established between a service request and a service provision as the opportunistic goals of the federation change over time. In ambient intelligence settings, the requirements for dynamically composing services derive from the goal to support context-aware behaviors. In most practical cases, context is defined by the physical location, which may change because of mobility. For example, a print command issued by a mobile device should print a document from a closest printer, dynamically discovered in the surrounding physical environment. The concept of context, however, is more general. Suitable sensors, in general, may provide context information. For example, depending on where the service requester is located and on light conditions, the command to illuminate a room might imply sending signals to actuators to switch the light on or to open the window shades. Both cases are characterized by a novel distinctive feature: not only do the bindings among components of the application change dynamically, as the application is running, but the components that are available for composition do so as well.

The concept of service is the cornerstone of *service-oriented architectures* (SOAs). SOAs have been emerging as a most promising architectural paradigm which provides support to dynamically evolving software architectures, where both the components and their bindings may change dynamically. The style is characterized by the following features:

- Publication: Through publication, a service description is made available by a service provider in a standardized manner that potential clients may understand.
- Discovery: A service is searched based on the requested features it should offer, and by matching the request with the available published descriptions.
- Binding: Based on the search results, a binding is established between a service request and a service offer.

Since publication and discovery may be performed at run-time, bindings may also be established and modified dynamically. This high degree of dynamism, while providing great benefits in terms of flexibility, has a severe impact on the system's correctness and on the way verification can be performed. Traditionally, verification of correctness is performed statically, based on the known components that compose the application. After the application is deployed, no further verification is needed (nor possible). In the case of SOAs, however, an application is made out of parts (services) that are deployed and run independently, and may change unpredictably after deployment. Thus, correctness cannot be ascertained statically, but rather requires continuous verification that the service delivered complies with the request. In the case where serious problems are discovered, suitable recovery reactions at the architectural level should be put in place.

Many stakeholders are involved in service-oriented applications: clients, providers, and third-parties. Typically, they have different needs and different requirements. They have different business goals, and tend to state what they expect from a system differently, both in terms of functionalities and in terms of quality of service. Consequently, run-time monitoring has different objectives for each of them. In this chapter, we focus on the role of a service requester. This may be an end-user who acts as client, or a service provider who acts as a third-party by composing a new service out of existing services. Run-time monitoring, in this case, takes the requester's viewpoint: the service should deliver what it promised and should match the requester's expectations. If it does not, the system should take or initiate appropriate subsequent reactions, such as notifications, remedy, compensation, etc. This work focuses on monitoring; a study of reaction strategies falls beyond its scope. Also, we do not focus on the process that elicits business goals and derives run-time monitoring goals. Rather, we assume that such process is in place, and focus our attention on the monitoring process itself. Although our main focus is on requester-side monitoring, provider-side monitoring is also quite relevant. In this case, the objective is to monitor the quality of the delivered service and drive possible run-time optimizations, such as dynamic resource allocation in SOAs implemented on grid architectures [10]. We will only briefly touch on this point, which has received considerable attention in a number of industrial research approaches.

Although this chapter concentrates on service composition providers and their needs, most of what we present is general enough and easily extendible to cover the needs of different stakeholders [6]. Most of what we say here also holds for SOAs in general. However, we focus on web services and discuss solutions that hold in the case of the available web service technology.

The chapter is organized as follows. Since many different monitoring approaches exist, and since they all behave quite differently (i.e., each with its own strengths and weaknesses), Sect. 9.2 starts off by providing the reader with an overview of some key aspects that can be used to better understand and classify them. Section 9.3 continues by discussing the example introduced in the initial chapters of this book in the context of run-time monitoring. Section 9.4 introduces our own assertion-based approach to monitoring called "Dynamo," and its monitoring definition language called WSCoL (Web Service Constraint Language). Section 9.5 compares some of the other most prominent research and industrial monitoring approaches, while Sect. 9.6 concludes the chapter.

9.2 Run-Time Monitoring

The need to monitor SOAs at run-time has inspired a large number of research projects, both academic and industrial. The differences between these research proposals are manifold, and quite evident after an accurate analysis. This has led to an unfortunate situation in which the term "monitoring" is commonly used, but with many possible interpretations. Although their main goal—discovering potential critical problems in an executing system—remains the same, there are differences that concern important aspects, such as the goals of monitoring, the stakeholders who might be interested in them, the potential problems one might try to detect, etc.

A thorough understanding of these aspects that characterize SOA monitoring is important in order to classify the different monitoring approaches available in the literature, to evaluate them, and to choose the monitoring approach most suitable for the problem at hand.

Our presentation will concentrate on the following most significant aspects: the type of properties that can be monitored, the type of collaboration paradigm with which they can be coupled, the methods they use to collect data, their degree of invasiveness, and their timeliness in discovering anomalies. Even though the classification items are presented separately, they are heavily intertwined, and the choices made in the context of one dimension may influence the others. For example, the choice to monitor certain functional properties impacts the way run-time data are collected, which in turn has an impact on the degree of invasiveness of the approach.

9.2.1 Types of Properties

Monitoring approaches can be tailored to the verification of two main families of properties: functional and non-functional (or quality-of-service related) properties. When monitoring the former, we are interested in verifying whether a given service delivers the function we expect from it. This obviously requires that some specification of the expected behavior be available. Since the invocation of a service can be seen as a black box that, given certain inputs, produces certain outputs, most monitoring approaches tend to rely on procedural specifications expressed in terms of pre- and post-conditions.[1] The monitoring approaches, therefore, typically consist of mechanisms that produce suitable oracles for the service being monitored.

Since we focus on web services, most descriptions—such as those given using the WSDL standard[7]—only specify the syntactical aspects involved in invoking a web service. A number of special-purpose specification languages have been proposed to address this problem. Some of the proposals originated in the field of software engineering, such as our own language WSCoL, were built on the legacy of Design by Contract [22,23] and assertion languages for standard programming languages such as Anna [20] or APP [30]. Others originated in the field of Semantic Web, such as the current specification language candidates being considered in the context of OWL-S [29]. Their principal candidate is the Semantic Web Rule Language (SWRL) [11].

Regarding non-functional or "quality of service" related properties, monitoring focuses on those that can be measured in a quantitative way. This leaves out a number of relevant properties (such as usability or scalability) that are either qualitative (and subjective), or for which quantitative metrics do not exist. Some of the most common non-functional properties are as follows:

- *Availability*, which measures the readiness of a web service to be used by its clients. It also considers aspects such as how long a given service remains unavailable after occurrence of a failure.
- *Accessibility*, which considers the capability of the service provider's infrastructure to instantiate a service and guarantee provisioning even in the case of heavy traffic. In some way, it measures scalability of the provider's infrastructure.
- *Performance*, which is usually measured in terms of throughput and latency. The former defines the number of requests that can be addressed in a given time-frame. The latter defines the round-trip time of a request and its response.
- *Security*, which is perceived as an extremely important aspect due to the open environment (the Internet) in which service interactions occur. Its most important goals are to guarantee confidentiality, non-repudiation, and encryption.

[1] This works fairly easily for stateless services, which behave like functions. Stateful services require a way to model the hidden abstract state as well.

- *Reliability*, which measures the capability of a service to guarantee the promised or negotiated qualities of service.

9.2.2 Collaboration Paradigms

The true advantages of service-oriented architectures become evident when remote services are used cooperatively to achieve some overall business goals. Different existing collaboration paradigms exist, each presenting its own unique aspects. This leads to monitoring approaches that are tailored toward a specific style of collaboration.

Collaboration paradigms differ in the degree of coupling among the participating services and the degree with which business responsibility is distributed amongst them. A typical distinction is between *orchestration* and *choreography*. In the case of orchestration-based approaches, a single party is responsible for the correct evolution of the business process. The current state of the art in orchestration-based approaches is the Business Process Execution Language for Web Services (BPEL)[16], which became a de-facto standard in the last few years. BPEL is an executable workflow language that is processed by a suitable engine. Most current implementations are based on a centralized workflow engine (e.g., ActiveBPEL), although distributed BPEL engines have also been proposed [28]. The workflow engine is responsible for correctly executing the business process and invoking the required external services, as specified in the process. As we mentioned, the binding between an invocation of an external service and the actual service exported by a service provider can change dynamically at run-time. The monitoring approaches that are tailored to such a scenario are typically concerned with checking whether the external services behave as promised, and expected. That is, one needs to check that external services—when invoked—satisfy certain functional or non-functional requirements that allow the business process to achieve its goals.

At the other end of the spectrum, it is possible to envision paradigms in which services are individually responsible for the overall coordination effort and the correctness of different parts of the business process. This is the case of choreography-based approaches to collaboration. The current state of the art is the Web Service Choreography Description Language (WSCDL)[17]. WSCDL is a non-executable specification language that describes the messages exchanged among the different parties during a collaboration. It defines both the message formats and the order in which they must be sent. In a choreography, no central party is responsible for guiding the collaboration, but rather each partner must be (1) aware of the defined business-logic, (2) capable of correlating messages, and (3) capable of performing its role in the process. In such scenarios, it is important to monitor the functional and non-functional qualities of service invocations. It is also necessary to monitor the evolution of the business logic, by checking that the required messages are sent and received in the specified order. To achieve this goal, the monitor must be provided with a behavioral specification of the process being carried out, which can be derived from the specification of a choreography.

9.2.3 Collecting Monitoring Data

Data can be collected from different sources. At least four very prominent cases can be identified.

1. Collection of process state data: In orchestration-based systems, data may be collected through appropriate probes that are placed throughout the process. The properties that can be checked are those that predicate on process states. To make the approach less invasive, it is possible to limit the check of process states to the points where the workflow process interacts with the outside world, by capturing the data that flow in and out. In a centralized execution environment, this can be achieved quite simply by intercepting the incoming and outgoing messages. In a choreography, the probes must be set up in a distributed fashion.
2. Collection of data at the SOAP message level: In service collaborations, data can also be collected at the message level. This can be achieved through the use of appropriate probes that intercept all the SOAP messages entering or leaving the system on which a service is deployed. This is especially useful when we need to check the properties of a SOAP message header, or of a message's payload.
3. Collection of external data: Some monitoring approaches require additional data that must be collected externally. This happens when a certain property to monitor predicates on data that does not belong to the current business process. For example, it might be necessary to verify if the value of the interest rate, returned by a banking web service, satisfies the correctness requirement that it should not exceed a threshold defined by the National Bank. Since the threshold may change dynamically, it must be retrieved at run-time by invoking an appropriate external data source. Obviously, the monitoring framework must be aware of the existence of this data source in order to verify such a constraint.
4. Collection of low level events: Some monitoring approaches rely on data collection that is achieved at a lower level of abstraction, such as at the execution engine level. The events generated by the execution are collected and logged for on-the-fly or later analysis. For example, the ActiveBPEL execution engine[1] associates special states to the BPEL activities being executed. An invoke activity can be in an "inactive" state, a "ready to execute" state, or an "execute" state, and produces an event each time when there is a valid state transition. Data collection can be wired into the execution engine to capture these transitions, allowing analysis to predicate on the order in which they occur, when they occur, etc.

9.2.4 Degrees of Invasiveness

Existing monitoring approaches differ in the degree of invasiveness with respect to specification of the business logic and its execution.

Regarding specification—a typical design-time activity—in certain approaches the definition of the business logic and the monitoring activities are highly intertwined (e.g., through the use of annotations in the process definition). Other approaches keep the specification of the monitoring logic entirely separate from the business logic, thus encouraging a "separation of concerns" which allows designers to reason separately on the two problems.

Regarding execution, it is possible to distinguish between approaches in which the execution of the business and of the monitoring logic are highly intertwined, and approaches in which they execute independently.

An example of a highly invasive approach to monitoring is the use of pre- and post-conditions. Since they require process execution be blocked when the properties are checked, they have an adverse effect on performance. On the other hand, approaches that have a low degree of invasiveness usually take place externally to the process.

9.2.5 Timeliness in Discovering Undesirable Situations

A timely monitor detects an anomalous situation as soon as the data indicating the anomaly have been collected. In general, the distance between the two time points denotes the degree of timeliness of the monitoring approach, which can vary from early to late detection.

At one end of the spectrum, we can find approaches that adopt highly intrusive techniques, which aim at discovering erroneous situations as early as possible. These should be used in situations that are critical for the business process, such as cases in which we need to be sure that a message is transmitted in encrypted form, using the appropriate encryption algorithms. A possible way to ensure high degrees of timeliness is to express the properties in terms of assertions (e.g., pre- and post-conditions on service invocations) that block the business process while the run-time checking is being performed.

At the other end of the spectrum, we can find approaches that allow designers to do post-mortem analysis to discover erroneous situations. These approaches can be used to plan changes that may affect future executions, bindings, or versions of a business process. A possible implementation may be based on logging events and execution states for later analysis.

A special mention should also go to approaches that perform proactive monitoring. Thanks to the data collected both during previous executions of the business process and on-the-fly, these approaches try to identify situations in which it is progressively more and more likely that global process qualities (e.g., the overall response time) will not be maintained. However, since erroneous behaviors—especially those regarding non-functional qualities—can be transient, pro-active monitoring may lead to situations in which the monitoring signals a problem that actually does not manifest itself.

9.2.6 Abstraction Level of the Monitoring Language

The language used to specify the monitor depends on the expected end-user. Highly structured approaches provide a low-abstraction level and are heavily influenced by aspects such as the collaboration paradigm being used and its data formats. These must be considered tools for the designers responsible for delivering high quality and dependable processes.

On the other hand, it is also possible to envision approaches in which higher abstraction levels are used. These hide the intricacies of the business process' collaboration paradigm, and allow non-technical end-users to define functional and/or non-functional properties they consider important for their applications.

9.2.7 Other Aspects

Many other classification dimensions can be considered when analyzing existing monitoring approaches. An example is the degree of expressiveness provided by the monitoring specification language. Depending on the nature of the properties the approach is capable of verifying, we can find languages that require a more theoretical background, such as first-order logics or temporal logics, or that are closer to a more typical object-oriented system designer's background, such as OCL.

Another possible classification dimension is the degree of automation in the derivation of the monitoring directives. In fact, it is possible to envision approaches that require the designer to manually define the properties to be checked, and approaches in which the properties are automatically derived by the system, by formally reasoning on the requirements.

Monitoring approaches can also be classified based on the validation techniques they adopt. Some examples of techniques for verifying properties are assertion checking, trace analysis, model-checking, etc.

The approaches can also be classified based on their degree of adoptability. Some approaches, thanks to the adoption of standards, do not depend on the run-time infrastructure chosen by a service composition provider. Others, instead, from a technological and implementation standpoint, are tied to a certain proprietary run-time environment, and therefore cannot be easily configured to interoperate and integrate with different ones.

Finally, monitoring approaches can be classified based on the nature of their support infrastructure. It is possible to conceive monitoring infrastructures as centralized components that overlook service execution, or as distributed components that collaborate to check the functional and/or non-functional properties we need.

9.3 Case Study

The case study introduced in the initial chapters of this book provides informal common grounds for reasoning on the different facets of web services. To dwell deeper in the real intricacies of the monitoring problem, we need to further detail some key aspects of the proposed scenarios, such as their functional and non-functional requirements, the required collaboration paradigms, the underlying architecture, and its binding policies.

9.3.1 Functional Correctness of the Holiday Location-Finder Web Service

The process starts with John looking for suitable locations for his get-away weekend, locations that must satisfy certain requirements (they must be close to where he lives, by a lake, near the mountains, etc). Using his office computer, John interacts with a fairly simple orchestrated process that guides him in finding the location, booking the rooms in a hotel, etc.

Figure 9.1 specifies the interface of the holiday location-finder web service, using a stereotyped UML class diagram to avoid the low-level details of a WSDL XML interface. In this abstraction, web services are seen as boxes that only provide public methods. The input and output parameters for these methods are described through "dataType" stereotypes, which only contain public attributes.

Given a request that specifies the departure location (i.e., a location name and GPS coordinates), a maximum traveling distance the client is willing to go, and an articulate description (the format of which is omitted for simplicity)

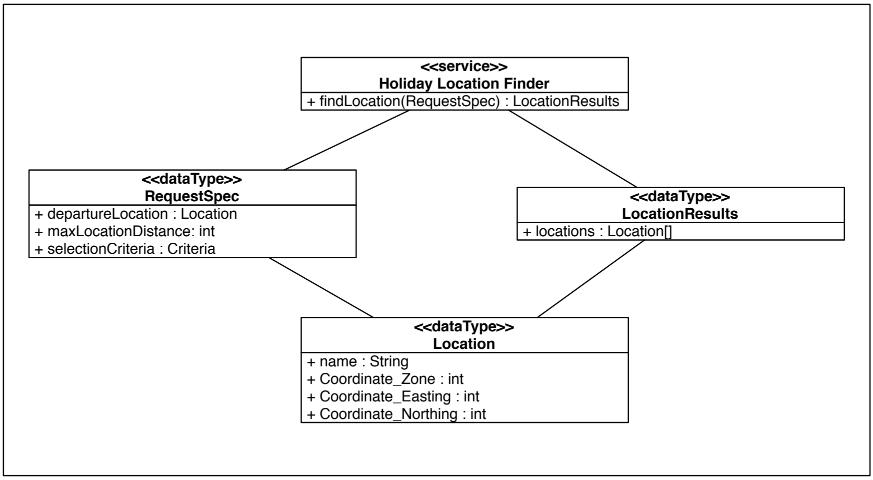
<<service>>
Holiday Location Finder
+ findLocation(RequestSpec) : LocationResults
<<dataType>>
RequestSpec
+ departureLocation : Location
+ maxLocationDistance: int
+ selectionCriteria : Criteria
<<dataType>>
LocationResults
+ locations : Location[]
<<dataType>>
Location
+ name : String
+ Coordinate_Zone : int
+ Coordinate_Easting : int
+ Coordinate_Northing : int

Fig. 9.1. The holiday location-finder web service

of key interests, such as proximity to a lake, mountains, etc., the web service responds with an array of possible holiday locations.

We assume that the external location-provider web service invoked by the workflow is used by John under a temporary trial license. Before subscribing a contract with the provider, John wishes to verify that the service delivers what it promises, and therefore he turns the monitor on. The monitor checks whether the returned locations satisfy all the user-defined selection criteria. To simplify our example, we will concentrate on verifying whether the locations are within the maximum distance specified in the request.

John decides to adopt an invasive monitoring approach, in which post-conditions block the process while executing run-time checks. The post-condition that checks the validity of the service can be expressed as in Fig. 9.2.[2] However, an invasive monitoring approach based on a blocking post-condition is not the only possible solution. John could have instead adopted a solution that checks the property in a less timely fashion, using a less intrusive approach, and with a lower impact on the overall performance of the process.

9.3.2 Functional Correctness of the Map Web Service

The example scenario states that John and his wife decide to travel by car. The car is equipped with a haptic device to communicate with remote services for entertainment reasons (e.g., purchase a multimedia stream), or for gathering useful information from the environment. John decides to use his device to obtain a map illustrating how to reach the vacation resort. The device can show only certain image formats with a given resolution. Therefore, it is important that the map returned by the external service satisfies both requirements. Suppose that John's workflow has a pre-installed binding to a free-of-charge external service that does not always guarantee fulfillment of the requirement. It may, in fact, sometimes deliver maps whose format or resolution are invalid for the haptic device. The monitor is therefore turned on,

For all the returned locations l,
(RequestSpec.departureLocation.Coordinate_Easting - l.Coordinate_Easting)^2 +
(RequestSpec.departureLocation.Coordinate_Northing - l.Coordinate_Northing)^2 <=
RequestSpec.maxLocationDistance ^{2}

Fig. 9.2. A functional property

[2] We assume that the monitoring language allows properties to be specified using universal quantifiers over the elements of a certain data set.

to allow for delivery of unacceptable maps to be trapped. A suitable reaction to a detected anomaly might consist of switching to another service provider who provides maps under payment.

In order to discover an image's format and resolution, special-purpose tools are needed. Since the delivery environment (i.e., the BPEL execution engine) does not possess the necessary tools for manipulating and/or aggregating the monitoring data, the monitor itself is responsible for retrieving the data it needs. Furthermore, John decides to adopt an invasive but timely monitoring policy, which prevents the haptic device from using a non-compliant image. As a non-compliant image is detected, the system starts a reaction strategy which tries to find a suitable substitute for the current map service. To achieve this goal, John decides to define the property in terms of a process-blocking post-condition.

9.3.3 Monitoring Security

In the example, John uses a service—provided by his bank—to pay for his reservations. John expects his banking services to provide standard encryption strategies and technologies capable of ensuring "safe transactions." Safe transactions prevent eavesdropping, message tampering, fake messages, etc. The main standards proposed for tackling these problems are WS-Security [2] and WS-Trust [15]. The former supports end-to-end security issues, such as origin authentication, integrity, and confidentiality, while the latter supports the creation of trust relationships between different parties.

In order to ensure end-to-end security, the monitor must have access to the SOAP messages flowing in and out of the execution engine. In fact, it is necessary to verify whether the messages carry the appropriate signature elements, and whether certain message parts are encrypted as specified. In practice, the messages should be intercepted after their preparation has been finalized by the sending party, but before they are sent out. Due to the importance security issues have, an intrusive and timely approach should be used, to prevent insecure messages from being sent out.

On the other hand, if the goal is to monitor the correct establishment of trust relationships, a slightly different approach should be used. Since WS-Trust embeds special tokens in messages using the WS-Security specification, it is important to verify their presence at the message level. However, WS-Trust also specifies multi-party protocols for obtaining the needed tokens, and these should be verified as well. Moreover, in these protocols, it is often the case that a number of intermediaries—already in a trust relationship—are used to help establish the new relationship (i.e., between John's system and the bank web service). These collaborations are typically choreographic in nature, especially when concepts such as trust federations are introduced. As a consequence, monitoring should also verify that the desired protocols perform as expected.

9.3.4 Monitoring Response Time

Web service response times are typically monitored by web service providers, who establish control policies on their assets and plan changes in their deployment strategies, should response time degrade over time (e.g., due to request overload). Examples of how a deployment strategy can be modified and improved are the migration to more capable servers or the deployment of new instances of the service.

However, clients are also directly interested in monitoring the response times of services they interact with. For example, in our scenario, John's haptic device could be interested in monitoring the time taken by his bank's web service to open a secure channel with the highway's tollgate payment service, pay the toll, and have the tollgate lift its bars. In this case, one might define a non-invasive approach that proceeds in parallel with the normal process execution. Through statistical analysis, the monitor may proactively discover non-functional problems before they actually occur. This would give the onboard computer the time to let John know if he should slow down, or avoid the automatic gates entirely and proceed to one where he can pay manually.

9.4 Dynamo

Dynamo (Dynamic Monitoring) is an approach and a toolset we developed to support service monitoring. Its conceptual roots originate from the software engineering community, and in particular can be traced back to assertion languages like Anna (Annotated Ada [20]), JML (Java Modeling Language [3]), and the notation added to the Eiffel language [22] to support "Design by Contract" [24]. These languages allow designers to add constraints to their programs in the form of assertions, typically pre-conditions, post-conditions, and invariants.

Dynamo provides a language called WSCoL [4]—similar to the light-weight version of JML—which allows designers to specify constraints on orchestrated collaborations. WSCoL is tailored toward the de-facto standard BPEL and supports the definition of pre- and post-conditions for activities that interact with external services (i.e., invoke, receive, reply, and pick). Dynamo monitors the evolving client-side state of the process and assumes that it can be modified erroneously only through external collaboration. That is, the approach trusts the internal business logic, but not the execution of the external services the process is bound to. This is the reason why post-conditions must be checked. On the other hand, pre-conditions may be useful in the debugging phase of a service composition to check that external services are invoked correctly. Dynamo also fosters separation of concerns since monitoring is defined as a cross-cutting concern. Designers can concentrate on the business logic and on the monitoring directives independently. Therefore, we can say the approach is non-invasive at the specification time.

To favor adoption of our monitoring approach, the BPEL execution environment was not changed: appropriate external services—called Monitoring Managers—are responsible for analyzing WSCoL constraints. The business logic is unaffected by the monitoring, but to allow the process to interact with the external monitors, additional BPEL code is added to the process at deployment time by means of static weaving. This leads to an intrusive approach (with regard to the execution of the system itself), which blocks the process execution to check pre- and post-conditons to discover erroneous situations in a timely fashion, i.e., as soon as they occur.

Dynamo explicitly supports—through the WSCoL specification language—two main kinds of data collection: (1) directly from the process and (2) from external data sources, if these are provided via web service interfaces.

Figure 9.3 summarizes the approach and gives a better idea of the static weaving that occurs at deployment time. The component responsible for weaving the code that ties the process to the external monitoring managers is called BPEL^2. It takes as inputs both the non-monitored version of the business process—specified in terms of BPEL code—and an external *Monitoring Definition File*. This file contains both the WSCoL constraints to be checked and the "locations" within the process in which (i.e., the BPEL activities for which) the constraints should be verified. These locations are expressed using an XPATH [8] expression (since BPEL is an XML specification language) and a keyword indicating whether the condition is a pre-condition or a post-condition.

The monitored version of the process that is produced substitutes each BPEL invocation for which a pre-condition or a post-condition, or both, has been defined (see invocation of service B in Fig 9.3), with a call to the Monitoring Manager, which acts both as a proxy for the service invocation and as a gateway toward external components that can act as WSCoL constraint

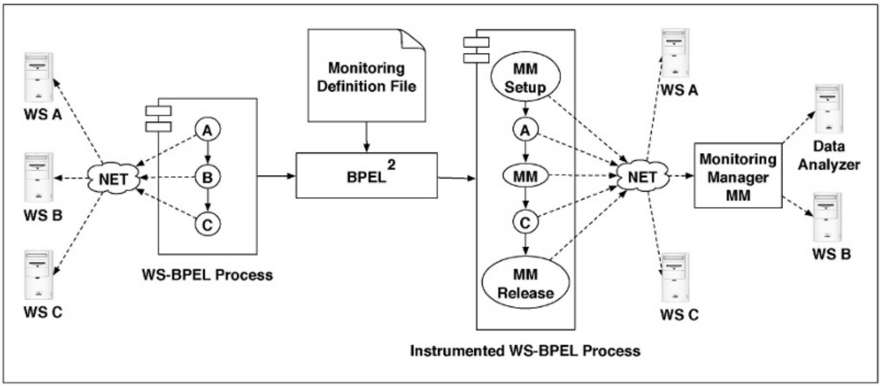

Fig. 9.3. An overview of Dynamo

analyzers. The weaving also adds some additional code at the beginning of the process and at the end, respectively responsible for the set up of the monitoring manager, and its release once the process execution terminates.

9.4.1 WSCoL

WSCoL, our monitoring language, allows the designer to do the following:

- Define and predicate on variables containing data originating both within the process and outside the process.
- Use pre-defined functions, e.g., string concatenation.
- Use the typical boolean operators such as && (and), || (or), ! (not), => (implies), and <=> (if and only if), the typical relational operators, such as <, >, ==, <=, and >=, and the typical mathematical operators such as $+$, $-$, $*$, and $/$.
- Predicate on sets of variables through the use of universal and existential quantifiers.

Since the web services invoked by a workflow may be considered as black boxes that expose public methods, which take an input and produce an output, there is no side effect on input variables. Assertion expressions may, therefore, refer to variables in the input message without distinguishing between the value prior to service invocation and the value afterward.

WSCoL will be introduced via examples, to describe properties that can be verified using Dynamo in the case study outlined in Sect. 9.3.

Internal Variables

It is common practice in BPEL to use one variable to contain the data that must be sent to a web service, and another variable to contain the data that the invocation returns to the process. These variables match the XSD types of the input and output messages of the web method being called, as defined in the service's WSDL description. WSCoL can refer to internal BPEL variables through use of a syntax which is somewhat similar to XPATH. The designer must specify the name of the variable, and the internal path from the root of the variable to the actual content he/she wants to refer to. The XPATH must point to a simple data type, since WSCoL does not allow the definition of relationships between complex data types.

In Sect. 9.3, to express the functional requirements of the "Holiday Location Finder" service, we need to refer to the maximum location distance. Figure 9.4 shows the structure of the internal BPEL variables "RequestSpec" and "LocationResults" used to call the "Holiday Location Finder Web Service" web method. To refer to the maximum location distance we can write:

```
($RequestSpec/maxLocationDistance)
```

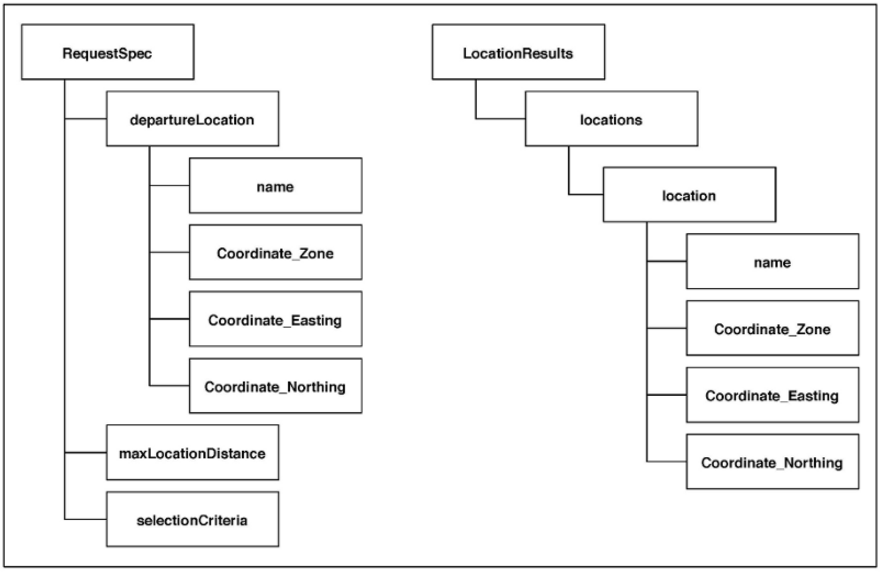

Fig. 9.4. The input and output messages for the Holiday Location Finder Web Service

The first part of the expression is introduced by a dollar sign and indicates the BPEL variable we are referring to (i.e., "RequestSpec") while the remaining part specifies how to obtain the "maxLocationDistance" value from the variable. In this case the XPATH expression matches a node containing a integer value (see Fig. 9.1), on which a function like "abs" can be used to evaluate the absolute value.

External Variables

WSCoL allows the designer to refer to external variables through the concept of external data collection. External variables can be simple data types such as strings, integers, longs, booleans, etc. WSCoL provides a number of functions for data collection, one for each simple data type that can be returned, and assumes the external data collectors being used can be queried through a web service interface.

In the example discussed in Sect. 9.3, we need to first discover the map's resolution (of which we only had a byte representation), and then compare it with the highest resolution accepted by the haptic device—say 300 by 200 pixels. To do so, we use a data collector (e.g., the "imageInfoService," whose return type is shown in Fig. 9.5), which provides the resolution of an image it is given as input.

The common signature for WSCoL's data collection functions is

```
(\return<X> (W, O, Ins, Out))
```

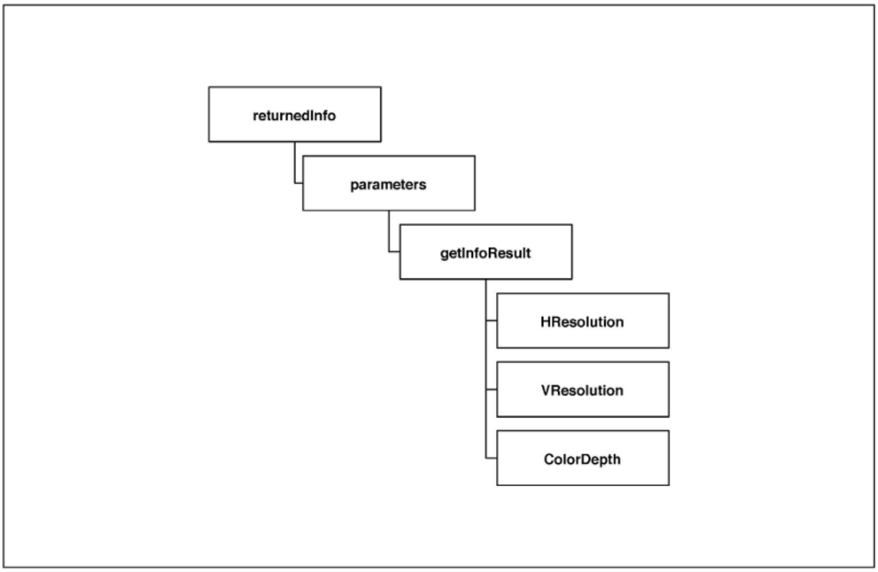

Fig. 9.5. Structure of the return type for “imageInfoService”’s “getInfo” web method

where

- *X* is the XSD type of the function’s return value.
- *W* is the location of the WSDL specification for the data collector that is to be used.
- *O* is the name of the operation (web method) that is to be called on the data collector.
- *Ins* is a string concatenation of the input values that should be used when calling the data collector’s web method.
- *Out* is an XPATH indicating how to obtain the correct return value within the complex data type returned by the data collector.

Figure 9.6 shows a post-condition that specifies the requested resolution (higher than 300 by 200 pixels) of the map returned by the service.

Quantifiers

WSCoL also offers designers the possibility to use universal and existential quantifiers. These are useful in cases in which we want to express constraints on sets of values.

Universal quantifiers indicate a constraint that must be true for each element in a given range. They follow a simple syntax:

```
(\forall $V in R; C)
```

```
(\returnInt('WSDL', 'getInfo', ($getRoute/parameters/getRouteResult),
'//parameters/getInfoResult/HResolution') <= 300 &&
(\returnInt('WSDL', 'getInfo', ($getRoute/parameters/getRouteResult),
'//parameters/getInfoResult/VResolution') <= 200
```

Fig. 9.6. The post-condition on the map web service

They indicate a constraint that must be true for each element in a given range. The meanings of the different parts are as follows:

- $\$V\ in\ R$ defines the variable and the finite set in which the variable is considered. The set is defined using the syntax previously introduced for variables, where the XPATH expression returns a set of nodes, instead of a single node.
- C defines the constraint that must hold.

For example, the "findLocation" web method in the "Holiday Location Finder" web service returns an array of locations (see Fig. 9.4 for the structure of the returned data type). In Sect. 9.3 our post-condition for this method was that "all the returned locations should be within the maximum location distance specified in the request." The WSCoL constraint can be seen in Fig. 9.7.

Existential quantifiers follow a similarly simple, and equally intuitive, syntax:

```
(\exists $V in R; C)
```

9.4.2 The Monitoring Manager

The internal architecture of the Dynamo monitoring manager is shown in Fig. 9.8. It follows a plug-in style, which allows it to interact with different kinds of data analyzers for different kinds of properties. In its current implementation, Dynamo uses the XlinkIt engine [27] as its external data analyzer. The following are the monitoring manager's principal components:

```
(\forall $l in ($LocationResults/locations/location/);
($l/Coordinate_zone)==($RequestSpec/departureLocation/Coordinate_Zone) &&
[($l/Coordinate_Easting) - ($Request/departureLocation/Coordinate_Easting)] ^2 +
[($l/Coordinate_Northing) - ($Request/departureLocation/Coordinate_Northing)] ^2 <=
($RequestSpec/maxLocationDistance)^2)
```

Fig. 9.7. The post-condition for the "findLocation" web method

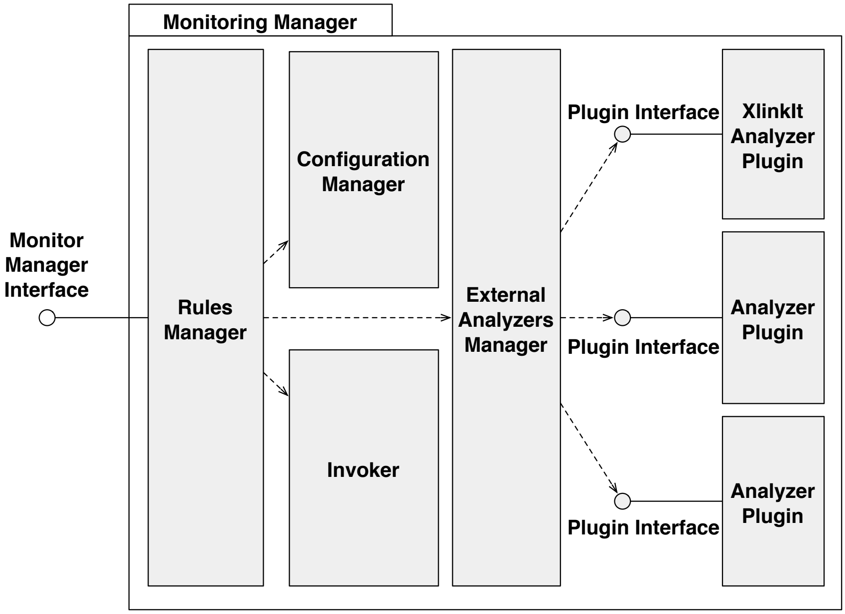

Fig. 9.8. The architecture of the monitoring manager

- *Rules manager*, which represents the interface through which the monitoring manager interacts with its clients. It is responsible for managing the monitoring manager's set up, for how the other internal components collaborate to achieve constraint verification, and for releasing the monitoring manager's resources, once the executing process no longer needs monitoring.
- *Configuration manager*, which contains all the information needed by the other components to verify the constraints. Every time weaving is performed, the BPEL^2 component adds a snippet of BPEL code at the beginning of the monitored process. This allows the configuration manager to be set up independently for each process to be monitored. In particular, the extra code sends the monitoring manager all the WSCoL assertions it will be asked to verify during the process execution, thereby reducing the amount of data that will be sent each time a constraint needs to be checked, by restricting it to information that can be obtained only at run-time.
- *External analyzers manager*, which allows different external data analyzers to be used by the monitor. This component is responsible for transforming the collected monitoring data and the WSCoL assertions into the specific formats that can be understood by the external data analyzers. In the case of XlinkIt, the data are transformed into XML data files, while the WSCoL rules are transformed into [25] rules.

- *Invoker*, which can invoke any external component, provided it has a WSDL interface. It is used for external data collection, to invoke external data analyzers, and to invoke the external web service being checked and for which the monitoring manager is acting as a proxy.

The collaboration diagram of Fig. 9.9 illustrates how run-time monitoring is achieved. The figure illustrates a simple case in which (1) the *Rules manager* checks whether a pre-condition is defined in the *Configuration manager* for the specific service invocation being monitored (steps 1–2), (2) discovers that a constraint exists and asks the *External analyzers manager* to use the appropriate analyzer plug-in to transform the monitoring data and the WSCoL constraint into suitable formats (steps 3–6), (3) asks the *Invoker* to call the external data analyzer to verify the constraint (steps 7–10), (4) finds out that the constraint holds and asks the *Invoker* to call the external service (steps 11–14), and finally gets back to the process with the data it is expecting (step 15). Although many interactions take place, the implementation is extensively configurable. All components can be kept local in order to minimize the amount of needed distributed interactions.

The actual cost of our approach in terms of distributed interactions is difficult to quantify. On the one hand, each call to an external service being monitored is substituted by a call to our *Monitoring manager* proxy. At that

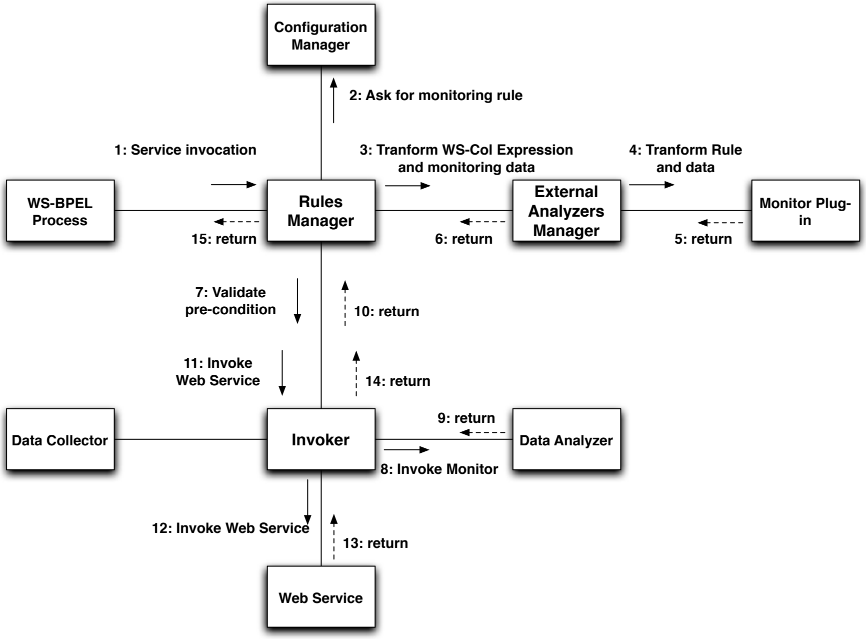

Fig. 9.9. Checking a pre-condition

point, two things can happen depending on whether a pre-condition has been defined or not. In the first case, if the pre-condition is verified correctly then the proxy will call the actual external service. In the second case, a post-condition must be present (if not the proxy would not have been called). In order to verify such a condition, the proxy must first call the actual external service. Therefore, in the worst case, from a performance standpoint, two service invocations are performed: one to the proxy and one to the service itself. Their payloads are similar, except that the call to the proxy contains the extra monitoring data collected from within the process in execution. The actual amount of extra data depends solely on the nature of the WSCoL expressions defining the pre- and/or post-condition being checked. External data collection, through which data are gathered from external sources that expose a web service interface, also affects performance. In fact, the occurrence of external variables in pre- or post-conditions implies extra remote invocations that must be performed at run-time.

Regardless of the actual number of service invocations being performed, however, the main performance bottleneck in the current version of Dynamo is due to the verification of the CLiX rules (after they have been translated from the original WSCoL rules) performed by XlinkIt, which uses XML files to perform its tasks. We are currently producing a pure WSCoL analyzer based on Java that will solve this problem by keeping the data in main memory, without leaning on filesystems and databases.

9.5 Other Monitoring Approaches

This section reviews a number of research and industrial monitoring approaches and discusses their properties in terms of the classification items presented earlier. For some of these approaches, more in-depth presentations can be found in the other chapters of this monograph. A summary of our comparative analysis of all the approaches is presented in Table 9.1.

9.5.1 Research Approaches

Requirements Monitoring

Spanoudakis and Mahbub [26] present an approach in which the requirements to monitor in a BPEL workflow are defined using event calculus, a first-order logic that incorporates predicates for expressing temporal features. An event interceptor component is needed to capture phenomena, such as operation invocations, return messages, etc. By tying the event interceptor to a centralized execution engine, with this approach it is not necessary to instrument the individual services in the collaboration.

Two kinds of requirements are considered: behavioral properties, automatically obtained from the BPEL collaboration specification, and behavioral assumptions that are manually specified. When events are collected at run-time, they are stored in an event database. The specified properties are then verified against the collected data, using variants of integrity-checking techniques in temporal databases.

This approach is meant to capture erroneous situations post-mortem. Even though the approach is tailored toward monitoring functional properties, non-functional properties can also be expressed and verified, such as properties regarding response times. Since events are collected in parallel with the process execution, a low degree of invasiveness is ensured.

For a deeper analysis of this approach, see Chap. 10.

Planning and Monitoring Service Requests

A significantly different approach is proposed by Lazovik et al. [18]. They present a planning architecture (with a specially tailored run-time environment) in which service requests are presented in a high-level language called XSRL (Xml Service Request Language). They adopt a proprietary orchestrated approach to collaboration, since they claim that current standards, like BPEL, do not have the necessary flexibility to satisfy user requirements that heavily depend on run-time context information.

The planing architecture is based on a continuous interleaving of planning steps and execution steps. Because BPEL lacks formal semantics, the authors decided to extrapolate state-transition systems from BPEL specifications and to enrich them with domain operators and constructs.

This framework is based on reactive monitoring. In particular, designers can define three kinds of properties: (1) Goals that must be true before transitioning to the next state (2) goals that must be true for the entire process execution, and (3) goals that must be true for the process execution and evolution sequence. The XSRL language also allows for the definition of constraints as boolean combinations of linear inequalities and boolean propositions. It provides sequencing operators such as “achieve-all,” “before” and “then,” “prefer” goal x “to” goal y, and “then.” It also defines a number of operators that can be used on the propositions themselves, defining how these propositions should be satisfied such as “vital” and “optional.”

The delivery platform continuously loops between execution and planning. In particular, the latter activity is achieved by taking into account context and the properties specified for the state-transition system. This makes it possible to discover, each time it is undertaken, whether a property has been violated by the previously executed step, or if execution is proceeding correctly.

9.5.2 Industrial Approaches

In the last few years, numerous industrial approaches to monitoring have been developed. With respect to research proposals, industrial approaches tend to be tailored on the requirements of service providers and concentrate on monitoring low-level events. Most of the monitoring approaches are part of a deployment environment, and consist of either ways to capture low-level information (such as response time and throughput) or exceptions that occur while trying to enforce certain non-functional properties (or policies). We will start by presenting examples of the latter, by looking at two industrial proposals: Cremona and Colombo. We will then conclude by investigating lower-level approaches such as GlassFish and IBM'sTivoli Composite Application Manager for SOAs.

Cremona

Cremona is a proposal from IBM, which is currently distributed within the Emerging Technologies Toolkit (ETTK) [19]. Cremona, which stands for "Creation and Monitoring of WS-Agreements," is a special-purpose library devised to help clients and providers in the negotiation and life-cycle management of WS-Agreements (i.e., their creation, termination, run-time monitoring, and re-negotiation).

A WS-Agreement is an XML binding between clients that require specific functional and/or non-functional properties be ensured at run-time and providers that promise them. The standard, proposed by the GRAAP(Grid Resource Allocation and Agreement Protocol) workgroup, provides XML syntactical templates for agreements—protocols that should be followed during the creation of an agreement—and a number of operations that can be used to manage them throughout their life-cycle.

Regarding the monitoring problem, the Cremona framework provides an "Agreement Provider" component, whose structure incorporates, among other things, a "Status Monitor." This component is specific to the system providing the service. By consulting the resources available on the system and the terms of an agreement, it helps decide whether a negotiation proposal should be accepted or refused. Once an agreement has been accepted by both parties (the client and the provider), its validity is checked at run-time by a "Compliance Monitor," a sophisticated system-specific component that can check for violations as they occur, predict violations that still have to occur, and take corrective actions. Since both monitoring components are system dependent, designers are guaranteed great flexibility in terms of the properties they can check.

Colombo

Colombo [9] is a lightweight middleware for service-oriented architectures proposed by IBM. It advocates that an optimized and native run-time environment, which does not build upon previously existing application servers, can provide greater performance, and guarantee simplified models for development and deployment. It supports the entire web service stack and, in particular, orchestrated collaborations defined using BPEL. It also supports declarative service descriptions, such as those expressed using WS-Policy [14].

Table 9.1. Comparing monitoring approaches

Approach Name	Types of Properties	Collaboration paradigm	Collecting monitoring data	Timeliness	Abstraction level	Validation technique	Monitoring goals
Dynamo	Mainly functional (and simple non-functional) properties	BPEL-based orchestrations	Collected by the process itself, or through external data sources	Blocking pre- and post-conditions	Programming level	Assertion-checking	Tools for composition providers who need to monitor the external services used
Requirements monitoring	Mainly functional (and simple non-functional) properties	BPEL-based orchestrations	An interceptor component listens for low-level engine events	Post-mortem	Low-level sequences of engine events	Variant of integrity checking in temporal deductive databases	Tools for composition providers who need to monitor the external services
Planning and monitoring Service Requests	Process and evolution sequence goals	Proprietary orchestration-based delivery framework	Collected within the proprietary framework	Errors discovered as soon as they occur	Requirements and specification level (market domain terminology)	Assertion-checking approach	Tools for composition providers who need to monitor process evolution
Cremona	Functional and non-functional, and properties of histories of interactions	No specific paradigm, but any interaction between a caller and the provider	Server-side regarding the interaction channel and the system's resources	Reactive approach	WS-Agreement templates with different property description languages.	Implementation-specific techniques	Tools for service providers who need to monitor agreements with their clients
Colombo	Mainly non-functional properties (WS-Policy)	Optimized middleware for SOA that supports BPEL	Through a pipe of dedicated policy-specific verifiers	Before a message leaves the system, or before the incoming message is processed.	Service, operation, or message level	Validation is policy dependent	Tools for service providers who need to monitor policy compliance of incoming and outgoing mesages
GlassFish	Mainly non-functional properties	Proprietary deployment infrastructure	Response times, throughputs, numbers of requests, and message tracing	No automatic analysis. Timeliness does not depend on the system	Three standard macro-degrees of monitoring	No automatic validation	Tools for service providers who need to monitor statistics of client-service interactions
IBM TIvoli Composite Application Manager	Mainly non-functional properties	Event-based system (integration bus)	Messages as they enter or leave the integration bus.	No automatic analysis. Timeliness does not depend on the system	WS-Policy for QoS	No automatic validation	Tools for service providers who can personalize monitoring on-top of the service bus structure

WS-Policy is a declarative language that aggregates quality-of-service assertions that are defined using domain-specific languages. Of the many domain-specific policy languages already defined or being defined, WS-Security and WS-Transactions are the most prominent. Policies are statements that can be attached to a service, to a single operation, or even to a single message type. Therefore, recalling the example of Sect. 9.3, Colombo could be used to monitor the messages being sent to the bank service and to check whether they satisfy the specified security policies (i.e., encryption, authentication, etc.). Colombo manages incoming and outgoing messages by passing them through two corresponding pipes of dedicated policy verifiers and enforcers (i.e., one for each kind of policy supported by the system), it can discover erroneous behavior in a timely fashion, but is intrusive in nature. It provides support for important issues, such as security.

9.5.3 Other Approaches

Many other industrial approaches to the monitoring of service-oriented systems exist. Most of them, however, tend to interpret monitoring at an even lower level of abstraction. In fact, they limit themselves to logging the messages being sent in and out of a system. They can be assimilated to mere data collectors, since there is seldom any automatic analysis of functional or non-functional properties, and data are interpreted manually.

GlassFish

GlassFish [12] is an open-source community implementation of a server for Java EE 5 applications. Regarding monitoring of deployed services, GlassFish provides a number of specific tools. Using technologies such as "J2EE Management" [13] and "Java Management Extensions"[21], GlassFish makes it possible to access information on resources and properties that are tied to the web services to be monitored. This information is given in the form of operational statistics (and in graphical form as well). The nature of the monitored aspects depends on the level of monitoring chosen for a given service. There are three possible levels: *low*, which monitors response times, throughput, and the total number of requests and faults; *medium*, which adds message tracing under the form of content visualization; and *off*, in which no data is collected. Captured information can also be automatically aggregated to obtain "minimum response times," "maximum response times," "average response times," etc.

Regarding the examples presented in Sect. 9.3, this approach could be helpful in monitoring response times. Analysis of the monitored data could then be achieved either manually, or automatically, possibly in conjunction with a more sophisticated monitoring approach, such as Dynamo. This could be the case of the examples presented in Sect. 9.3, in which John's haptic device needs to know how much time it usually takes to interact with the bank service, pay the toll, and open the toll bars.

IBM Tivoli Composite Application Manager for SOAs

Another similar approach is the IBMTivoli Composite Application Manager for SOAs [31]. This application manager uses an event-based collaboration paradigm, implemented through a special-purpose integration bus. Messages enter and leave the bus continuously, passing through special components called the "ServiceBusInbound" and the "ServiceBusOutbound," making it easy to monitor their behavior and, in particular, their performance. However, the application manager lacks the specially tailored tools present in other similar approaches.

9.6 Conclusions

In this chapter we argued that dynamic software architectures, like SoAs, require verification to extend to run-time. In fact, since both the components of an application and their interconnections may change after deployment, traditional pre-deployment verification is not enough to guarantee that the application will satisfy the required quality requirements. We discussed run-time monitoring as a possible solution to this problem, and we analyzed the possible dimensions that may characterize the monitoring activity. In particular, we zoomed into an approach to monitoring that we investigated in our research, based on assertions.

We believe that monitoring should also be the basis for architectural recovery. It should be possible to design SOAs that provide self-organized reactions, which may occur as deviations from the expected quality requirements detected by the monitor. This is still an open and challenging research direction in which we plan to invest our future efforts.

We are also considering a new version of the Dynamo framework that relies heavily on Aspect-oriented Programming technology. In particular, we are using AspectJ to enhance the ActiveBPEL engine [1] with Dynamo's monitoring capabilities. Such an approach is allowing us to treat business logic and monitoring as two completely cross-cutting concerns that are only intertwined at run-time. The original process is no longer modified at deployment-time and is directly deployed to the framework, regardless of the number of monitoring strategies defined by the different stakeholders. The approach also has another advantage. Since the actual service invocations are no longer performed by the Dynamo framework, which is only responsible for monitoring, but by the ActiveBPEL engine itself, all general-purpose policies supported by ActiveBPEL are a given. Such an approach also provides slightly better performance.

Finally, we have also been using WSCoL and slightly extended versions of Dynamo to enable the management of general policies such as those used within the WS-Policy spec [14]. Some initial results have been achieved [5], but the work is still ongoing.

References

1. ActiveBPEL The Open Source BPEL Engine, 2006.
2. B. Atkinson, G. Della-Libers, S. Hada, M. Hondo, P. Hallam-Baker, J. Klein, B. LaMacchia, P. Leach, J. Manferdelli, H. Maruyama, A. Nadalin, N. Nagaratnam, H. Prafullchandra, J. Shewchuk, and D. Simon. Web Services Security (WS-Security), 2002.
3. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.
4. L. Baresi and S. Guinea. Towards Dynamic Monitoring of BPEL Processes. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer, 2005.
5. L. Baresi, S. Guinea, and P. Plebani. WS-Policy for Service Monitoring. In C. Bussler and M. Shan, editors, *TES*, volume 3811 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2005.

6. L. Baresi, S. Guinea, and M. Plebani. Business Process Monitoring for Personal Dependability. In *Workshop SOAM 06 Modeling the SOA – Business Perspective and Model Mapping*, 2006.
7. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. WSDL: Web Services Definition Language. W3C Technical Reports on WSDL, published online at http://www.w3.org/TR/wsdl/, 2004.
8. J. Clark and S. DeRose. Xml path language version 1.0, 1999.
9. F. Curbera, M. J. Duftler, R. Khalaf, W. A. Nagy, N. Mukhi, and S. Weerawarana. Colombo: Lightweight Middleware for Service-Oriented Computing. *IBM Syst. J.*, 44(4):799–820, 2005.
10. I. Foster and C. Kesselman. Scaling system-level science: Scientific exploration and it implications. *Computer*, 39(11):31–39, November 2006.
11. I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004.
12. H. Hrasna. GlassFish Community Building an Open Source Java EE 5 Application Server, 2006.
13. H. Hrasna. JSR-000077 J2EETM Management, 2006.
14. IBM, BEA Systems, Microsoft, SAP AG, Sonic Software, and VeriSign. Web Services Policy Framework, 2006.
15. IBM, Microsoft, Layer 7 Technologies, Oblix, Verisign, Actional, Computer Associates, OpenNetwork Technologies, Ping Identity, Reactivity, and RSA Security. Web Services Trust Language, 2005.
16. IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services 1.1, 2005.
17. N. Kavantzas, D. Burdett, and G. Ritzinger. Web Services Choreography Description Language Version 1.0, 2004.
18. A. Lazovik, M. Aiello, and M. P. Papazoglou. Associating Assertions with Business Processes and Monitoring their Execution. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 94–104. ACM, 2004.
19. H. Ludwig, A. Dan, and R. Kearney. Cremona: an Architecture and Library for Creation and Monitoring of WS-Agreements. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 65–74. ACM, 2004.
20. D. C. Luckham and F. W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.
21. E. McManus. JSR-000003 JavaTM Management Extensions, 2006.
22. B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.
23. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.
24. B. Meyer. Design by Contract, Components and Debugging. *JOOP*, 11(8):75–79, 1999.
25. M. Marconi and C. Nentwich. CLiX ¡constraint language in xml/¿, 2004.
26. K. Mahbub and G. Spanoudakis. A Framework for Requirents Monitoring of Service Based Systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing*, pages 84–93. ACM, 2004.
27. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. Xlinkit: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.
28. M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Execution of Composite Web Services. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 170–187. ACM, 2004.

29. The OWL Services Coalition. OWL-S: Semantic Markup for Web Services, 2003.
30. D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Trans. Software Eng.*, 21(1):19–31, 1995.
31. IBM Tivoli Composite Application Manager for SOA, 2006.