A Self-healing Component Sandbox for Untrustworthy Third Party Code Execution

Kiev Gama and Didier Donsez

University of Grenoble, LIG, ADELE team
kiev.gama@imag.fr, didier.donsez@imag.fr

Abstract. This paper presents an architecture and implementation of a self-healing sandbox for the execution of third party code dynamically loaded which may potentially put in risk application stability. By executing code in a fault contained sandbox, no faults are propagated to the trusted part of the application. The sandbox is monitored by a control loop that is able to predict and avoid known types of faults. If the sandbox crashes or hangs, it can be automatically recovered to normal activity without needing to stop the main application. A comparison between an implementation of the sandbox in a domain-based isolation and operating-system based isolation analyses performance overhead, memory footprint and sandbox reboot time in both approaches. The implementation has been tested in a simulation of an RFID and sensor-based application.

Keywords: Fault containment, sandboxing, components, services, autonomic

1 Introduction

Component-based software development allows the construction of applications assembled from software components. Application development may likely involve the integration of different commercial off-the shelf (COTS) components, typically coming from a third-party vendor. This integration implies in testing how the different components will interact as well as trying to detect in advance any incompatibilities or application errors that may arise at runtime. If a component fails[1] during execution, the whole composition that depends on it could fail, and depending on the failure, the whole application may also be down.

Formal methods used in static code analysis are effective ways for testing and detecting errors in scenarios where components involved in a composition are known ahead of application execution. Indeed, there are drawbacks such as the size of software that such approaches are able to analyze (i.e. state explosions in larger software analysis) and the limited amount of people that master these techniques, which are not trivial. If components are not known ahead of execution, the task of integration testing becomes more difficult. Combinatorial explosions may be faced if

[1] A *failure* occurs when a delivered service deviates from correct service state. The deviation is called an *error*, and the hypothesized cause of such deviation is called a *fault*. [2]

we try to predict combinations by validating a component against all possible system configurations [1]. This is something very difficult to achieve in an open COTS market where new components are periodically released. Possible combinations still grow if other components can still be integrated after deployment of the system.

The usage of COTS "as-is" has lead to more error-prone and less dependable[2] applications, hence a recovery oriented approach [3] must be considered to achieve dependability. By acknowledging that hardware fails, that software has bugs and that human operators make mistakes, recovery oriented computing tries to reduce application recovery time (maintainability) thus increasing availability (directly influenced by maintainability), and consequently dependability. Another key factor for providing high availability is to modularize the system so modules can be the unit of failure and replacement [4]. By having well separated modules the application can give the impression of having instantaneous repair. Therefore, with a tiny mean time to repair (MTTR) the failure can be perceived as a delay instead of a failure.

The Java platform dynamic class loading mechanism allows the development of frameworks like SOFA-DCUP [5] and OSGi [6] where components can be loaded at runtime. The OSGi platform is becoming the *de facto* dynamic module system for Java applications allowing to install, to start, to stop, to update and to uninstall components during application execution. A COTS market around OSGi is emerging [7] where third party components are becoming available increasingly, but evaluating their quality and trustworthiness is not a precise task. There are also issues due to the fact of components not being completely isolated from each other. Components that have been incorrectly developed may leave threads still executing or perhaps can still have their objects erroneously referenced after component uninstallation. Component uninstallation is possible in OSGi, but since they are not actually purged from memory these error scenarios are likely to exist [8]. In the long run, applications may accumulate inconsistencies due to dynamicity mishandling.

Fault tolerance and containment are useful for systems that may face unanticipated events at runtime that are difficult or impossible to test during development [9]. By establishing barriers for containment, we can minimize failures impact in the application. If a new component deployed into the system introduces a problem, the application should not stop working. Although sources of errors due to direct memory allocation and handling pointer variables are not present in the Java platform, applications are not free of memory leaks neither completely exempt of other types of faults that may crash or hang the application. Code of poor quality or not exhaustively tested, resource consuming code, and component incompatibilities, among others reasons, may bring a program down or significantly degrade application performance and responsiveness. Importing or wrapping native libraries (e.g., a device driver) in Java applications (and .NET as well) also increases the risk of an eventual crash. This is one of the motivations of our work in the EU funded Aspire project[3]. In one of the scenarios we have an application that collects RFID and sensor data for building reports, eventually using native drivers wrapped as Java components for accessing the

[2] *Dependability*, which encompasses some primary attributes like availability, safety, integrity and maintainability, is defined as the ability to avoid service failures that are more frequent and severe than *acceptable* [2]

[3] http://www.fp7-aspire.eu/

sensors and RFID readers. Sensors and drivers may be plugged at runtime. The reporting application must run non-stop and be able to recover in case of severe faults.

An important point is to provide mechanisms to avoid the propagation of faults from one component to another, so the system can still execute even if one of its components crash. The identification of the faulty component is also an important issue. In the same way, there is a need to automatically react to possible faults and reestablish normal system behavior/execution upon component faults.

Among our motivations is also the possibility of enabling the execution of untrustworthy (but not necessarily malicious) third party code without compromising application stability. An application's core functionality must be separated from untrustworthy third party code (native or not). Minimizing the possibility of error propagation also minimizes application disruption. Microreboots [10] for performing a full reset on an isolated component can actually purge faulty components from memory and bring them back without needing to reset the whole application.

In our precedent work [11] we have enumerated the different approaches for component isolation in the Java platform (namespace based, domain based and process based). By using domain based isolation we have presented a proof of concept that provided fault containment by means of a sandbox for the execution of untrustworthy components. In case of severe failures of components in the sandbox, the main application was not disturbed. In the current paper we extend our previous work by adding the desired automatic reaction to faults and failures. We have introduced autonomic computing principles to our sandboxing architecture, allowing self-healing based on fault prediction (e.g., the application state is not adequate to the application's policy) and fault handling (e.g., recovering from a fault). The paper presented herein has two objectives:

- To perform a comparison of the cost (memory footprint, communication overhead) of the sandboxing approach using two different component isolation approaches, namely domain based and O.S. based (i.e. separate processes).
- To detail the sandboxing architecture autonomic aspects and implementation.

The architecture and its implementation were validated in a controlled experiment of an application that deploys different components in a sensor based application. Although the solution presented here is directly linked with OSGi technology, some of the risks that we point out exist in most of the centralized component based systems. In addition, the self-healing sandbox principle is useful for other dynamic component technologies. The rest of this paper is organized as follows: section 2 gives some motivations and background; section 3 details the architecture and the implementation; section 4 details the experiments and their results. Section 5 presents related work, and section 6 presents conclusions and perspectives of our work.

2 Motivations and Background

The OSGi Service Platform keeps gaining popularity in the development of modular Java applications, with a COTS market [7] that keeps growing. The next subsections provide some background on OSGi and the limitations we try to address, followed by some background on the techniques employed in our solution.

2.1 OSGi and Dynamicity

The OSGi Alliance specifies[4] a framework that allows the dynamic deployment and undeployment of components and services. Applications can take advantage of Java's dynamic class loading feature for updating software components without the need to stop the application. For example, a production system may have a component updated with a new version due to minor bugs fixed or other types of improvements without needing to stop the application for the update.

The deployment unit in OSGi is called bundle, which is an ordinary compressed jar file which contains classes and resources. The jar file manifest contains OSGi specific attributes describing the bundle, which provide metadata and the bundle dependencies (e.g., a list of imported and exported class packages). A bundle can be dynamically loaded or unloaded on the OSGi framework and may optionally provide or consume services, which can be any Java object. Services need to be registered in the OSGi service registry as providers of the specified interface or interfaces.

A bundle needs to have its type dependencies resolved before it can be started. In fact, there are two levels of dependencies: package dependencies (types), and service dependencies (objects). The former are dynamically resolved and necessary for bundle activation, while the latter are also dynamically resolved but not necessary for bundle activation. Compared to the intercomponent dependence types proposed by [12], they can be seen as prerequisites and dynamic dependencies, respectively.

Optionally, an OSGi bundle can provide an *Activator* class, specified in the manifest, that is instantiated and called when the bundle is started. At that moment of startup the activator code can spawn threads and register services in the OSGi service registry. The actual dynamic composition mechanisms rely in a *service-oriented composition* approach. OSGi uses service-oriented principles for providing strong decoupling between components. In a Service Oriented Architecture (SOA) there are the service provider, service requestor and service catalog which in the OSGi framework take the form of a bundle that provides a service, a bundle that requests a service, and the OSGi service registry, respectively.

Different service-based component models (e.g., Declarative Services, iPOJO) have been constructed as layers on top of the OSGi service registry helping to manage the complexity and to minimize the burden of service registration/unregistration that govern the service dependencies and bindings. However, such models are not enough for guaranteeing the mishandling of references. Stale references [8] are a typical problem in OSGi applications when the dynamicity is mishandled. It is characterized by objects provided by the classloader of a bundle that has been stopped or uninstalled; and by references to services that are no longer available, shown in Fig. 1.

The dynamic replacement of components is not complete since we cannot purge the components from memory. This happens, in part, due to the fact that the in OSGi objects have just namespace isolation. This limitation in isolation may also have consequences if a component crashes (e.g., due to an unstable wrapped native driver) the application. In fact, such risk is common to most centralized component based applications that share the same memory space.

---

[4] Different open source implementations exist such as Apache Felix, Equinox and Knopflerfish

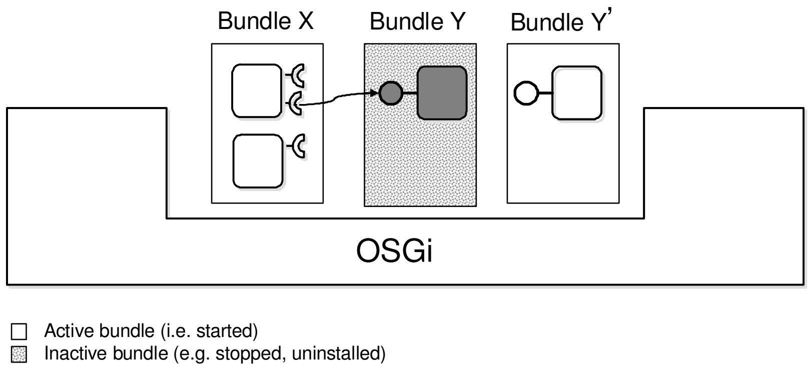
Bundle X
Bundle Y
Bundle Y'
OSGi
□ Active bundle (i.e. started)
▨ Inactive bundle (e.g. stopped, uninstalled)

Fig. 1. A stale reference example. *Bundle Y* has been replaced by *Bundle Y'*, however a service from *Bundle X* keeps referencing an object from an old version of *Bundle Y*.

2.2 Self-Management

Some recent efforts from different research communities in computer science are motivated by the goal of developing applications that can minimize human intervention for system maintenance. One of the motivations to attain is a reduction of costs concerning installation, configuration, tune up and maintenance of software applications. Usually under the self-* flag (self-adaptive, self-healing, etc) these efforts try to provide systems that work autonomously with no human intervention.

IBM has coined the term *autonomic computing*, inspired by the autonomic nervous system, for describing systems that are self-manageable. According to the vision shared by IBM, the four main aspects of autonomic-computing are [13]:

– Self-configuration. Based on high-level policies, the system transparently reacts to internal or external events and adjusts its own configuration automatically.
– Self-optimization**.** The system is able to improve continuously its performance.
– Self-healing. Automatic detection, diagnosis and repair of software and hardware problems.
– Self-protection. Automatic anticipation and reaction of system wide failures due to malicious attacks or cascading failures which were not self-healed.

A *managed element* or *managed resource* consists of hardware (e.g., a processor, an electronic device) or software (e.g., a component, a subsystem, a remote service). A managed element exposes *manageability endpoints* (or *touchpoints)* which provide *sensors* and *effectors* [13]. The sensors provide data (e.g., memory consumption, current load) from the element and the effectors allow performing operations such as reconfiguring. An *autonomic element* consists of one or more managed elements controlled by an *autonomic manager* that accesses the managed elements via their touchpoints. Autonomic managers fall into the four above mentioned self-* categories. Such managers are implemented using an intelligent control loop.

IBM proposes a MAPE-K (Monitor, Analyze, Plan, Execute, Knowledge) control loop (Figure 2) model which is taken as a one of the main references for autonomic control loops. Basically, the control loop monitors data (e.g., the inspection of system performance or current state) from a managed element; interprets them verifying if any changes need to be made; if it is the case, the action needed is planned and executed by accessing the managed element's effectors. *Knowledge* is a representation of live system information (e.g., an architectural model, reified entities) that may be used and updated by any of the MAPE components, thus influencing decision taking.

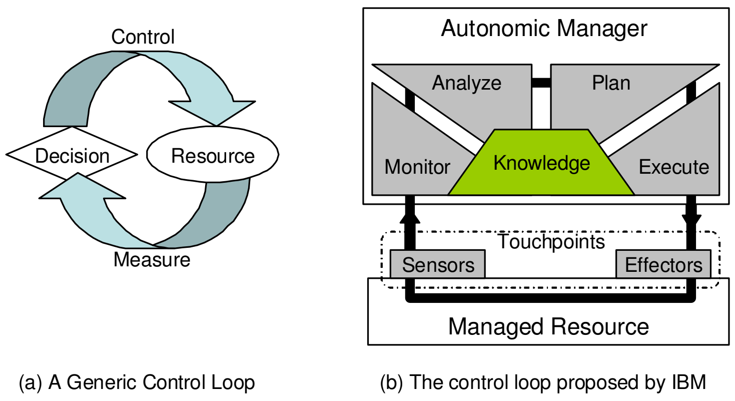

Fig. 2. Illustration of the control loop principle (*a*) and the MAPE-K loop proposed by IBM for autonomic elements (*b*). Figure adapted from [14] and [15], respectively.

An autonomic manager can also have just portions of its control loop automated [15]. Functionalities that are potentially automated could also be under manual supervision (e.g., decision taking upon certain events) of IT professionals. The administrators are also responsible for configuration, which can ideally [16] be done by means of high-level goals, which are usually expressed by means of event-condition-action (ECA) policies, goal policies or utility function policies.

2.3 Microreboots as a Self-recovery Approach

A self-healing system must be able to recover from a failed component by detecting and isolating the failed component, taking it off line, fixing or isolating it, and reintroducing the fixed or replacement component into service without any apparent application disruption [14].

Several studies suggest that many failures can be recovered by rebooting, even when their cause is not known [10]. Normally, hard to identify faults could be caused by diverse sources difficult to track such as race conditions, resource leaks or intermittent hardware error, and reboots are the only solution for reestablishing correct application execution and bring the system back to an acceptable state [17]. This is often the case of day-to-day embedded systems in devices like portable phones or ADSL modems as well as server and desktop applications that may present faults which disappear after rebooting. In [18], the authors show evidence that a significant amount of software errors are a consequence of peak conditions in workload, exception handling and timing. Such errors typically disappear upon software re-execution after clean-up and re-initialization.

The microreboot technique [17] proposes the individual reboot of fine-grained components, achieving the same benefits of an application restart but at lower costs and without losing application availability. OSGi allows microreboots in individual components by performing calls to the `stop` and `start` methods, or even an `update`, which refreshes a component. However, as previously pointed out, the restart of individual components in OSGi may still leave state inconsistencies. A restart of the whole application would be necessary to reestablish its correct state. However a complete shutdown of the application is not desired especially in applications that need to provide high availability.

We are mostly concerned with third party code that would be dynamically loaded and has not been previously tested with the application. Potential faults or errors in the third party code could be fixed by simple microreboots, and if they persist, a full reboot by purging the component out of memory and bringing it back would restore its correct state and behavior. Our goal is to put the third party code in a sandbox which would work as a separate fault contained environment that is independently rebootable. Such isolated code running is a separate boundary would still be able to communicate with the trusted part of the application and vice-versa. These boundaries are fault contained in the sense that crashes or misfunctioning would not be propagated from one boundary to another.

3 Architecture and Implementation

In our precedent work we have provided a proof of concept that validated the isolation approach of our sandbox. However, even if sandbox crashes would do no harm to the trusted application, the sandbox still needed a manual restart by the application administrator. As we consider the sandbox an important point for executing untrustworthy code, we have improved its architecture so it could be constantly monitored, and be able to predict potential faults, as well as recovering itself from crashes or other failures. Our motivation was to provide the sandbox as an autonomic element. An ideal scenario would be having one fault containment boundary per component, but that is an expensive solution in terms of memory footprint with current Java technology. Our approach uses one sandbox only, and one "trustworthy" platform. While the isolated sandbox is being restarted, the system can still provide its functionality in a degraded mode. The next section provides details on the architecture and implementation of the autonomic approach introduced for the sandbox.

3.1 Architecture

An overview of the architecture is depicted in Figure 3, which presents a component diagram containing the main elements for realizing the self-healing sandbox. Each component presented in the diagram is detailed next.

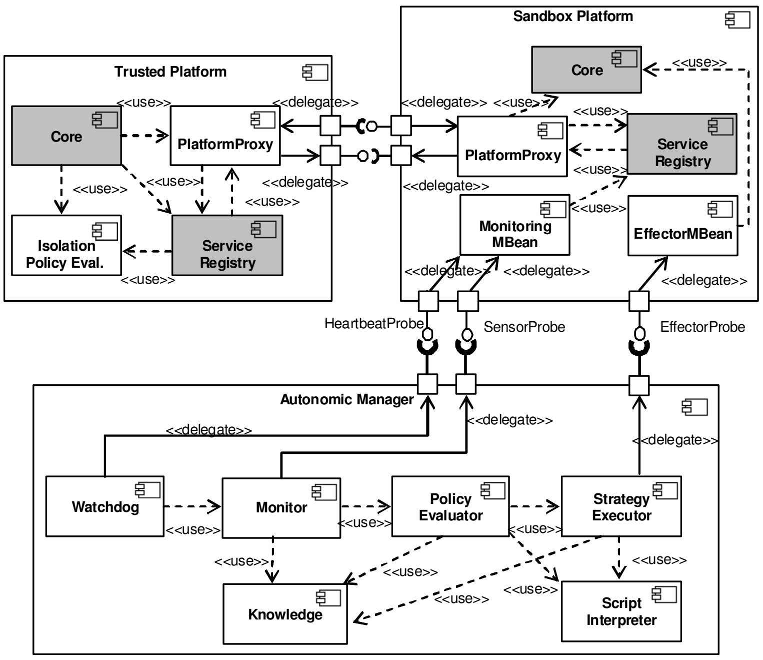

Fig. 3. Component diagram illustrating the interactions of the main components of the trusted platform, sandbox platform and the sandbox autonomic manager. The grey ones are parts of the OSGi framework that were changed, while the white ones are part of our architecture.

Trusted Platform. The *Core* and *Service Registry* correspond to components that are part of the OSGi implementation, which we had to change by adding the code for enabling the sandbox approach, while the other components detailed in Figure 3 concern the improvements that we implement.

Core. The main change in this component concerned bundle life cycle operations (install, start, stop, update, uninstall). The core component exists in both platform, but the aforementioned changes apply only to the trusted platform. With the code that has been added, upon an OSGi bundle installation the core installs that bundle in the trusted platform and also in the sandbox so the same dependencies are present in both platforms. In our initial approach we installed only the necessary dependencies, but the dependency resolution we provided calculated only one level (for A → B, install A and also B). Since dependency transitivity could lead to several levels (A → B → C) and possibly cycles, we decided to simplify and replicate all components and perform also uninstall and update operations, keeping the bundle set synchronized in both platforms. In OSGi, a component installed is not necessarily loaded in memory, therefore it would not imply in loading all bundles in the sandbox container. Before starting up a bundle, the core component verifies with the isolation policy evaluator if the component can be installed in the main platform or if it needs to be installed in the sandbox.

Isolation Policy Evaluator. This component exists only in the trusted platform. It is responsible for verifying the policies for services and component isolation. Two files

describe the respective isolation levels to be attained. The service isolation is used inside the main platform, by adding a proxy layer between service consumer and provider [19]. The component isolation policy provides the rules that indicate which components need to be started in the sandbox. Both files are defined using a simple Domain Specific Language.
Service Registry. This was the other existing component that was changed in order to make the sandboxing work. Upon calls to the `getServiceReference` method, if no match is found in the registry, the call is forwarded (via the *PlatformProxy*) to the sandbox. If the service instance is located in the sandbox, an `IsolatedServiceReference` instance is provided to the caller. For services local to the trusted platform, one extra step is performed by verifying with the Isolation Policy Evaluator if the service instance needs to be proxied, so the service provider is not directly referenced by the consumer code.
PlatformProxy. This component provides a communication layer between the two platforms. Method calls are translated into the appropriate protocol messages so the main platform can perform life cycle operations (install, update, start, stop, uninstall) in isolated bundles; execute queries looking for services in the sandbox service registry; as well as execute the method calls on isolated service instances. In order to avoid services to implement interfaces like `java.rmi.Remote`, so the isolation could be transparent, the protocol used in this layer was custom made.

Sandbox Platform. The sandbox has components that are also present in the trusted platform. Additional components are available, as depicted below, so the control loop of the sandbox autonomic manager can be realized.
Core. In the sandbox platform side, the role of the core component is rather passive. When the sandbox's *PlatformProxy* receives messages concerning bundle life-cycle operations, the calls are delegated to the core component, which does not need any special code or any sort of customization for executing such operations.
Service Registry. The service registry works just like the trusted platform: if no service match is found locally for a service search, the call is forwarded to the trusted platform using the *PlatformProxy*. For both platforms, a flag is set on the service queries received in order to avoid infinite cycles when no service query is satisfied in any of the two platforms. The additional functionality that exists in the sandbox service registry relates mainly to two points concerning the proxies that point to services of the trusted platform. The first point is the logging of service calls so it is possible to identify the number of calls per second on a service, and the second point concerns the invalidation of proxies. If the sandbox has a proxy to a service running in the trusted platform and that service becomes unregistered, the sandbox is notified and the proxy invalidated. By doing this it is possible to throw an exception and identify if a component (and which one) hosted in the sandbox is using a stale service.
PlatformProxy. This component works in the same way as it does in the trusted platform. One difference lies only in its usage: there are no bundle life cycle calls performed. The calls are rather received — since it is the trusted platform that will control the installation of bundles in the sandbox — processed and their response is sent back to the trusted platform. The communication initiated on the sandbox towards the trusted platform only concerns service related calls (service registry queries, service method calls and service registration/unregistration events).

MonitoringMBean. This component is implemented on top of Java Management Extensions (JMX)[5], which is a technology specialized for monitoring Java applications. The Monitoring MBean (Manageable Bean) defined by our architecture can be easily accessed by other Java processes. It provides information concerning CPU consumption, memory usage, number of allocated threads, list of bundles, list of proxied services, service calls per minute (per service basis), stale service count and potential bundles that are retainers of a stale service.
EffectorMBean. Using the same technology as the Monitoring MBean, the effector component makes available a set of operations on the sandbox platform so the autonomic manager can be able to reconfigure it and adapt the application at runtime. Through its interface it is possible to stop the framework (graceful shutdown), to kill the platform, to perform a garbage collection, to invalidate a given service proxy, to stop and to start a given bundle.

Sandbox Autonomic Manager. Our approach is based on the MAPE-K control loop with some simplification by unifying the Analysis and Planning into one component (Policy Evaluator). The autonomic manager is maintained as a separate process that controls the sandbox. Therefore, problems in the sandbox will not interfere in the functioning of its autonomic manager. Its main components are described next.
Watchdog. The watchdog component is responsible for restarting the sandbox platform in case the process is crashed or hung. A process is crashed if its image is no longer in the system, and it is hung if the process image is alive, but the process is not making any progress from a user's point of view [18]. Heartbeat messages are periodically sent to the JVM process and depending on the time taken for the response (or no response) it can be inferred that the process is hung and then the autonomic manager can restart it. If a sudden crash also happens, the watchdog can recover the process and reestablish the connections as well as restart the control loop. The watchdog relies on the `java.lang.Process` API for starting up the sandbox as well as for killing it. The instantiation of the monitor component is made right after the sandbox is launched or restarted.
Monitor. If compared to the watchdog, the monitor component does a much more specialized monitoring. It plays a role in the control loop for collecting information from the managed element (i.e., the sandbox platform), saving pertinent information in the knowledge base and delegating the monitored values to the policy evaluator component. Different types of monitoring readings are done either in push mode (e.g., event for call on invalidated proxy) or in a periodic poll (e.g., memory, CPU, threads, stale service count). When the monitor starts up it logs that information in the knowledge base so it can analyze later how many and how frequently startups have been done.
Knowledge. The knowledge component stores some historical events for later analysis of the other components. It store information such as events of sandbox reboot and reason; lists of offender bundles; trespassing of thresholds (e.g.: memory, CPU). By using such information it is possible to know, for example, if the reboots are too frequent and depending on the custom policy, to change the set of active components in the sandbox based on other logged information.

[5] http://java.sun.com/jmx

Policy Evaluator. The policy evaluator makes use of the scripting engine to interpret the policies. The monitored data is made available to the scripting execution context, so the policy can have access to the current values. If no action is to be taken, the current loop iteration ends. In case of action to be taken, the name of the strategy is passed to the strategy executor.
Strategy Executor. The strategy executor is responsible to load the strategy file, instantiate the scripting engine and execute the strategy. During this process the strategy may gather information (e.g., which bundles are the potential retainers of a stale service) from the knowledge component in order to take a decision. The code outside the application as beanshell scripts gives flexibility and leaves the possibility of customizing the behavior without needing to recompile code.
Script Interpreter. The script interpreter allows the instantiation of a scripting engine which can be used by both the policy and strategy evaluators. The current implementation uses scripts written in Java, interpreted by beanshell[6]. Integration to another Java standardized scripting engine, like Rhino, is transparent to the other components but would require rewriting the policies and the strategies in the target scripting language.

3.2 Domain-based and OS-based Implementations

The proof of concept was implemented using domain-based isolation provided by the Java Isolation API (Java Specification Request-121)[7]. We have used the Multitasking Virtual Machine [20] (MVM) from SunLabs that implements the Isolation API. The JSR-121 presents the notion of *Isolates* which are a first class representation of a strong isolation container with an API to control their lifecycle. The implementation of the sandbox consisted of patching the Apache Felix v.1.4 OSGi implementation. It was made on top of Open Solaris (release 2008.11) using an x86 port of the MVM[8].

Some reasons have motivated us to also implement the sandbox using OS-based component isolation. Although standardized as a JSR, the Isolate API is usually available in Sun Microsystems products which normally have restricted access. The MVM version we have used was not a final product, and also presented instability. JMX functionality did not provide resource monitoring at the Isolate level (e.g., we could not differentiate the memory consumption of the sandbox platform from the trusted platform). It was one of the forces that lead us to implementing the sandbox approach using two different VMs (OS-based isolation) replacing the two Isolates as fault contained entities. This would be a solution reusable in any standard JVM.

Due to JMX limitations on the MVM we could not enable the full autonomic manager using Isolates. The domain-based version of the sandbox only had a stripped down version of the watchdog which directly monitored the connection between the two Isolates (trusted platform and sandbox platform).

The trusted and sandbox platforms needed to exchange messages in an Inter-Process Communication (IPC) fashion. Existing protocols for Java IPC (e.g., RMI, Hessian) rely on extending classes and implementing marker interfaces of such APIs. In order to enable an object to be used with RMI for example, an object must

[6] Beanshell Lightweight Scripting for Java. http://www.beanshell.org/
[7] Application Isolation API Specification. http://jcp.org/en/jsr/detail?id=121
[8] http://mvm.dev.java.net

implement an interface that extends the `java.rmi.Remote` and all methods must throw a `java.rmi.RemoteException`. Since we wanted to transparently enable the sandbox approach, we have implemented a simple protocol for communicating the trusted platform with the sandbox and vice-versa without forcing the source code or bytecode of service objects to be changed. The communication layer on the MVM was made using the Link API which allows two isolates to exchange messages. The multiple JVM approach reused the same protocol but with sockets instead of the link API.

Assumptions. In order to work properly, our approach is realized based on a set of assumptions:

– The set of components that coexist in the trusted platform has already been tested and has a minimal probability of bugs.
– Based on one of the microreboot conditions [10], services that will run on the sandbox are *stateless* otherwise they may have state corruption in case of reboots.
– The communication between platforms will be done through services with simple interfaces (String and primitive values as well as arrays of those types).

Limitations. The solution also has some limitations concerning the functioning of OSGi applications.

– Inter-platform protocol limitation (pass by copy, and method calls using primitive values only), thus restricting the set of services that could exchange messages across isolated platforms.
– No fine grained bundle resource monitoring (needed for a precise monitoring on the autonomic manager). Anyway, it does not exist in regular OSGi frameworks.
– No isolated bundle referencing (e.g., getBundle method will not work on isolated service notifications or on an isolated service reference)

Microreboot behavior and effects.

– Components running in the sandbox are actually purged from memory, since the sandbox platform goes through a shutdown and its isolated container (JVM process or Java Isolate, depending on the approach) is restarted.
– The state of service instances is not maintained (services are stateless as previously assumed). Service providers may use their own mechanisms for that.
– The state of the sandboxed bundles (e.g., started, stopped) is managed by the OSGi platform and it is maintained across reboots.
– The disruption of the communication between the main application and the sandbox causes the communication layer to generate events notifying the departure of the services registered in the sandbox.
– Upon sandbox shutdown, ongoing calls from the main platform towards the sandbox are not able to be completed. No calls remain blocked at all. In that case, the communication layer throws `RuntimeExceptions` for each waiting call. A possible solution would be a strategy for holding the call in the communication proxy and wait for the sandbox restart until the required service interface provider becomes available again and then retry the service call.

4 Evaluation

This section is divided in two parts: 1) a comparison between the sandboxing mechanism using two isolation approaches, namely domain isolation and OS-based isolation; 2) the tests on the sandbox with self-healing characteristics, using the OS-based approach. The experiments were executed on a Pentium 1.7 GHz 2GB RAM running OpenSolaris release 2008.11. Three different Java Virtual Machines were used: Multitask Virtual Machine (a JVM 1.5 implementation that provides the Isolate API); Sun HotSpot Server Virtual Machine versions 1.5.0_21 and 1.6.0_10. Except for the microbenchmark, all experiments were performed in a simulation of an OSGi application for collecting RFID and sensor data with a total of 14 bundles. Sensors and RFID reader simulator components were hosted in the sandbox.

4.1 Domain-based versus OS-based isolation

This subsection compares the two approaches in order to verify what would be the gains, if any, of using domain-based isolation. The following aspects were verified:

- The overhead of method calls across isolation boundaries.
- The memory footprint of OSGi applications using our isolated sandbox
- Sandbox microreboot time

The first measurement consisted on evaluation the communication overhead between the isolated platforms. On the MVM, we have evaluated it in two ways. On the first way, trusted and sandbox platforms were running in the same VM but in different Isolates, thus having domain-isolation. On the second one, we have used two MVM instances like an ordinary JVM (i.e. not using Isolates) so we could use the whole process as a fault-contained boundary, providing us OS-based isolation.

We have used the benchmark suite used in [21] with slight adaptations. The current microbenchmark consisted in measuring the time taken to perform method call from the trusted platform to a service which is isolated in the sandbox. Three methods with different signatures were evaluated: a parameterless method; a method with a String parameter; and a method with an integer array with 128 elements so we could see the impact of parameter serialization and deserialization. All methods were void, so not returning any value. Since RMI is the standard Java Inter-Process protocol, we have benchmarked our approach against it. Table 1 presents the result of our microbenchmark. The experiment data had acceptable precision since each set of measured data had a coefficient of variation (ratio of the standard deviation to the mean) inferior to 1% in most of the cases and rarely over 1%.

The results on the Custom Protocol column group concern the calls on the isolated service running in the sandbox as previously described. The RMI column group results actually did not execute in an OSGi application. We have taken the same interface as the tested service and changed its code to add what was necessary to enable RMI. Then it was tested on two non-OSGi applications (an RMI client and a server, respectively) coded exclusively for the benchmark. The usage of RMI in a non-OSGi application which used 35% less threads than the OSGi application also gives RMI a slight advantage. But it would still be more performing since our protocol was 2 to 3 times slower. Our protocol uses dynamic Java proxies in both ends, which is likely one reason for its low performance comparing to local RMI.

The usage of domain-based isolation concerns only the first result line. The second result line also uses the MVM but in an OS-based fashion. We can notice that two MVM Isolates (domain isolation) perform slightly better than using two MVM instances (OS-based isolation). This is due to the fact of a faster context switching since the Isolates run in the same process (the JVM instance). The third and second result lines performed slightly better which is most likely due to JVM optimizations since they are more recent versions. If running with the JVM configured as interpreted mode (-Xint option), without JIT optimizations, the performance reduction was relatively similar in all cases ranging from 3 to 6 times slower than in the optimized mode (-server option), which is the mode used for collecting the results.

Table 1. Microbenchmark in microseconds (μs) on a void method m with different signatures between isolated platforms. The custom protocol on an OSGi application was benchmarked againt local RMI calls in a non-OSGi application for comparative purposes.

	Custom Protocol (Sandboxed OSGi application)			Local Java RMI (non-OSGi application)		
Combination	m( )	m(String)	m (int[128])	m( )	M(String)	m (int[128])
MVM (2 Isol.)	178.72	225.22	277.56	75.68	80.93	103.36
2 x MVM 1.5	182.74	231.23	284.49	82.19	87.62	110.33
2 x JVM 1.5	162.58	203.71	241.39	63.58	67.40	87.14
2 x JVM 1.6	129.12	161.49	190.67	53.46	55.24	66.83

Another comparison we have performed concerned memory footprint, as shown in Figure 4. We have used the Solaris `pmap` command for verifying the resident and private memory of the tested combinations. The experiment consisted of measuring the total footprint of the OSGi test application (trusted platform + sandbox platform). In the OS-based approach used with two JVMs 1.5 and two JVMs 1.6 we have added the footprint of each JVM. In the case of domain-based approach a single MVM instance contained both OSGi platforms. The resident memory of the MVM running two isolates was inferior to the sum of sandbox and trusted platform running on the JVM 1.5. However, the two JVM 1.6 together performed with less footprint. If we consider just private memory the MVM performs better than the other ones.

The third and last comparison made consisted on the time taken to perform application startup and a sandbox microreboot. Although we did not use a full autonomic manager on the domain-based approach for this experiment, we could provide a watchdog that is able to restart the sandbox in case of crashes. Table 2 presents the time taken in each VM combination. By using Isolates we can significantly reduce the mean time to repair of the sandbox. The major difference is probably because the watchdog monitors directly the Link objects that are responsible for the communication of the two platforms. Since the watchdog resides in the same process, the crash detection is immediate upon the disruption of the Link object.

Based on these experiments we can verify that the main advantage of using domain-based isolation over an OS-based isolation implementation of our sandbox approach concerns the application startup time and, especially, sandbox microreboot time. The memory footprint (resident memory) differences were not very significant, at least for the evaluated application. Communication overhead across process boundaries is minimized in more recent and optimized JVM versions. Therefore, an OS-based approach seems to be a reasonable option for the realization of the sandbox.

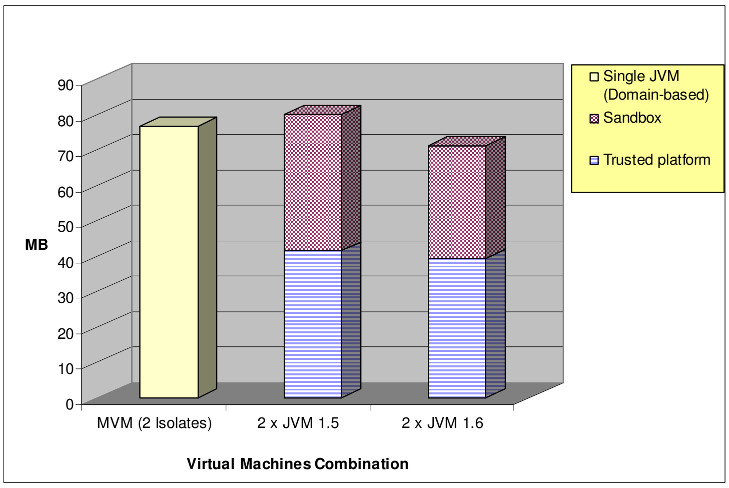

Fig. 4. Resident memory footprint of sandbox solution using different VM combinations. The one on the left uses domain-based isolation while the other two use OS-based isolation.

Table 2. Average startup time and sandbox reboot time in milliseconds

Combination	Application Startup time (ms)	Sandbox Crash detection time (ms)	Sandbox Reboot time (ms)
MVM (2 Isolates)	3186	32	303
2 x MVM 1.5	3449	697	3064
2 x JVM 1.5	3945	660	3047
2 x JVM 1.6	3859	658	2537

4.2 Autonomic Manager Validation

The validation of the autonomic manager using the OS-based approach consisted in simulating scenarios where the addressed errors would occur. It was necessary to use a technique for fault injection. The behavior of systems tested with faults injected in the interface level (e.g., passing invalid parameters) significantly differs when faults are injected in the component level (e.g. emulation of internal component errors), not representing actual application usage [22]. For testing the autonomic manager, we rather focused on test cases resembling component fault injection that could reflect possible faults in a realistic scenario. In our case, the term *fault deployment* would be more appropriate, since the dynamic platform allows components to be deployed and started at runtime.

The faulty components introduced faults for overutilization of CPU; excessive memory allocation; excessive thread instantiation; excessive invocation of services

(Denial of Service); stale reference retainer and application crash. Observations on the system lead to findings such as the CPU information retrieved via the Java API not reflecting actual CPU usage verified via Solaris' `top` command. Also, in cases of overutilization of memory the watchdog would kill the sandbox before the memory rule could take action. This would happen in occasions where the watchdog is configured with a low value for unresponsiveness time.

The fault prediction approach we have used is rather pragmatic, and implemented as policy scripts interpreted by the policy evaluator component. The case of stale reference retainers is the one that involves more details on implementing the policy, being far from trivial. The scripts for fault detection concerning the other deviating states are straightforward and can be directly applied after reading the monitored data in most cases, needing to extract minor information from the knowledge component. The faulty bundles utilized were yet simplistic, and more robust validation is needed.

The microreboots triggered by the autonomic manager would restore the sandbox to its initial state, since a restart is performed. As previously described in our assumptions, we consider that services are stateless so the microreboot strategy can work without corrupting service state. Except if we replace a faulty component by another one which provides a correction for the detected fault, a microreboot cannot guarantee to permanently remove a fault. However, considering the software rejuvenation approach [23], which is one of the influences on the microreboot technique, intermittent errors are likely to disappear after an application or component restart (in this case, a sandbox restart). Due to limitations on resource monitoring at the component (bundle) level, it is difficult to identify the bundle that is the source of some errors such as memory consumption. In cases where a component that causes abnormal execution is identified (e.g., stale reference retainer), it is possible (though not yet implemented), to provide reactive code to indentify the misbehaving component with the help of information stored on the knowledge component and then stop it. Like so, replacing the component with an appropriate or alternative version would also be possible if implementing a search mechanism accessing and using metadata of OSGi Bundle Repositories (OBR[9]), which are federated bundle repositories described by XML files. In order to have such bundle replacement mechanism one would need to query an OBR using its capability metadata (e.g., metadata about provided packages and services) as a filter for finding a bundle that would provide the same services as the faulty bundle does.

5 Related Work

A survey on self-healing systems [24] presents several considerations and existing approaches for developing such types of systems. It highlights different strategies for maintaining system health. Among those strategies we have find in our techniques: system maintaining by probing in a feedback control loop; isolation of faulty components; performance monitoring; prediction of events; and system rejuvenation. The technique of having a watchdog process [18] for monitoring if an application

[9] OSGi standard based on the OSCAR Bundle Repository (http://oscar-osgi.sourceforge.net/)

hung or crashed is also proposed as a fault tolerant technique [25] for third-party components, as well as the microreboot technique.

The Rainbow [27] framework uses software architectures to perform the self-adaptation of systems. Rainbow's control loop uses an abstract architectural model to monitor a system's runtime properties. If any constraint violation is detected in the architecture model, the control loop performs global-level and module-level adaptations on the running system. The JADE framework [26] uses an architecture-based approach for autonomic distributed applications, specifically JavaEE clusters. Among other employed techniques, JADE uses a component-based design on top of the Fractal component model, a control loop for monitoring the managed elements, and system replication for fault tolerance. Failed nodes are detected using a heartbeat technique. Its architecture-based approach is realized by means of a representation of the managed system (system representation) that is synchronized with the running system. Changes on the representation are mirrored to the system and vice-versa.

Different efforts can be found [28, 29] using the OSGi platform targeting autonomic computing, but they are usually focused on the dynamicity and flexibility of the platform, not addressing dependability aspects. We can find an experimental solution [30] that also performs custom adaptations in the Apache Felix OSGi implementation. They try to introduce into the OSGi framework some autonomic computing concepts, also based on control loops. However, the work focuses on self-configuration, not mentioning any strategies for handling faults or application crashes. There are some assumptions concerning resource usage that are true only in limited scenarios. One of the perspectives of the work is to employ more efficient isolation techniques.

An adaptive fault management approach targeting the Robocop framework [31, 32] concentrates efforts on mechanisms for fault detection and failure repair. They use service instances wrappers for intercepting calls to the underlying service and perform model-based verifications on the incoming calls in order to check if there is any deviation from the service specification. If it is the case, then the appropriate repair action is taken using the adequate repair rule. This mechanism works in an adaptive fashion: a sort of knowledge base stores the repair rules taken for given services. Future repairs may use that information to choose the best repair action for a given case, or generate a new repair rule if the stored ones are not appropriate.

Concerning other isolation mechanisms related to ours, we also can cite Microsoft technologies such as COM (Component Object Model) components which can be either loaded in the client application process or provided in an isolated process [33]. In the latter case, a surrogate process (dllhost.exe) can load the DLL and act as a server. It enables fault isolation between the client and the component server, but the inter-process communication between them introduces a performance overhead. A similar approach can be done in .NET platform, the successor to COM technology, by using Application Domains. .NET allow the dynamic loading of classes similar to the Java technology, but the unloading remains limited [34] thus restricting the usage of .NET for the creation of a dynamic platform such as OSGi.

The .NET framework 4.0, provides the Managed Add-In Framework (MAF) [35] which is a programming model allowing to create and to host add-ins, typically third party code that needs to be used without compromising the host application stability. To achieve that, the MAF allows an add-in to be hosted in a separate Application

Domain or in a separate process. A MAF's architecture comprises a pipeline of seven assemblies (Host, Host View, Host Adapter, Contract, Add-in Adapter, Add-in View, Add-in) which need to be provided if an add-in is to be used. Although they provide a robust approach with isolation in mind for loading and using third-party code, realizing managed add-ins is overly complex considering the number of assemblies to be provided and maintained if compared to the transparent approach we propose.

We use very similar principles to those of R-OSGi [36], which proposes a protocol for communicating remote OSGi services. It uses dynamic bytecode generation for service consumer proxy code which is loaded as a bundle into the client platform, which significantly increases the number of executing bundles. Their motivation is also to provide a transparent communication mechanism but with distribution in mind, however R-OSGi client code is not completely unaware of distribution.

6 Conclusions and Perspectives

In this paper we have presented an autonomic approach of a sandbox for the execution of third party components. Third party code such as wrapped native code or untested components may put the application in risk. Our approach allows the execution of such components outside the main application memory space, in a sort of sandbox which provides fault containment without disturbing the trusted part of the application. The sandbox has been validated using two different types of isolation: domain-based and operating system-based. The former has been developed using Java Isolates on a Multi-tasking Virtual Machine, and the latter has been implemented using two Java Virtual Machine instances. A comparison between our implementation using the two approaches shows that the main advantage of domain-based isolation over OS-based isolation concerns the application startup and especially the sandbox reboot time. Memory gains of the domain-based approach are not that significant if we take into account more recent and optimized Virtual Machines. An evaluation of the proposed transparent communication protocol shows that it is 2 to 3 times slower than local RMI, mostly due to the utilization of Java dynamic proxies on both ends.

The autonomic manager of the sandbox has been implemented only in the OS-based isolation approach. A control loop for monitoring the sandbox identifies any deviation in system state based on directives defined as scripts which are dynamically loaded by the autonomic manager. The infrasctructure allows to restart individual components or depending on the state and directive, a full reboot on the sandbox. This approach has been tested in a simulation of an RFID and sensor-based application.

The correct functioning of the sandbox approach is based on a set of assumptions which may not apply to all types of applications. Another issue that can be pointed out is that the main platform could also be an autonomic element. We plan to transform the sandbox autonomic manager into an OSGi-based application so we could add dynamicity and extensibility to other components. Currently the only flexible part of the autonomic manager is the scripting for rules execution. Of course, the problematic described here of executing third party code would not be the case of this autonomic manager, since no third party code would be involved. Our ultimate goal is to automatically "promote" a component from the sandbox to the trusted platform based on an analysis of a component's history in the knowledge base. Yet, the promotion is possible but as a manual process done by the application administrator. For now we do not consider that we have enough fine grained information for taking such decision at runtime.

Acknowledgements. The authors thank the anonymous reviewers for their suggestions that helped improving the final version of this paper; Laurent Daynès, from Sun Microsystems, for the advices on the MVM, the discussions about Isolates and domain isolation; and Johann Bourcier, from our research team, for the debates and insights on autonomic computing. Part of this work has been carried out in the scope of the ASPIRE project (http://www.fp7-aspire.eu), co-funded by the European Commission in the scope of the FP7 programme under contract 215417. The authors acknowledge help and contributions from all partners of the project.

References

1. Szyperski, C, Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, second edition (2002)
2. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Secur. Comput. Vol. 1, number 1, 11--33 (2004)
3. Fox, A., Patterson, D.: Guest Editors' Introduction: Approaches to Recovery-Oriented Computing. IEEE Internet Computing, vol. 9, no. 2, 14--16 (2005)
4. Gray, J: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems, pp. 3--12. (1986)
5. Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: architecture for component trading and dynamic updating. In: 4th Intl. Conf. on Configurable Distributed Systems, pp.43--51 (1998)
6. OSGi Alliance, http://www.osgi.org/
7. OSGi Alliance. About the OSGi Service Platform, Technical Whitepaper Revision 4.1, 7 June 2007, http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf (2007)
8. Gama, K., Donsez, D.: A Practical Approach for Finding Stale References in a Dynamic Service Platform. In: CBSE 2008. LNCS, vol. 5282, pp. 246--261. Springer Berlin/Heidelberg (2008)
9. Tian, J.: Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement. Wiley-IEEE Computer Society Press (2005)
10. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot — A technique for cheap recovery. In: 6th Conference on Symposium on Operating Systems Design & Implementation (2004)
11. Gama, K., Donsez, D. Towards Dynamic Component Isolation in a Service Oriented Platform. In: CBSE 2009. LNCS, vol. 5582, pp. 104--120. Springer Berlin/Heidelberg (2009)
12. Kon, F. and Campbell, R. H.: Dependence Management in Component-Based Distributed Systems. IEEE Concurrency 8, 1, 26--36 (2000),
13. Kephart, J., Chess, D. The Vision of Autonomic Computing, *Computer,* vol. 36, 41--50, (2003)
14. Ganek, A.G., Korbi, T.A.: The Dawning of the Autonomic Computing Era. IBM Systems Journal, vol. 42, no. 1, 5--18 (2003).
15. IBM. An architectural blueprint for autonomic computing. Autonomic computing whitepaper, 4th edition. (2006)

16. Huebscher, M., McCann, J.: A survey of autonomic computing—degrees, models, and applications. ACM Computing Survey, 40(3):1--28 (2008)
17. Candea, G., Kiciman, E., Kawamoto, S., Fox, A.: Autonomous recovery in componentized Internet applications. Cluster Computing 9, 2, pp. 175--190 (2006)
18. Huang, Y., Kintala, C.: Software Fault Tolerance in the Application Layer. Software Fault Tolerance, John Wiley (1995)
19. Gama, K., Rudametkin, W., Donsez, D.: Using Fail-stop Proxies for Enhancing Services Isolation in the OSGi Service Platform. In: MW4SOC'08, pp.7--12, ACM, NY (2008)
20. Czajkowski, G., Daynès, L.: Multitasking without Compromise: a Virtual Machine Evolution. In: 16th conference on Object-oriented programming, systems, languages, and applications (OOPSLA), pp 125--138, New York, USA (2001)
21. Seinturier, L., Pessemier, N., Escoffier, C., Donsez, D.: Towards a Reference Model for Implementing the Fractal Specifications for Java and the .NET Platform. In 5th Fractal Workshop at ECOOP'06 (2006)
22. Moraes, R., Barbosa, R., Duraes, J., Mendes, N., Martins, E., Madeira, H.: Injection of faults at component interfaces and inside the component code: are they equivalent? In: European Dependable Computing Conference, EDCC '06, pp.53--64 (2006)
23. Huang, Y., Kintala, C. M. R, Kolettis, N., Fulton, N. D.: Software Rejuvenation: Analysis, Module and Applications. In: 25th international Symposium on Fault-Tolerant Computing. (1995)
24. Ghosh, D., Sharman, R., Rao, H. R., Upadhyaya, S.: Self-healing systems survey and synthesis. Decision Support Systems 42(4):2164--2185 (2007)
25. Li, J., Chen, X., Huang, G., Mei, H., Chauvel, F.: Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support. In: CBSE 2009. LNCS, vol. 5582, pp. 69--86. Springer Berlin/Heidelberg (2009)
26. Bouchenak, S., Boyer, F., Krakowiak, S., Hagimont, D., Mos, A., Jean-Bernard, S., Palma, N. d., Quema, V.: Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters. In: 24th IEEE Symposium on Reliable Distributed Systems. IEEE Computer Society, Washington (2005)
27. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer, vol. 37, no. 10, 2004, pp. 46--54. FTCS. IEEE Computer Society, Washington, DC (1995)
28. Bottaro, A., Bourcier, J., Escoffier, C., Lalanda, P.: Autonomic Context-Aware Service Composition. In: IEEE International Conference on Pervasive Services, pp. 223--231(2007)
29. Diaconescu, A., Maurel, Y., Lalanda, P.: Autonomic Management via Dynamic Combinations of Reusable Strategies. In: 2nd International Conference on Autonomic Computing and Communication Systems (2008)
30. Ferreira, J. Leitao, J., Rodrigues, L.: A-OSGi: A framework to support the construction of autonomic OSGi-based applications. In: Autonomics 2009, Cyprus (2009)
31. Su, R., Chaudron, M.R.V., Lukkien, J.J.: Runtime failure detection and adaptive repair for fault-tolerant component-based applications. In: Software Engineering of Fault Tolerant Software Systems. 230--255, World Scientific Publishing. (2007)
32. Su, R., Chaudron, M.R.V.: Self-adjusting Component-Based Fault Management. In: 32nd EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE Computer Society, Washington, DC, pp. 118--125 (2006)
33. Lowy, J.: COM and .NET Component Services. 1st. O'Reilly & Associates, Inc. (2001)
34. Escoffier, C., Donsez, D., Hall, R. S.: Developing an OSGi-like service platform for .NET. In: Consumer Comm. and Networking Conf., pp. 213--217, USA (2006)
35. Nagel, C., Evjen, B., Glynn, J., Watson, K., Skinner, M.: Professional C# 4 and .NET 4. Wiley Publishing (2010)
36. Rellermeyer, J. S., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications through Software Modularization. In: 8th Intl ACM/IFIP/USENIX Middleware Conference (2007)