# OSGiLarva: a Monitoring Framework Supporting OSGi's Dynamicity

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

***Abstract*—Service-Oriented Architecture is an approach where software systems are designed in terms of a composition of services. OSGi is a Service-Oriented Framework dedicated to 24/7 Java systems. In this Service-Oriented Programming approach, software is composed of services that may dynamically appear or disappear. In such a case, classical monitoring approaches with statically injected monitors into services cannot be used. In this paper, we describe ongoing work proposing a dynamic monitoring approach dedicated to local SOA systems, focusing particularly on OSGi. Firstly, we define two key properties of loosely coupled monitoring systems: *dynamicity resilience* and *comprehensiveness*. Next, we propose the OSGiLarva tool, which is a preliminary implementation targeted at the OSGi framework. Finally, we present some quantitative results showing that a dynamic monitor based on dynamic proxies and another based on aspect-oriented programming have equivalent performances.**

***Keywords*-Monitoring, Dynamic SOA, OSGi, Larva, LogOs.**

## I. Introduction

This article is an extended version of [1], which has been published in IARIA Conferences 2012.

The service-oriented architecture (SOA) is one of the current approaches to develop well structured software supporting agility. It is focused on loosely coupled client-server interaction enabling the client to request server functionality through a repository that exposes appropriate interfaces. 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 distinguish between web services and other more local approaches such as OSGi [2] and .NET [3]. The main difference is that in the case of web services one would not typically be able to have the full view of the system, i.e., one can either monitor the client or the server but not both. On the other hand, in the case of local approaches one can reason about the full picture by also taking into consideration the OSGi framework events such as registration of services, service requests by different clients, etc.

In this work, we focus on OSGi, usually used in 24/7 systems, where the system is not restarted when a service appears or disappears. This framework is targeted to embedded systems such as cars, ADSL boxes, or network systems. In such systems, web services cannot be used either due to the lack of connectivity, network limited bandwidth, or for efficiency reasons. In the following, we focus on the OSGi framework, but the same principles can be applied to other local SOA systems, such as .NET.

In dynamic SOA, each service invocation must be considered as a complete context switch since potentially new services may appear and others disappear at runtime. 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. So, after interface matching, continuously ensuring the client's authenticity and the validity of the activities carried out are important for critical systems. For instance, each time a client makes a request to a server, a formally specified constraint can be checked to ensure that the client is authorized to perform that call.

Existing runtime monitoring tools such as JavaMOP [4] or Larva [5] weave interception calls using aspect-oriented programming techniques. This approach works fine in non-dynamic 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, due to runtime dynamic changes in the underlying service implementation, the monitoring state woven into the service implementation gets reset.

Our proposal is to bring a dynamic approach to runtime monitoring systems by inserting monitors at the point of client-server binding rather than "statically" at compile-time or loading-time. This means that both the service bindings and the behavioral monitoring bindings are dynamic and loosely coupled, thus supporting service substitution. This approach would preserve behavioral monitoring states across different service versions and check that both versions are behaviorally compatible.

Another major concern in a highly dynamic context, where the implementation of an interface may be substituted, is to ensure that no implementation, or part thereof, can bypass the monitoring framework. Note that if this could happen, the monitor would not be able to detect any malicious code which might be executed. Moreover, what can be concluded about a system's observation if some events could have been missed? Our aim is to enable the monitoring system to be fully active, even if the service provider ignores it.

In this context, we conjecture that a dynamic runtime monitor must have two significant traits: *dynamicity resilience* and *comprehensiveness*. The former refers to the preservation of the behavior flow: in case the monitored service is substituted, the monitor and its state should be transfered; meaning that 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. Note that we are not assuming that every service behaves as expected, but only that if an authorized service is to be checked for a particular property, then no event of the service behavior can bypass the monitor observations. For this reason, the architecture relies on a generic event-interception mechanism and a dynamic, loosely coupled, wiring mechanism for automaton verification. The verification logic of the automaton is then handled by an adaptation of the existing monitoring tool Larva [5].

Finally, the introduction of dynamicity to the monitor also increases the scope of properties we are able to address. Thus, we introduce some dynamic primitives in the property description language in order to make it possible to describe behavioral properties, where the registration/un-registration of a service is an expressible event. Furthermore, we also adapt the life cycle of properties, since, under different circumstances, the monitor state might need to be preserved or reset when the underlying service is substituted.

Section II is a case study showing some requirements of this work. Section III presents some runtime verification approaches and proposes a classification of them, showing the gap we propose to fill. It also discusses the trade-off between the observation scope and the expressible properties. Section IV expresses the architectural model for a dynamic runtime verification tool and introduces our OSGi reference implementation. Section V describes our modifications of the Larva specification language in order to consider dynamicity. Section VI illustrates the OSGiLarva tool by some quantitative results. Finally, Section VII shows our initial conclusions and Section VIII our future works.

## II. CASE STUDY

In order to ease the understanding of our contribution, this section introduces an example of a dynamically monitored system in line with our proposition.

Let us consider an embedded client on a mobile device based on a dynamic SOA platform, which needs to communicate with a distant system according to a particular protocol (Fig. 1). Let two services $S_1$ and $S_2$ provide an identical interface to access the distant system through different media: $S_1$ using a WiFi connection, and $S_2$ using a 3G connection. With such a configuration, we can consider that each time the WiFi connection goes down, the system unregisters $S_1$, effectively switching the client onto $S_2$, and vice-versa.

Moreover, we consider that the use of the distant system requires that the client is authenticated with the service and that some system actions have to execute atomically. Such requirements correspond to any typical secured system supporting concurrent access by transactions.

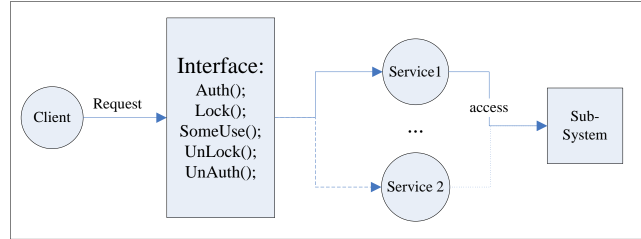



Figure 1. Dynamic SOA system supporting service substitution

In such an example, the possibility of service substitution is crucial. We then propose, in Fig. 2, an example of an execution scenario that has to be supported by the system. In this scenario, the service $S_1$ is substituted by $S_2$ during the atomic part of the run.

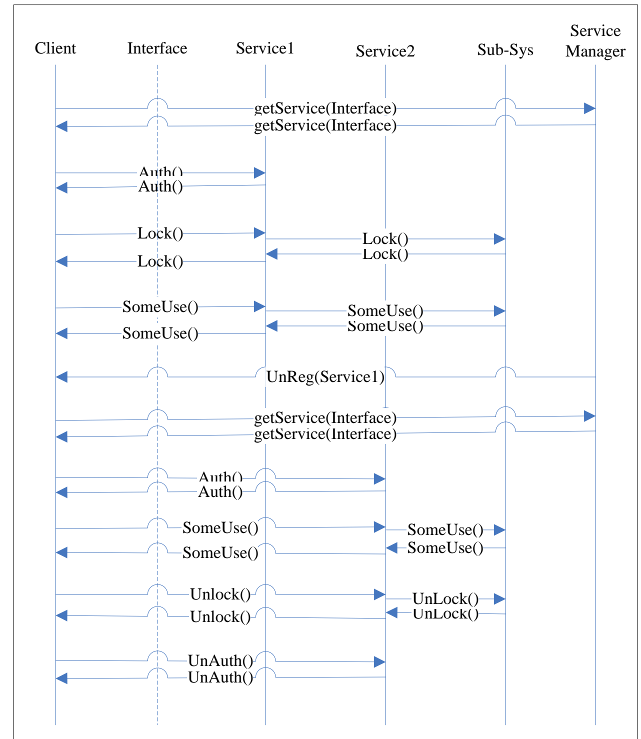
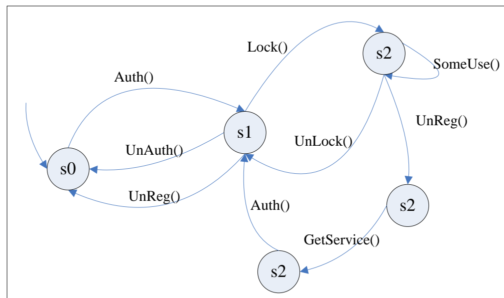



Figure 2. Example of scenario supported by example in Fig. 1

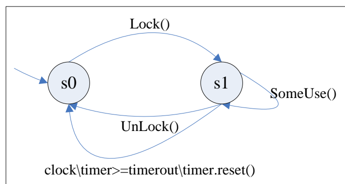
In another part, we can describe the correct use of the system in some property and check it by monitoring at

runtime. For instance, the two following properties express the expected behavior, described earlier: (i) the client is locally authenticated on the service before using it, and (ii) the concrete use of the sub-system requires that the client opens the lock and closes it after use. In this example, one would like to ensure that the execution described in Fig. 2 is correct with respect to these properties. Such verification and the description of the property itself are the main contributions of this article.

These properties can be described by a couple of automatons (Fig. 3), but with a different interpretation of each. The local authentication automaton's (Fig. 3.A) internal state should be maintained in case of service substitution and should be instantiated for each distinct client using the system. In the following, we will call such properties as *Instance-Properties* as they are instantiated on a per object basis; in this case a client. On the contrary, the management of the atomic use of the sub-system (Fig. 3.B) needs to be centralized and shared by all clients. Even if a service is removed and substituted, we would want to keep the current state of the sub-system in memory. In the following, we call such properties *Class-Properties* because its lifetime spans throughout the system's life cycle and is not bound to a particular entity.




**A.** Client-side: instance property




**B.** Interface-side: class property

Figure 3. Example of a property associated to example in Fig. 1

In summary, our proposition is to provide a monitoring framework, which is able to monitor such properties by listening to method calls and OSGi framework events in a dynamic, resilient, and comprehensive manner.

## III. Related Works

The contributions of this article include a monitoring approach for dynamic SOA and the expressiveness of its associated description language. In this section, we discuss separately related works about each of these two parts of our contribution.

### A. Resilience to Dynamicity and Monitoring Comprehensiveness

We propose to 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 two points:

- resilience to dynamicity
- monitoring comprehensiveness

*1) Hard-coding:* In this category, where properties are manually added at source time, we can cite all annotation techniques, like JML [6] and Spec# [7]. 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. 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 that its description is correct.

*2) Soft-Coding:* In this category, where properties are injected at compilation time, or load-time, we can cite Enforcement monitor [8], Larva [5] and JavaMOP [4]. These tools use a standalone description of a property and inject the synthesized monitor inside the code by AspectJ technology.

Advantages of Soft-Coding approach are then the same as in the previous case, but specifying the monitor is easier, since the description of the property is centralized. However, these approaches from Enforcement monitor [8], Larva [5] or JavaMOP [4] are only partially *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 for the whole execution of the class. Indeed, while it is technically possible to use AspectJ to support dynamic class loading and unloading in OSGi, then the monitored bundle must declare the import of the AspectJ library inside its Manifest file — an operation which is not really transparent to the service. Note that this restriction does not exists in Equinoxe implementation of OSGi (Eclipse), but it is because some choices would have been done in the configuration of the framework, requiring to restart the whole framework each time a new service is installed. Furthermore, if monitors need to be started or stopped at runtime it cannot be done directly through AspectJ without restarting the service — something which is undesirable in 24/7 services.

*3) Agnostic-Coding:* In this third category, where the monitor is kept out of the code, we include any trace analyzes approach, such as intrusion detection systems [9] or logging systems [10]. 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 Agnostic-Coding systems can be bypassed, e.g., [9] and [10] can only observe what services accept to push. If a package provides a service without writing sufficient logs, then the monitor does not have sufficient information to check a particular property [11].

*4) Monitoring of Web Services:* There are a number of works (e.g., [12], [13]) that support the monitoring of web services. These provide both dynamicity resilience and comprehensiveness (although these are not explicitly identified as such) by listening to events from a web service composition engine. Furthermore, they also enable properties to be defined both as class properties and instance properties. However, to the best of our knowledge, no similar monitoring techniques have been proposed for the OSGi framework. Moreover, the context is not the same, since in a web service context, we can easily distinguish between callers by their IP address and port number, but it is impossible to know who is the caller, or which class or software is making the call. This can be a considerable restriction in the expression of security policies.

### B. Property description

This part discusses the property description language and focuses mainly on the scope of the property, mainly induced by the location of its associated monitor. Indeed, since we are not in a 1-1 system, we could have many clients using many services at the same time. In such a case, the location of the monitor can change the point of view of the property and hence its expressiveness. Each property can be defined with at least three points of view (eg. Fig. 4): (i) client point of view, (ii) service implementation point of view and (iii) interface point of view.

The proposition made in this work consists in considering all properties as a composition of two parts: a part that handles the client's point of view and a part that handles the interface point of view. In this section, we discuss each of these three possibilities to justify our proposition.

*1) Property Described from Service Side Point of View:* If the designer describes a property with this point of view, shown in Fig. 5, he/she considers the use of a single service [14]. It is easy to consider some behavioral dependence in some parallel uses by multiple clients. However, since we are considering automaton-based properties, it is not obvious how to distinguish between clients within the property. Moreover, it is complex to consider the use of multiple implementations of an interface simultaneously, with potentially some communication between them.

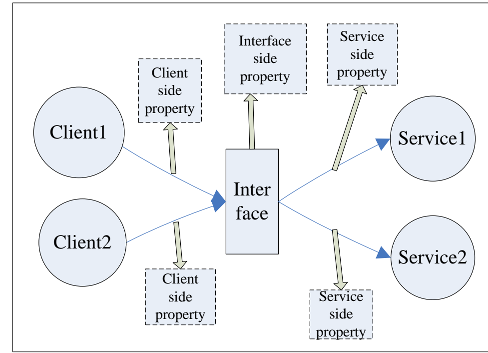



Figure 4. Possible point of view for properties

For the dynamical part, it is not intuitive to describe and use the fact that a new implementation of the same service interface has been loaded on the platform. Moreover, it seems to be complex to share property memory between implementations of the same interface. Hence, if a service is substituted, there is no means of keeping its property in memory, with its internal state, and to map it on another implementation designated to continue the started work.

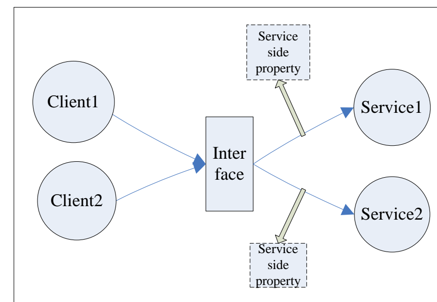



Figure 5. Property description: service implementation point of view

Advantages:

- Simplicity to describe behaviors of each service implementation without the need to make the link with other possible implementations.
- In case of stateful services, with a different memory address space for each implementation, it is very easy to describe the system.

Disadvantages:

- Complexity to describe shared memory between services.
- Impossibility to describe a generic behavior for each client, since we cannot distinguish between clients.

*2) Property Described from Service Interface Point of View:* In this point of view, we consider what can be done through a service interface. It is easy to describe the global use of any implementation of this interface by any client, but not to make distinction between clients or between used implementations.

By its nature, such a property is not directly associated to a service and thus describes a property shared by all implementations. Note that it is easy to consider the loading or unloading of a service implementation, even if it is a substitution, willing to keep the current state of the property.

Since our property description language is automaton-based, the only manner to consider parallel use of many clients is to make some composition between the property and itself. However, such technique leads to a combinatorial explosion of the automaton size. Moreover, it limits the maximum number of clients and services, since we need to have this information to make the composition.

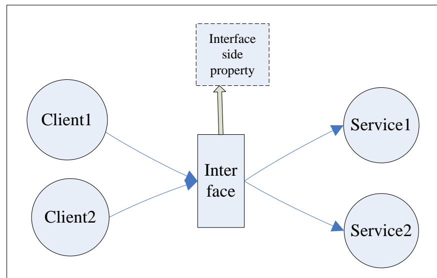



Figure 6. Property description: service interface point of view

Advantages :

- Easy to make a description of the authorized uses, with a global point of view
- Easy to consider loading/unloading of implementations
- Possibility to share a single property state between service implementations

Disadvantages :

- Risk of the shared property description size explosion if we want to describe the concurrent behaviour of several clients.
- Impossibility to describe a generic behavior for each client, since we cannot distinguish between clients

*3) Property Described from Client Point of View:* This third possibility considers that each client has its own instance of the property (Fig. 7). Hence, it is easy to describe the correct use of a service from one client point of view and to consider as many parallel uses as we want, without any combinatorial explosion.

Moreover, it is easy to describe the use of multiple services by a single client and the behavioral dependence in case of concurrent use of services.

In case of substitution of a service, this approach can be resilient, since the property is attached to the client.

However, in case of the simultaneous use of a single service by several clients, if there is some interactions between these usages, it is more complex to describe it.

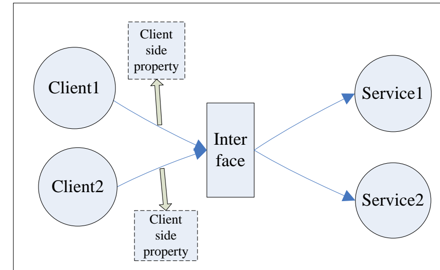



Figure 7. Property description: client point of view

Advantages :

- Easy to make a description of a particular client authorized usages
- Easy to consider loading/unloading of implementations
- Possibility to share a single property state between several service implementations
- No risk of size explosion of the shared property, since it cannot be described

Disadvantages :

- Complexity of describing global behavior including several clients

In this paper, we propose to consider properties as a combination of two kind of properties, associated to two point of views: client and interface. These two parts are respectively called **Instance-Property** and **Class-Property** and are more detailed in section V-B. We propose not to consider the first case (i.e., service point of view), since in typical use of OSGi, if multiple services implement a single interface, the framework favours the use of the same implementation by all clients. Moreover, from our experience we conjecture that properties are typically client side, since an interface property cannot consider the concurrent use of services by many clients without a state explosion. Finally, to have the possibility to add a centralized property, interface properties can be useful to express some shared constraints such as locking/unlocking systems.

In the following, we present the first part of our contribution: the architecture.

## IV. Dynamic-SOA Monitoring Architecture

In the first part of this section, we describe an abstract architecture of a monitoring system supporting specific features of dynamic SOA systems, and we discuss its characteristics. In the second part, we propose a concrete implementation of this architecture under OSGi: OSGiLarva.

### A. Proposition of a Generic Architecture

Our proposition consists of dynamically inserting a monitoring proxy in front of each service, and executing monitors in some autonomous services (Fig. 8). When a service usage event occurs, a notification is sent to each associated monitor, which checks the event against its property.

An interesting advantage of using a dynamic proxy over AspectJ, is that we can start or stop the monitoring of a property without restarting the service. Indeed, since the proxy is bound upon a service request, this can be handled easily, while AspectJ aspects are bound at class load-time, requiring to restart the service.

Since services are treated as black boxes from the running environment's point of view, such an architecture is designed to consider only properties of their external interface. This corresponds to properties expressing the normal authorized use of a service. However, since we are considering dynamic systems, we also want to consider dedicated framework events, such as unregistration of a service or getting a new service. In this approach, we will then focus on behavioral properties.

Since several clients can be running simultaneously within the framework, the scope of properties should not be restricted to the use of a single client. We consider the possibility of adding a monitor in front of several client. By considering both the monitoring of Instance-Properties and Class-Properties, we enable the possibility of simultaneously checking both local as well as global properties on the system.

In order to enable properties expressed in terms of method call events and framework events (requests, registration, unregistration, etc.), we need to capture both kinds of events — the ones between the client and the service, and events from the service registration system. To inject a monitor between a service and a client using it, we adapt the framework in order to make this invisible both to the client and the service. Two interesting characteristics of this approach are that it does not change the binary signature of the service and that neither the service, nor the client, are aware of a potentially running monitor. By adding another proxy in front of the service management system of the framework, we are notified of requests for getting service references.

Fig. 8 describes the abstract architecture. In the following, we delve deeper into our two main principles.

*Resilience to Dynamicity:* Since the monitoring system is externalized in an autonomous service, monitors are separated from the code. When changes occur in the framework, the observation mechanism and its properties 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 can add a proxy between the client and the service to observe their communications. This observation is comprehensive and no communication can bypass this proxy, since neither the client nor the service know each other directly.

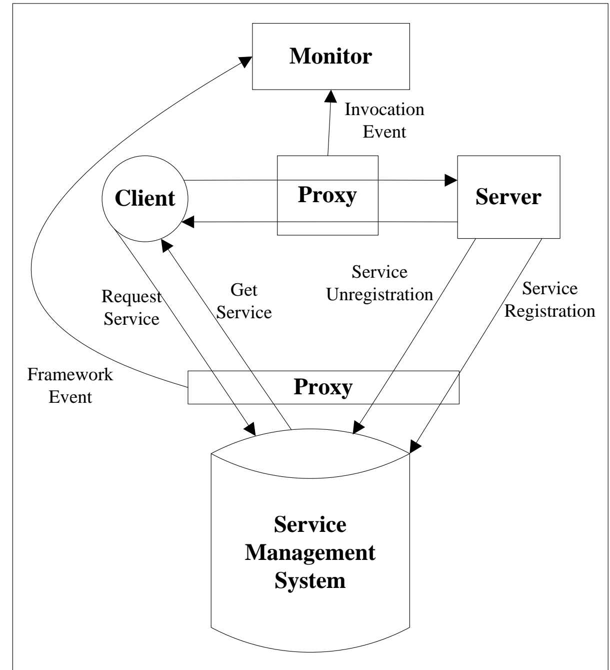



Figure 8. Proposed abstract architecture for monitoring system

### B. OSGiLarva — A monitoring tool for OSGi

OSGiLarva (Fig. 9), is an implementation of the described abstract architecture in the context of the OSGi framework. In our tool, we use Java mechanisms in order to generate a proxy between each client and service. This proxy is dynamically generated from a framework proxy, hooked onto the OSGi framework, and listens to all framework events such as the introduction of a new service or the requesting of a service by a client.

This implementation integrates two existing tools: Larva [5] and LogOs [15]. LogOs is a special logging tool based on the OSGi framework, developed at the CITI Lab during the LISE project [16]. We will use it as a hooking mechanism to observe services' interactions. Larva is a compiler which generates a verification monitor that may be injected into Java code. We use an adaptation of Larva to enable property verification on events reported by LogOs.

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; 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.

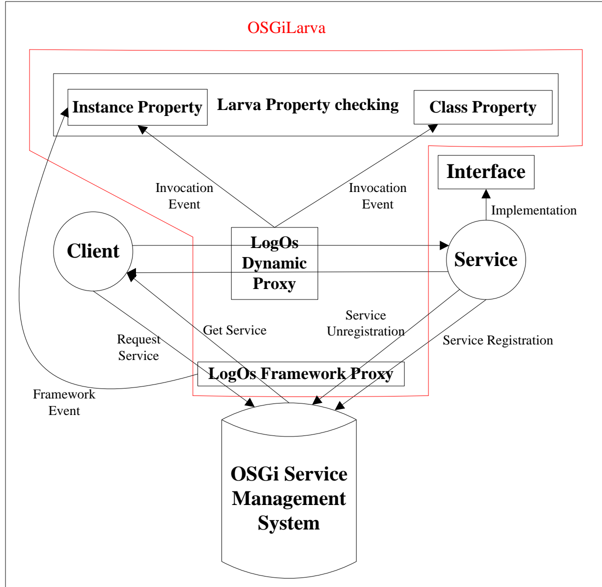



Figure 9. OSGiLarva implementation

*1) LogOs – a Hook to Intercept Services' Interactions:* LogOs is a transparent logging toolkit for the service activity inside the OSGi architecture. As soon as the LogOs bundle is started, each service registration is observed by the system. Thanks to the OSGi hooking mechanism, a LogOs proxy is generated between the service and its consumer. Hence, every method call, including parameters and returned values, are automatically intercepted.

For each event captured by a LogOs proxy, a corresponding LogOs event-description is forged and propagated to LogOs. In our adaptation, LogOs proxy forwards them to the associated monitors.

We have extended LogOs annotations to enable the user to declare whether an interface is to be monitored or not. If an annotation is present, the monitoring class is loaded when a service implementation is registered.

Moreover, LogOs integrates a mechanism to observe services registration, which is originally used to generate service proxy at load-time. This information is sent to the Larva monitor.

*2) Larva — a Monitoring Tool:* Larva is a tool that 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 with the source code. An *aspect* is defined to statically inject some calls to the monitor inside the Java software by using the AspectJ compiler. The Java code translating the property is called each time an expected event occurs.

We adapted Larva to OSGiLarva by removing the part associated with the injection of aspects. In order to replace this part by a call from LogOs, we make the generated Java code from the properties implement an interface provided by LogOs. In order to consider dynamic events in described properties, we introduced some new primitives in the property description language (Section V) corresponding to event descriptions generated by the latest version of LogOs.

*3) Registration of a Service Providing Specification:* We propose to enable the declaration of properties to be monitored to be included as part of OSGi bundles, as shown in Fig. 10. Indeed, an OSGi bundle is an archive providing three elements: a collection of *interfaces*, a collection of *services implementations*, and *bootstrap code*, which is called when loading or unloading the bundle. Thanks to the OSGi architecture, service interfaces, service implementations and bundles may have different life cycles depending on the deployment scheme, since interfaces may be deployed with a bundle other than the one containing the service implementation.

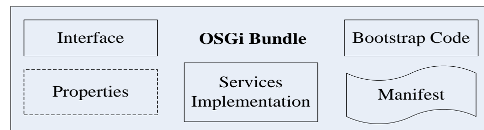



Figure 10. Structure of an OSGi bundle providing properties

As such, we propose to keep the same philosophy when providing properties. We consider that they can be either provided by the same bundle as implementation or by another one. Since interfaces are typing specifications of services and OSGiLarva Class-Properties are behavioral specification of services, it makes sense to map the life cycle of class properties to that of interfaces. On the other hand, the Instance-Properties life cycle describes the behavior of a single service interaction and thus it makes sense to map its life cycle to the client-service connection life cycle.

In next section, we introduce the property description language, which is an adaptation of the one used in Larva.

## V. Introducing Dynamicity in Property Description

The OSGiLarva description language is originally based on the Larva property description language, but we adapt it in order to support more dynamicity. This adaptation is done through two extensions. The first one is the introduction of framework-event primitives in the language. The second one expresses a property as a composition of Class-Properties and Instance-Properties. In this section, we introduce these modifications.

### A. Adding Dynamic Primitives

Larva uses as input a property description language based on automatons, extended by timers, variables and actions.

In the property itself, the user can define the set of symbols used in the automaton. These symbols are events which, in the original version of Larva are defined in terms of method names. We thus propose to add some new primitives in the event definition in order to support framework-event.

A monitor is started when a monitored service is registered in the framework. From this moment, each event related to this service is propagated to this monitor. Since we are in a dynamic framework, dynamic events can occur, e.g., the loading of a second implementation of the same interface, or the un-registration of an existing service. We propose to introduce the three following primitives:

- **REGISTER**: this event occurs when a new service implementation is registered on the framework. It means that a client can now get this service reference at any time. If another implementation is already registered, it shares the same Class-Property.
- **GETSERVICE**: this event occurs when a client is asking for a service, by calling the framework `getService` method. It can lead to two situations: client gets a service or the client does not get any service. If a client could not get a service from the server, it means that there is no registered service corresponding to the client request. For this reason, we introduce the **NOGETSERVICE** event to handle this case.
- **UNREGISTER**: this event occurs when a bundle is unregistered from the framework. It means that, the `stop` method of the bundle has been executed. Used resources are then considered as released. However, if any reference to an instance of the code provided by this bundle still exist, they are now called *Stall references* — meaning that if a client was using this service it has to consider this code as perhaps no longer safe or functional.

In order to generate and provide these events to Larva-monitors, LogOs needs to register some listeners on the framework.

Event **GETSERVICE** is obtained by using an OSGi `FindHook` instance, registered in the OSGi framework. When registered, such object is called each time a service is obtained. Originally, this mechanism was defined in order to make a filter on services obtained as a result of `getService` call. Indeed, the `getService` method accepts as an input a description of the expected service and returns an array of corresponding service implementations among the available ones. The `FindHook` mechanism has been introduced in order to allow service filtering (i.e., to hide some services). Note that LogOs also uses this mechanism to ensure that, if a service is monitored, every calls to this service are necessarily done through a proxy, and never directly.

**REGISTER** and **UNREGISTER** events are obtained by registering an OSGi `EventHook` with the service management system. An object implementing the `EventHook` class and registered in the framework is called each time the service management system observes a modification, such as new incoming service, a service un-registration, or a service property modification.

In each of these cases, an event descriptor is forged by LogOs and sent to the Larva monitor. On its end, Larva treats such events like all other events. Hence, the event descriptor is compared to the list of events the monitor is listening to, and, if the property is expecting this kind of event, it triggers upon it.

### B. Property Description Language

Since our contribution is based on the Larva description language [17], chosen for its closeness to our requirements, we mainly orient our proposition according to Larva. In Larva, properties are described by automatons, where a single script file can contain several automatons. Moreover, Larva provides in its language the possibility of defining parametrized automatons which can be instantiated using event parameters, through the **FOREACH** keyword. We exploit this characteristics in order to use properties composed by two parts (Instance-Property and Class-Property):

- **Instance-Property:** If a property is defined as an Instance-Property, then each time a new client accesses the interface, a new instance of the property is generated and added inside the monitor. When the client terminates, the associated instance of the property can also be removed. Hence, while such properties are still resilient to service implementations' dynamicity, they are intentionally not resilient to clients' dynamicity.
- **Class-Property:** This case corresponds to a centralized property, meaning that several clients using a particular interface will share the same Class-Property. Such property is more resilient to dynamicity since a Class-Property can be kept in memory until the associated interface is unloaded. As such it is not associated to a particular user's interaction or a particular service implementation, and can thus be used, for instance, to express some centralized locking/unlocking mechanisms. However, if several implementations are used concurrently, then they would probably need to be synchronized.

In the following, we present the main principles of the Larva property description language together with small modifications done in the context of OSGiLarva.

*1) Existing Larva Property Description Language:* A Larva property description file can contain several automatons. The file is structured in terms of contexts. The global context can contain several properties and each of them can introduce a new context. A context is defined by variables and listened events. Each inner context can access the global variables.

A **FOREACH** structure allows a property to be instantiated for each different value of an element, considered as an identifier.

Channels can be used by automatons to communicate together. These channel-generated events are broadcasted to the current context and below. So, if two inner contexts need to communicate, they can do it through channels.

A generic structure of a Larva property file is given in Fig. 11. It shows a file containing two properties: a global one and an instantiated one.

```
GLOBAL{
   VARIABLES{ ... }
   EVENTS{ ... }
   PROPERTY P1 {
      STATES{...}
      TRANSITIONS{ ... }
   }
   FOREACH (Object u ){
      VARIABLES{ ... }
      EVENTS{
          %% Property designer needs to
          %% express how to retrieve the
          %% identifier:
          someEvent(User u1) = {
             u1.someMethod(); where u=u1;
          }
          ...
       }
       PROPERTY P2 {
          STATES{...}
          TRANSITIONS{ ... }
      }
   }
}
```

Figure 11. Generic larva property file with two properties of two types

*2) OSGiLarva Properties:* One way of introducing a new context is to use a **FOREACH** clause. This clause is a quantification on an object. Hence, for each instance of a given class, Larva generates a new instance of the inner property. We propose to adapt this structure to our needs, by introducing a new clause: **FOREACHCLIENT**.

In classical Larva, in order to distinguish between users, Larva uses the information given by the caller such as a Session ID passed as a parameter. Hence, Larva only has the same information as the service implementation to check a property. We propose to improve on this by introducing this construct based on the address of the caller. As an example, such a clause could make it possible to check that there is no IDsession spoofing.

A property described in the **FOREACHCLIENT** context will be re-instantiated for each loaded client. It will be the instance-part of the property. Conversely, the class part of the property is instantiated only once and is then shared by all clients. We will then express it in the **GLOBAL** clause. It can communicate with all "instance-part" instances of the property. Fig. 12 shows the global syntax of a global property, composed by an Instance-Property and a Class-Property.

A very important difference between the **FOREACH** and **FOREACHCLIENT** clauses is that the first one is based on values computed inside the **EVENTS** clause from observed parameters, while the second one is based on values provided directly by LogOs observation, without any interpretation of parameters.

Moreover, since **FOREACHCLIENT** is an extension of the **FOREACH** clause, then we keep all language characteristics of the latter.

```
GLOBAL{
    VARIABLES{ ... }
    EVENTS{ ... }
    PROPERTY P1 {
        %% Class property (same as Larva)
        STATES{...}
        TRANSITIONS{ ... }
    }
    %% Introduction of this new keyword
    FOREACHCLIENT(Long pid,String s){
        %% Instance property.
        %% Parameters are:
        %% - pid: client identifier
        %% - s:   name of the client
        %%        (for logs)
        VARIABLES{ ... }
        %% EVENTS clause do not need to
        %% provide method to compute the
        %% identifier. It is intricated
        %% inside the language.
        EVENTS{
            %% Just an event description
            someEvent()=frameworkEvent();
            anotherEvent()=someMethod();
            ...
        }
        PROPERTY P2 {
            STATES{...}
            TRANSITIONS{ ... }
        }
    }
}
```

Figure 12. Introducing the **FOREACHCLIENT** keyword

# VI. Evaluation

In this section, we present some benches of OSGiLarva. There are mainly two implementations used for executing OSGi services: Apache Felix and Eclipse Equinox. In our benches, we use the current Apache Felix which is an open source implementation of the OSGi Release 4 core framework specification, on the top of the Java 1.6.0-06 Virtual Machine. The machine used for these tests runs on an Intel Pentium M at 1.4GHz CPU with 640MB of RAM

and running under Gentoo 4.2.3 with 2.6.22-gentoo-r8 kernel version.

In the following, we are using two examples: one without dynamicity and another with dynamicity. Indeed, since we will make efficiency comparisons against Larva, which does not support dynamicity, we then need to have a static example. This example is just a loop making some calls to a function provided by a service. On the other hand, the dynamic example is very close to the one described in Section II, but with a loop on the client side. This loop specifies the concrete actions from the client and contains a call to a service, followed by an unregistration of the service, a get service to have a second service, a second call, and finally a new registration of the unregistered service. In our benches, we modify the amount of loop iterations to study the variation of the time cost in the long run and its variation due to JIT compilation.

We made three kinds of tests to study performances of OSGiLarva: a comparison between the execution time of OSGiLarva and Larva, a comparison between the execution time of OSGiLarva and OSGi, and a comparison between the execution time of OSGiLarva and a Class-Property-only in OSGiLarva. Indeed, we hypothesized that the identification of the client (and hence the Instance-Property) is a bottleneck, but benches show that it is not so costly.

Here is the definition of some keywords appearing in this section:

- Larva: the time cost from the example with the original Larva system.
- OSGiLarva: the time cost from the example with the OSGiLarva tool.
- WithoutOSGiLarva: the time cost from the example running under OSGi, but without any monitoring system.
- OSGiLarvawithoutPID: the time cost from the example with a weaker version of OSGiLarva where we removed the generation of a caller Id from the system.

Finally, for each test, we made two curve charts. The "Time cost comparison" curve chart shows amount of loop iterations on the horizontal axis, and time cost in milliseconds on the vertical axis. The "Cost ratio" curve chart shows amount of loop iterations on the horizontal axis, and change ratio of time cost in percentage points on the vertical axis. The cost ratio is calculated by the time cost of the example with the monitor divided by the time cost of the example without the monitor.

### *A. Monitoring cost by Using a Proxy (OSGiLarva VS Larva)*

The goal of this test is to evaluate the performance of OSGiLarva (with a proxy) and to compare with the one of the Larva tool (with AspectJ) on the same functions example. Since Larva does not support OSGi dynamicity, we made the comparison on a example without loading of services. In this kind of comparison, we just use the two tools to monitor the normal events from the communication of client using services.

Fig. 13 is a comparison of the time cost in the execution of a static example with Larva and OSGiLarva monitors. We can observe that both curves are very close. Hence, OSGiLarva does not add too much cost by its proxy approach.

In order to be more precise, in Fig. 14 we plot the curves of the cost ratio between Larva and OSGiLarva time cost results. The change ratio of time cost is lower than 1%. This change ratio is from the proxy in OSGiLarva. Thanks to this proxy, OSGiLarva can make the behavioral monitoring bindings dynamic and loosely coupled. The pre-condition of this test is that the monitored service is never replaced by another one. If the monitored service is replaced during runtime, Larva will not be able to detect any of its events. But OSGiLarva can continue to monitor it.

Since these two technologies are not using the same Virtual Machine, the JIT is also not the same. We think that this difference is the explanation for the behaviour observed in the first run, which is stable and always faster on OSGiLarva. This difference is probably also the explanation for diminution of the overhead when the loop is longer.

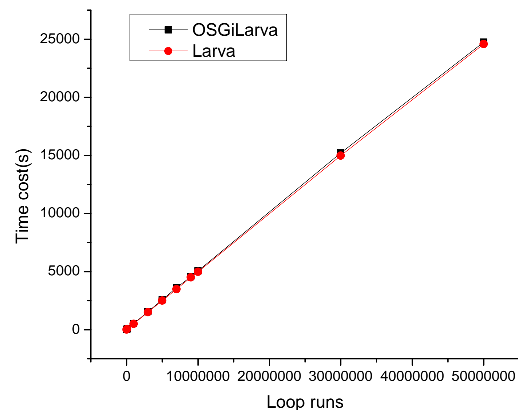



Figure 13. Comparing time cost of a static example with OSGiLarva and Larva

### *B. OSGiLarva Efficiency (OSGi VS OSGiLarva)*

This test runs the dynamic example described as a running example in this article, but with a loop inside the client. We then run it with and without OSGiLarva in an OSGi environment. It aims to evaluate the raw impact of OSGiLarva on service invocation and service events from the framework. The property events includes normal events and framework events.

From Fig. 15, we know that the performance impact of OSGiLarva is stable at around 23% on this example.

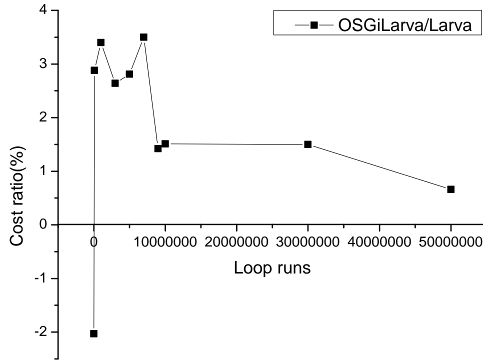



Figure 14. Comparing cost ratio of a static example with OSGiLarva and Larva

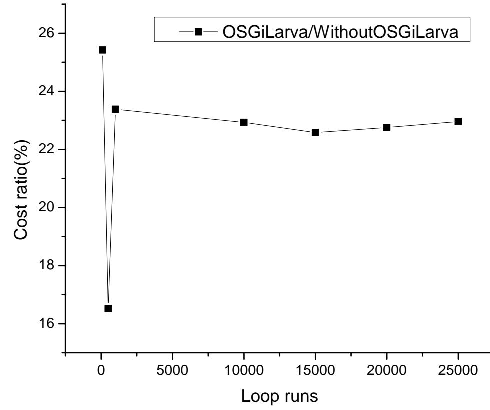



Figure 16. Comparing cost ratio of the case study example with and without OSGiLarva (simple method in service side)

For every monitored service invocation and framework events, OSGiLarva performs its indirection work: it verifies the actions from the original system and computes the current client id, and finally it outputs the monitored traces to the developer or the user at real-time. The cost ratio almost becomes a horizontal line shown in Fig. 16, except for the two first points at about loop 100 runs and 500 runs. We presume that it is the initialization of the JIT which is causing this anomaly.

It is important to note that this 23% overhead is a metric including the call of methods events and the framework events. The biggest part of this overhead is associated to the cost of generating a new proxy and placing it in front of newly requested service.

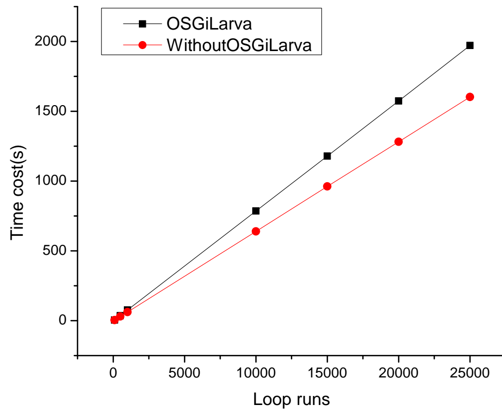



Figure 15. Comparing time cost of the case study example with and without OSGiLarva (simple method in service side)

### C. *Overhead Associated to Getting the Caller Id*

In order to associate each communication to the right client in Instance-Properties, we compute a caller Id. However, we get it through the SecurityManager which is a non-internal way of finding the caller class and caller Id. As such, one would expect extra time costs because of the SecurityManager, warranting further investigation.

Thus, the following test is just for knowing the performance impact from compute current caller Id during runtime. We then compare the cost of the Case Study with and without the Instance-Property and then, with or without getting the caller Id.

From Figs. 17 and 18, we observe that the time cost of the two kind of monitoring are very closed. The impact cost is lower than 5%.

Indeed, in such a simple test example, the body of the called methods are very small. Hence, the most of the time cost is from invocation itself. So, if the service method is a more complex and real one, the time cost for getting caller id and caller name will far less than 5%.

Moreover, even at 5% time cost, we conjecture that it is an acceptable price to pay for obtaining the crucial information for identifying which client is currently using a particular service.

## VII. Conclusions

In the highly dynamic environment of the SOA, where software can be replaced on the fly at runtime, the challenges for ensuring correct behavior increase as the software has to be checked at runtime. In this context, we have identified two properties, that we consider are required to make a dynamic monitor for dynamic SOA systems: (i) *resilience to dynamicity*, i.e., the monitor is able to maintain state even if the service implementation is substituted at runtime, and

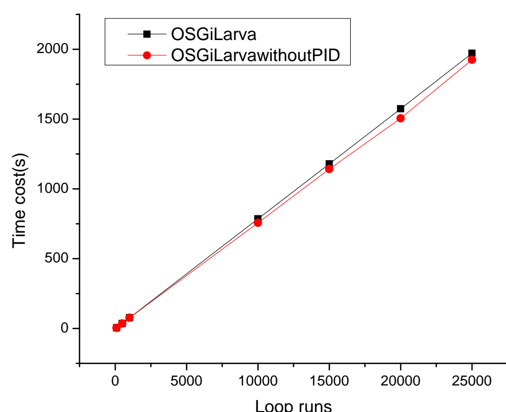



Figure 17. Comparing time cost of the case study example with OSGiLarva but with or without client Id

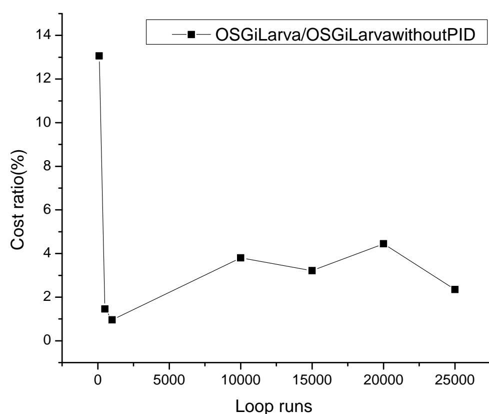



Figure 18. Comparing cost ratio of the case study example with OSGiLarva but with or without Client Id

(ii) *comprehensiveness*, i.e., that no implementation of the service can bypass the monitor's observation.

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. Furthermore, it enables the description of both class properties and instance properties. This feature has been instrumental for OSGiLarva to monitor both properties which span the whole duration of the interface life cycle, and the individual client's point of view of the service, possibly spanning over different implementations of the service requests. We have also extended the Larva event description language, in order to consider not only calls or return of method calls, but also OSGi framework events such as the registration of a service or its request by a client; this has been achieved by introducing reserved event names which are usable transparently as if using standard method calls.

As observed in section VI, our approach is not so inefficient when compared to injection-based monitoring tools like Larva. While our approach is based on an OSGi hook observing all occurring events instead of aspect-oriented programming, the extra cost is small: tending to less than 1% increase in overheads. Since, this approach is crucial for dynamicity resilience, the cost incurred seems to be a reasonable.

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 useful if we want to switch off a monitor or be able to check the binary signature of the code as an authentication credential [18].

Finally, the notion of comprehensiveness also has a number of benefits since anybody with some privileged access to the platform (user, developer, or service) can define a behavioral property and ask the system to check if services respect it. This can be done for many reasons, such as: debugging deployment, privacy concerns, or to learn about typical usage patterns of a service.

## VIII. Future Works

The current implementation of OSGiLarva is not complete with respect to our requirements. For instance, we have some works to do on the deployment step, in order to make it more autonomous. Each Larva property file is associated to a single interface. In the future, we aim to enable the framework to associate one file to possibly several interfaces. Moreover, 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 to separate threads, or we can limit monitoring to only occur within a fixed period of 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 removal of a monitor is straightforward since it is non-intrusive. Similarly, one can consider sampling: monitoring only a random distribution of users, relying on the probability that the error would still occur in the sample.

## References

[1] Y. Dan, N. Stouls, S. Frénot, and C. Colombo, "A Monitoring Approach for Dynamic Service-Oriented Architecture Systems," in *The Fourth International Conferences on Advanced Service Computing*, Nice, France, 2012. [Online]. Available: http://hal.inria.fr/hal-00695830

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

[3] T. Thai and H. Lam, *.Net Framework Essentials*. O'Reilly Media, Incorporated, 2003.

[4] 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.

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

[6] 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.

[7] 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.

[8] T. Jeron, H. Marchand, A. Rollet, Y. Falcone, and O. N. Timo, "Runtime Enforcement of Timed Properties," in *3rd international conference on Runtime Verification (RV)*, Septembre 2012.

[9] 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.

[10] S. Axelsson, U. Lindqvist, and U. Gustafson, "An approach to UNIX security logging," in *21st National Information Systems Security Conference*, 1998, pp. 62–75.

[11] 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.

[12] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini, "Validation of web service compositions," *IET Software*, vol. 1, no. 6, pp. 219–232, 2007.

[13] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti, "Run-time monitoring of instances and classes of web service compositions," in *Proceedings of the IEEE International Conference on Web Services*, ser. ICWS '06. IEEE Computer Society, 2006, pp. 63–71.

[14] Y.-C. Wu and H. C. Jiau, "A monitoring mechanism to support agility in service-based application evolution," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 5, pp. 1–10, Sep. 2012. [Online]. Available: http://doi.acm.org/10.1145/2347696.2347714

[15] S. Frénot and J. Ponge, "LogOS: an Automatic Logging Framework for Service-Oriented Architectures," in *38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Izmir, Turquie, Sep. 2012. [Online]. Available: http://hal.inria.fr/hal-00709534

[16] 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.

[17] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *FMICS*, ser. Lecture Notes in Computer Science, D. D. Cofer and A. Fantechi, Eds., vol. 5596. Springer, 2008, pp. 135–149.

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

View publication stats