A Monitoring Approach for Dynamic Service-Oriented Architecture Systems

Yufang Dan*‡, Nicolas Stouls*, Stéphane Frénot*, and Christian Colombo†
*Université de Lyon, INRIA, INSA-Lyon, CITI, F-69621, France – Email: first.second@insa-lyon.fr
†Department of Computer Science, University of Malta – Email: first.second@um.edu.mt
‡College of Computer Science Chongqing University, Chongqing, China

***Abstract*—In the context of Dynamic Service-oriented Architecture(SOA), where services may dynamically appear or disappear transparently to the user, classical monitoring approaches which inject monitors into services cannot be used. We argue that, since SOA services are loosely coupled, monitors must also be loosely coupled. In this paper, we describe an ongoing work proposing a monitoring approach dedicated to dynamic SOA systems. We defined two key properties of loosely coupled monitoring systems: *dynamicity resilience* and *comprehensiveness*. We propose a preliminary implementation targeted at the OSGi framework.**

Keywords-Monitoring; Dynamic SOA; OSGi; Larva.

I. Introduction

Service oriented architectures (SOA) is one of the current approaches to develop well structured software. It is focused on loosely coupled client-server through interfaces. The client usually requests service access through a repository. Subsequently, the client is bound to the service and is allowed to invoke methods as long as the interface types match. Among SOA approaches, we will focus on dynamic SOA, such as OSGi [1], usually used in 24/7 systems, where system may not be restarted when a service appears or disappears. In dynamic SOA, each invocation (potentially with the same client) must be considered as a completely new context change since potentially new services may appear and others disappear. From a dynamic SOA point of view, binding a client to a service is a matter of interface matching, but, neither the client nor the service has a guarantee that the other part behaves as expected. For instance, each time a client makes a request to a server, a formally specified constraint can be checked to ensure whether the client is authorized to perform that call or not.

Existing runtime monitoring tools such as JavaMOP [2] or Larva [3] weave interception calls using aspect-oriented programming techniques. These approaches work fine in SOA since client-server bindings are usually generated upon the first invocation and preserved throughout the entire client life cycle. On the other hand, in dynamic SOA, bindings may be reconsidered at runtime. Hence, any monitoring state wove into the service implementation gets reset.

Fig. 1 illustrates an example needing a dynamicity resilient monitor. Let us consider a client embedded on a mobile device based on a dynamic SOA platform and needing to communicate with a distant system according to a particular protocol. If two services S_1 and S_2 provide a single dedicated interface for accessing this distant system, but through different medium (WiFi, 3G, etc.), then the system could need to substitute the use of S_1 by S_2 and conversely. We propose a monitoring approach allowing such substitution without impacting the communication and then the monitored property. Hence, if any behavioral property has to be respected in the use of the distant system (such as "after a call of A, a call of B must occur") then the property is exactly the same in the whole system without any modification, even if the intermediate service is substituted.

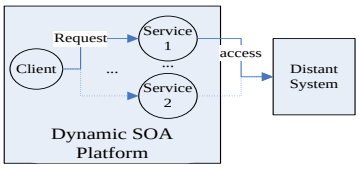

Figure 1. Example of Dynamic SOA System With Monitoring Restrictions

Our proposal is to bring a dynamic approach to runtime monitoring systems. We suggest that monitors must not be statically inserted into the observed system, but brought dynamically at binding time between the client and the service. Having a dynamic approach means that the service bindings and the behavioral monitoring bindings must be both considered as dynamic and loosely coupled. This article stresses the use for a dynamic runtime monitoring tool, that enables service substitution at runtime. This approach may preserve the current behavioral states and check that old and new service implementations are behaviorally compatible.

A dynamic runtime monitor must have two significant characteristics: *dynamicity resilience* and *comprehensiveness*. The former refers to the preservation of the behavior flow: in case of substitution of the monitored service, we want to keep alive the monitoring and the current state of the property. Hence, the property cannot be hard-linked to the code. The latter characteristic means that we cannot allow services to restrict what is observable by the monitor: if we want to check a property, we need to ensure that all the relevant events are monitored. We are not assuming that every service provides its authorized behaviors, but only that if an authorized service want to check the respect of a property on the framework, then no service can bypass

this observation. The architecture relies on a generic event interception mechanism and a dynamic, loosely coupled, wiring mechanism for automaton verification. The verification automaton is extracted from Larva.

Section 2 of the article presents some runtime verification approaches and proposes a classification of them, showing the gap we propose to fill. Section 3 expresses the architecture model for a dynamic runtime verification tool. Section 4 illustrates our OSGi reference implementations. Section 5 shows our initial conclusions, and Section 6 our future works.

II. Related works

We can classify existing runtime verification approaches according to the monitor configuration with respect to the monitored service. Property may be: manually written inside the code (Hard-Coding), automatically injected inside the code (Soft-Coding) and kept out of the code (Agnostic-Coding). For each of these families, we will discuss the resilience to dynamicity and the monitoring comprehensiveness.

A. Hard-coding

In this category, where properties are manually injected at source time, we can cite all annotation techniques, like JML [4] or Spec# [5]. In both cases, the monitor is not *resilient to dynamic* code loading. If the monitored system is substituted, then its monitor is also substituted, since it is inlined code. However, this approach is interesting in terms of *comprehensiveness*, since we can observe anything in the program. A limitation of this approach is the dispersion of the monitor throughout the code, requiring significant intervention to write the property or to check its description is correct.

B. Soft-Coding

In this category, where properties are injected at compilation time, we can cite Larva [3] or JavaMOP [2]. These tools need a standalone description of a property and inject the synthesized monitor inside the code. Advantages are then the same as in the previous case, but specifying the monitor is easier, since the description of the property is centralized. However, this approach is still not *resilient to dynamicity*; at best, the tool may inject the property at first-time binding, but once injected, the property is hard-coded within the service.

C. Agnostic-Coding

In this last category, where the monitor is kept out of the code, we include any trace analyzes approach, such as intrusion detection systems [6] or liability by logs approaches [7]. The main advantage of the approach is the loose linking between the property and the monitored system. Hence, if a package is substituted, the monitor can observe it inside the logs and the monitored properties are still the same for the whole system. Moreover, the description of the property is located into a single location, which facilitates property management. However, such a system can be bypassed, since it can only observe what services accept to push or what it can pull from observation points. If a package provides a service without writing sufficient logs, then the monitor does not have sufficient information to check a particular property [8].

In this paper, we propose a *comprehensive* and *dynamicity resilient* monitoring approach for dynamic SOA. The kind of system we target, sits between the *Soft-Coded* and the *Agnostic-Coded* approaches.

III. An Architecture enabling Dynamic and Comprehensive Runtime Monitoring

In this section, we describe an abstract architecture of a monitoring system supporting specific features of dynamic SOA systems. After that, we will discuss about the *resilience to dynamicity* and the *comprehensiveness* of the proposed architecture.

Our proposition consists in dynamically inserting a monitoring proxy in front of each service, and externalizing monitors in some autonomous services. When an event occurs, a notification is sent to each monitor, which checks the event against its property.

Since services are treated as black boxes from the running environment's point of view, such architecture is designed to consider only interface properties. It corresponds to properties expressing the normal/authorized use of a service. We then address behavioral properties.

In this architecture, the scope of properties is not restricted to the use of a single service. Indeed, there is no restriction to add a monitor in front of several services, in order to observe a global property on the system.

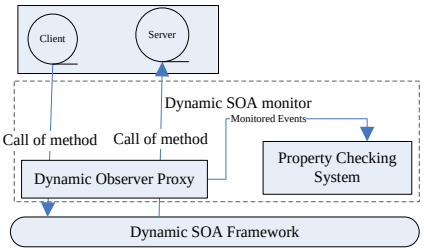

Figure 2. Monitor Abstract Architecture

Fig. 2 describes the whole abstract architecture, where we now detail the two main principles.

Resilience to Dynamicity: Since the monitoring system is not inlined inside the monitored service, but is externalized in an autonomous service, monitors are separated from the code. When changes occur in the framework,

the observation mechanism and its properties may remain unaffected.

Comprehensive Monitoring: One of the main concepts of dynamic SOA is to have a framework which allows dynamic loading and unloading of loosely coupled services. Since the framework is in charge of providing an implementation to each service request, the framework adds a proxy between the client and the service to observe communications. This observation is comprehensive and no communication can bypass this proxy, since the client and the service do not know directly each other.

IV. OSGiLarva — A Monitoring Tool for OSGi

In this section, we present OSGiLarva, an implementation of the proposed abstract architecture, using the OSGi framework. Our implementation integrates two existing tools: Larva [3] and LogOS [9]. LogOS is a special logging tool based on the OSGi framework, developed at CITI Lab during the LISE project [7]. We will use it as a hooking mechanism to observe services' interactions. Larva is a compiler which generates and injects a verification monitor into Java code. We will use an adaptation of Larva to enable property verification. Fig. 3 describes the OSGiLarva implementation of the abstract architecture we proposed.

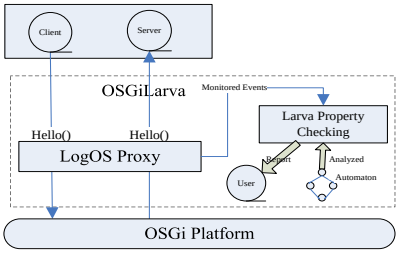

Figure 3. OSGiLarva Implementation

Currently, we use strictly the larva property description language. The addressed properties are the one that can be described by automaton.

We describe the monitor implementation with three key parts: we first present our adaptation of LogOS to intercept service interactions; Next, we give some details about our modifications of Larva and how it communicates with LogOS; Finally, we describe how the registration process of a service under OSGi will take into account an existing property monitor to insert it between the service consumer and the service itself.

A. LogOS: a Hook to Intercept Service Interactions

LogOS is a transparent toolkit for the OSGi architecture. As soon as the LogOS bundle is started, each registration of a service is observed by the system — thanks to the OSGi hooking mechanism — and a LogOS proxy is generated between the service and its consumer. Hence, every method call, including parameters and returned value, are automatically intercepted.

Each time such an event is captured, a corresponding LogOS event-description is forged and propagated to listeners. While the standard LogOS listener stores each event-description in a file, we have developed another listener that propagates events-description to the Larva monitor.

Finally, we have extended LogOS annotations, such that registered service interfaces may indicate at the deployment time their associated monitoring class. Interfaces and monitors can be provided separately from any implementation. In order to minimize the execution time of OSGiLarva, only registered events are notified to monitors.

B. Larva: a Property-Checking Monitor

Larva is a tool which injects monitoring code in a Java program to check a property described in a Larva script file. Upon compiling a script, the Larva Compiler generates two main outputs: (i) a Java class coding the property, and (ii) an aspect which links the monitoring code and source code. The aspect statically injects some calls to the monitor inside the Java software by using the AspectJ compiler. The Java code describing the property is then called each time an expected event occurs.

We keep the generation of the Java description of the property, which is the core of the monitor. However, we replace the static injection of the aspect into target services by an event-description propagation from LogOS to the monitor, based on dynamic proxy injection.

The new Larva provides a method accepting LogOS event-descriptions, which is dynamically called each time LogOS gets an event. Next, the Larva monitor starts analyzing this new event-description in the verified property.

C. Registration of a Service Providing Specification

In order to launch the monitoring of a service, we need to have a behavioral property to monitor. We propose to accept this property as a part of the OSGi bundle. An OSGi bundle provides three kinds of elements: a collection of *interfaces*, a collection of *services implementations* and a *bootstrap code* which is called when loading or unloading the bundle. Thanks to OSGi architecture, service interfaces, service implementations and bundles may have different life-cycles depending on the deployment scheme, since interfaces may be deployed with another bundle than service implementation.

As such, we keep the same philosophy, when providing properties that can be provided by the same bundle as implementation or by another one. Since interfaces are typing specifications of services, it makes sense to map the life cycle of properties to the one of interfaces. We then bind property monitor at the same time as interfaces and we keep monitoring until interfaces are removed.

In the current implementation of OSGiLarva, the property load is done by an explicit declaration referring to a, Larva-compiled, Java monitor in the manifest of the interfaces bundle. When this declaration is processed, LogOS is called. It loads the Larva property, generates a proxy between the new service and its consumer, and injects the given monitor inside the proxy.

V. Conclusion

In this paper, we have presented an approach to monitor dynamic SOA systems based on two main requirements: (i) *resilience to dynamicity* and (ii) *comprehensiveness*. The first one means that if a monitored service is substituted in the framework, the monitoring state is not reset. The comprehensiveness feature means that all services' interactions are monitored, i.e., it does not rely on the acceptance of the service designer or on the correct instrumentation of the service.

We have instantiated the approach in the context of the OSGi framework through a preliminary implementation, OSGiLarva, which integrates an adaptation of two existing tools: Larva and LogOS. Similar to Larva, OSGiLarva accepts the Larva property description language as input, hence inheriting all its features, including its expressiveness and its readability for non-expert users.

Our approach based on an OSGi hook to observe all occurring events seems to be inefficient when compared to injection-based monitoring tools, like Larva. However, this functionality is required to be resilient to the dynamicity of the system. Hence, we have to compare the OSGiLarva efficiency with that of other systems with the same features, such as classical log analyzes systems. In this context, our approach seems to have reasonable efficiency, since (i) we can configure filters in LogOS to push only the relevant events to Larva and (ii) we do not need to make hard drive accesses to read and write logs.

Finally, an interesting element of this approach is its non-intrusive aspect. Indeed, in contrast to the aspect oriented approach, we keep the original byte-code unchanged. This property can be interesting if we want to remove a monitor or always be able to check the binary signature of the code as an authentication credential [10].

VI. Future Work

At this point, we allow properties to take into consideration interface invocation events. In the future, we aim to introduce in the property description language the notion of loading/unloading of a bundle and the substitution of a service in order to express behaviors including such events.

Larva property description language allows timed properties, by the use of clocks. In the case of OSGiLarva, where a single service can be used by several consumers at the same time, we could want to introduce two levels of clocks: global to the service or associated to a consumer use. It seems that the foreach operator from Larva property description language could help us.

The current implementation of OSGiLarva is not finished according to our requirements. Indeed, the current declaration of events to log is done in the Java source file. In the future we aim to use the result of the Larva compilation to automatically define the list of events to observe. Finally, in a next version of the tool, we could make some propositions to reduce the OSGiLarva time cost. For instance, we could make OSGiLarva asynchronous, by exporting monitors in separated threads, or we can imagine that the use of a service need to be monitored only during a fixed time. If the property is respected during one week by a given consumer, we can consider that it will still respect it afterwards. In OSGiLarva, the removing a monitor is straightforward since it is non-intrusive.

References

[1] Open Service Gateway Initiative (OSGi), http://www.osgi.org/ [retrieved: June, 2012].

[2] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An Overview of the MOP Runtime Verification Framework," *International Journal on Software Techniques for Technology Transfer*, 2011.

[3] C. Colombo, G. J. Pace, and G. Schneider, "Larva - safer monitoring of real-time java programs," in *SEFM*, 2009.

[4] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs, "JML: notations and tools supporting detailed design in Java," in *OOPSLA 2000 COMPANION*. ACM, 2000, pp. 105–106.

[5] M. Barnett, R. DeLine, M. Fähndrich, B. Jacobs, K. R. M. Leino, W. Schulte, and H. Venter, "The Spec# Programming System: Challenges and Directions," in *VSTTE*, ser. LNCS, vol. 4171. Springer, 2005, pp. 144–152.

[6] C. Simache, M. Kaaniche, and A. Saidane, "Event log based dependability analysis of windows nt and 2k systems," in *International Symposium on Dependable Computing*, 2002, pp. 311 – 315.

[7] D. Le Métayer, M. Maarek, E. Mazza, M.-L. Potet, S. Frénot, V. Viet Triem Tong, N. Craipeau, R. Hardouin, C. Alleaune, V.-L. Benabou, D. Beras, C. Bidan, G. Goessler, J. Le Clainche, L. Mé, and S. Steer, "Liability in Software Engineering Overview of the LISE Approach and Illustration on a Case Study," in *ICSE'10*. ACM/IEEE, 2010, p. 135.

[8] H. R. M. Nezhad, R. Saint-Paul, F. Casati, and B. Benatallah, "Event correlation for process discovery from web service interaction logs," *VLDB J.*, vol. 20, no. 3, pp. 417–444, 2011.

[9] S. Frénot and J. Ponge, "LogOS: an Automatic Logging Framework for Service-Oriented Architectures," in *SEAAA*, 2012, p. to appear.

[10] P. England, "Practical Techniques for Operating System Attestation," in *1st international conference on Trusted Computing and Trust in Information Technologies*, ser. Trust '08. Springer-Verlag, 2008, pp. 1–13.