ScapeGoat: Spotting abnormal resource usage in component-based reconfigurable software systems

Inti Gonzalez-Herrera, Johann Bourcier, Erwan Daubert, Walter Rudametkin, Olivier Barais, François Fouquet, Jean-Marc Jézéquel, Benoit Baudry

ScapeGoat: Spotting Abnormal Resource Usage in Component-based Reconfigurable Software Systems

I. Gonzalez-Herrera[a,**], J. Bourcier[a,*], E. Daubert[a,d], W. Rudametkin[b], O. Barais[a,*], F. Fouquet[c], J.M. Jézéquel[a], B. Baudry[e]

[a] *University of Rennes 1 - IRISA, Campus de Beaulieu, 35042 Rennes, France*
[b] *University of Lille 1, Cite Scientifique, 59655 Villeneuve d'Ascq Cedex, France*
[c] *University of Luxembourg - Security and Trust Lab, 1457 Luxembourg, Luxembourg*
[d] *CTO Energiency, 2 rue de la Châtaigneraie, 35510 Cesson-Sévigné, France*
[e] *INRIA, Campus de Beaulieu, 35042 Rennes, France*

Abstract

Modern component frameworks support continuous deployment and simultaneous execution of multiple software components on top of the same virtual machine. However, isolation between the various components is limited. A faulty version of any one of the software components can compromise the whole system by consuming all available resources. In this paper, we address the problem of efficiently identifying faulty software components running simultaneously in a single virtual machine. Current solutions that perform permanent and extensive monitoring to detect anomalies induce high overhead on the system, and can, by themselves, make the system unstable. In this paper we present an optimistic adaptive monitoring system to determine the faulty components of an application. Suspected components are finely analyzed by the monitoring system, but only when required. Unsuspected components are left untouched and execute normally. Thus, we perform localized *just-in-time* monitoring that decreases the accumulated overhead of the monitoring system. We evaluate our approach on two case studies against a state-of-the-art monitoring system and show that our technique correctly detects faulty components, while reducing overhead by an average of 93%.

Keywords: Resource, Monitoring, Adaptation, Component, Models@Run.Time

*Corresponding author
**Principal corresponding author
Email addresses: inti.gonzalez_herrera@irisa.fr (I. Gonzalez-Herrera), johann.bourcier@irisa.fr (J. Bourcier), erwan.daubert@irisa.fr (E. Daubert), walter.rudametkin@inria.fr (W. Rudametkin), barais@irisa.fr (O. Barais), francois.fouquet@uni.lu (F. Fouquet), jezequel@irisa.fr (J.M. Jézéquel), benoit.baudry@inria.fr (B. Baudry)

1. Introduction

Many modern computing systems require precise performance monitoring to deliver satisfying end user services. For example, in pervasive, ubiquitous, building management or Internet of Things systems, devices that are part of the system have limited resources. The precise exploitation of these limited resources is important to take the most out of these systems, therefore requiring a fine and dynamic resource monitoring. Many other examples exist in the area of cloud computing where the notion of elasticity and on demand deployment is key to enable dynamic adaptation to user demand [1]. Indeed, the elasticity management mechanism in charge of allocating and shrinking computing resources to match user demand, requires a precise performance monitoring of the application to determine when to increase or decrease the amount of allocated resources. Another example can be found in the cloud computing domain at the Software as a Service level when services with various Quality of Services (QoS) are offered to end users. The differentiation of QoS between two users of the same services requires a precise performance monitoring of the system to cope with the specified QoS.

To implement such behavior, these modern computing systems must exhibit new properties, such as the dynamic adaptation of the system to its environment [2] and the adaptation of the system to available resources [3]. Modern Component based frameworks have been designed to ease the developers' tasks of assembling, deploying and adapting a distributed system. By providing introspection, reconfiguration, advanced technical services, among other facilities [4], component frameworks are good candidate to assist software developers in developing resource-aware system. These frameworks provide extensible middleware and assist developers in managing technical issues such as security, transaction management, or distributed computing. They also support the simultaneous execution of multiple software components on the same virtual machine [5–7].

Current monitoring mechanisms [8–10] continuously interact with the monitored application to obtain precise data regarding the amount of memory and I/O used, the time spent executing a particular component or the number of call to a particular interface. Despite their precision, these monitoring mechanisms induce high overhead on the application; this prevents their use in production environments. The overhead of a monitoring mechanism can be up to a factor of 4.3 as shown in the results presented in [11]. As we discuss in Section 4 the performance overhead grows with the size of the monitored software. Thus, overhead greatly limits the scalability and usage of monitoring mechanisms.

In this paper, we address excessive overhead in monitoring approaches by introducing an optimistic adaptive monitoring system that provides lightweight global monitoring under normal conditions, and precise and localized monitoring when problems are detected. Although our approach reduces the accumulated amount of overhead in the system, it also introduces a delay in finding the source of a faulty behaviour. Our objective is to provide an acceptable trade-off between the overhead and the delay to identify the source of faulty behaviour in the system. Our approach targets component-models written in object-oriented languages, and it is only able to monitor the resource consumption of components running atop a single execution environment. In this paper,

we discuss how we can leverage our proposal to provide the foundations for resource consumption monitoring in distributed environments.

Our optimistic adaptive monitoring system is based on the following principles:

- **Contract-based resource usage.** The monitoring system follows component-based software engineering principles. Each component is augmented with a contract that specifies their expected or previously calculated resource usage [12]. The contracts specify how a component uses memory, I/O and CPU resources.
- **Localized just-in-time injection and activation of monitoring probes.** Under normal conditions our monitoring system performs a lightweight global monitoring of the system. When a problem is detected at the global level, our system activates local monitoring probes on specific components in order to identify the source of the faulty behaviour. The probes are specifically synthesized according to the component's contract to limit their overhead. Thus, only the required data are monitored (e.g., only memory usage is monitored when a memory problem is detected), and only when needed.
- **Heuristic-guided search of the problem source.** We use a heuristic to reduce the delay of locating a faulty component while maintaining an acceptable overhead. This heuristic is used to inject and activate monitoring probes on the suspected components. However, overhead and latency in finding the faulty component are greatly impacted by the precision of the heuristic. A heuristic that quickly locates faulty components will reduce both delays and the accumulated overhead of the monitoring system. We propose using Models@run.time techniques in order to build an efficient heuristic.

The evaluation of our optimistic adaptive monitoring system shows that, in comparison to other state-of-the-art approaches, the overhead of the monitoring system is reduced by up to 93%. Regarding latency, our heuristic reduces the delay to identify the faulty component when changing from global, lightweight monitoring to localized, just-in-time monitoring. We also present a use case to highlight the possibility of using Scapegoat on a real application, that shows how to automatically find buggy components on a scalable modular web application.

An early version of the Scapegoat monitoring framework is presented in [13]. This paper introduces four new majors contributions:

- The paper includes **a new mechanism to monitor memory consumption that can be turned on/off**. In [13], memory monitoring cannot be turned off. As a consequence, the probes used to account for memory consumption are always activated. This impacts the performance of the system even when in global monitoring mode. In this paper, we propose a mechanism to avoid any kind of overhead when in global monitoring mode thank to the new monitoring mechanism. Using this new mechanism we reduce even more the performance overhead in terms of CPU consumption and we avoid overhead in terms of memory consumption.
- New **experiments to assess the performance impact** of the proposed mechanism **to compute memory consumption monitoring**.

- In addition to the experiments proposed in [13], a **new use case is used to evaluate the monitoring framework**.
- We show how to generalize the approach to deal with properties other than CPU, memory and related resources.

The remainder of this paper is organized as follows. Section 2 presents the background on Models@run.time and motivates our work through a case study which is used to validate the approach. Section 3 provides an overview of the Scapegoat framework. It highlights how the component contracts are specified, how monitoring probes are injected and activated on-demand, how the ScapeGoat framework enables the definition of heuristics to detect faulty components without activating all the probes, and how we benefit from Models@run.time to build efficient heuristics. Section 4 validates the approach through a comparison of detection precision and detection speed with other approaches. Section 5 presents a use case based on an online web application[1] which leverages software diversity for safety and security purposes. In Section 6 highlights interesting points and ways to apply our approach to other contexts. Finally, Section 7 discusses related work and Section 8 discusses the approach and presents our conclusion and future work.

2. Background and motivating example

2.1. Motivating example

In this section we present a motivating example for the use of an optimistic adaptive monitoring process in the context of a real-time crisis management system in a fire department. During a dangerous event, many firefighters are present and need to collaborate to achieve common goals. Firefighters have to coordinate among themselves and commanding officers need to have an accurate real-time view of the system.

The Daum project[2] provides a software application that supports firefighters in these situations. The application runs on devices with limited computational resources because it must be mobile and taken on-site. It provides numerous services for firefighters depending on their role in the crisis. In this paper we focus on the two following roles:

- A collaborative functionality that allows commanding officers to follow and edit tactical operations. The firefighters' equipment include communicating sensors that report on their current conditions.
- A drone control system which automatically launches a drone equipped with sensors and a camera to provide a different point-of-view on the current situation.

As is common in many software applications, the firefighter application may have a potentially infinite number of configurations. These configurations depend on the number of firefighters involved, the type of crisis, the available devices and equipment,

[1] http://cloud.diversify-project.eu/
[2] `https://github.com/daumproject`

among other parameters. Thus, it is generally not possible to test all configurations to guarantee that the software will always function properly. Consequently, instead of testing all configurations, there is a need to monitor the software's execution to detect faulty behaviours and prevent system crashes. However, fine-grained monitoring of the application can have excessive overhead that makes it unsuitable with the application and the devices used in our example. Thus, there is a need for an accurate monitoring system that can find faulty components while reducing overhead.

The Daum project has implemented the firefighter application using a Component Based Software Architecture. The application makes extensive use of the Kevoree[3] component model and runtime presented below.

2.2. Kevoree

Kevoree is an open-source dynamic component platform, which relies on Models@run.time [14] to properly support the dynamic adaptation of distributed systems. Our use case application and the implementation of the Scapegoat framework make extensive use of the Kevoree framework. The following subsections detail the background on component-based software architecture, introduce the Models@run.time paradigm and give an overview of the Kevoree platform.

2.2.1. Component-based software architecture

Software architecture aims at reducing complexity through abstraction and separation of concerns by providing a common understanding of component, connector and configuration [15–17]. One of the benefits is that it facilitates the management of dynamic architectures, which becomes a primary concern in the Future Internet and Cyber-Physical Systems [18, 19]. Such systems demand techniques that let software react to changes by self-organizing its structure and self-adapting its behavior [19–21]. Many works [22] have shown the benefits of using component-based approaches in such open-world environments [23–25].

To satisfy the needs for adaptation, several component models provide solutions to dynamically reconfigure a software architecture through, for example, the deployment of new modules, the instantiation of new services, and the creation of new bindings between components [26–29]. In practice, component-based (and/or service-based) platforms like Fractal [7], OpenCOM [30], OSGi [5] or SCA [31] provide platform mechanisms to support dynamic architectures.

2.2.2. Models@run.time

Built on top of dynamic component frameworks, Models@run.time denote model-driven approaches that aim at taming the complexity of dynamic adaptation. It basically pushes the idea of reflection [32] one step further by considering the reflection-layer as a real model: "something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality". In practice, component-based and service-based platforms offer reflection APIs that allow instrospecting the application (e.g., which components and bindings are currently in place in the system) and dynamic adaptation

[3] `http://www.kevoree.org`

(e.g., changing the current components and bindings). While some of these platforms offer rollback mechanisms to recover after an erroneous adaptation [33], the purpose of Models@run.time is to prevent the system from actually enacting an erroneous adaptation. In other words, the "model at runtime" is a reflection model that can be decoupled from the application (for reasoning, validation, and simulation purposes) and then automatically resynchronized. This model can not only manage the application's structural information (i.e., the architecture), but can also be populated with behavioural information from the specification or the runtime monitoring data.

2.2.3. The Kevoree framework

Kevoree provides multiple concepts that are used to create a distributed application that allows dynamic adaptation. The *Node* concept is used to model the infrastructure topology and the *Group* concept is used to model the semantics of inter-node communication, particularly when synchronizing the reflection model among nodes. Kevoree includes a *Channel* concept to allow for different communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (*Component*, *Channel*, *Node*, *Group*) obey the object type design pattern [34] in order to separate deployment artifacts from running artifacts.

Kevoree supports multiple execution platforms (e.g., Java, Android, MiniCloud, FreeBSD, Arduino). For each target platform it provides a specific runtime container. Moreover, Kevoree comes with a set of tools for building dynamic applications (a graphical editor to visualize and edit configurations, a textual language to express reconfigurations, several checkers to valid configurations).

As a result, Kevoree provides a promising environment by facilitating the implementation of dynamically reconfigurable applications in the context of an open-world environment. Because our goal is to design and implement an adaptive monitoring system, the introspection and the dynamic reconfiguration facilities offered by Kevoree suit the needs of the ScapeGoat framework.

3. The ScapeGoat framework

Our optimistic adaptive monitoring system extends the Kevoree platform with the following principles: i) component contracts that define per-component resource usage, ii) localized and just-in-time injection and activation of monitoring probes, iii) heuristic-guided faulty component detection. The following subsections present an overview of these three principles in action.

3.1. Specifying component contracts

In both the Kevoree and ScapeGoat approaches, we follow the contract-aware component classification [12], which applies B. Meyer's Design-by-Contract principles [35] to components. In fact, ScapeGoat provides Kevoree with *Quality of Service* contract extensions that specify the worst-case values of the resources the component uses. The resources specified are memory, CPU, I/O and the time to service a request. The exact semantic of a contract in ScapeGoat is: *the component will consume at most X resource if it receives at most N requests on its provided ports.*

For example, for a simple Web server component we can define a contract on the number of instructions per second it may execute [10] and the maximum amount of memory it can consume. The number of messages can be specified per component or per component-port. In this way, the information can be used to tune the usage of the component roughly or detailedly. An example is shown in Listing 1.[4] This contract extension follows the component interface principle [36], and allows us to detect if the problem comes from the component implementation or from a component interaction. That is, we can distinguish between a component that is using excessive resources because it is faulty, or because other components are calling it excessively.

```
add node0.WsServer650 : WsServer

// Specify that this component can use 2580323 CPU
// instructions per second
set WsServer650.cpu_wall_time = 2580323 intr/sec

// Specify that this component can consume a maximum of 15000
// bytes of memory
set WsServer650.memory_max_size = 15000 bytes

// Specify that the contract is guaranteed under the assumption that
// we do not receive more than 10k messages on the component and
// 10k messages on the port named service
// (this component has only one port)
set WsServer650.throughput_all_ports = 10000 msg/sec
set WsServer650.throughput_ports.service = 10000 msg/sec
```

Listing 1: Component contract specification example

In addition, there is a global contract for each *node* of the distributed application. This contract is used for the monitoring framework to drive the adaptation of the system. It simply states when the application container is running out of resource. When this condition is detected, Scapegoat starts to actively compute the resource consumption of different components. Figure 2 depicts a simple contract where an application container (*node0*) is used. In this case, Scapegoat activates only if the CPU consumption is over 80% or the memory consumption is over 70%.

```
// there is a device in the system
add node0 : JavaNode

// the system is running out of resources if the CPU consumption is over 80%
set node0.cpu_threshold = 80%

// the system is running out of resources if the memory consumption is over 70% of the maximum heap size
set node0.memory_threshold = 70 %
```

Listing 2: Global contract specification example

[4] Contract examples for the architecture presented in section 2.1 can be found at `http://goo.gl/uCZ2Mv`.

3.2. An adaptive monitoring framework within the container

Scapegoat provides a monitoring framework that adapts its overhead to current execution conditions and leverages the architectural information provided by Kevoree to guide the search for faulty components. The monitoring mechanism is mainly injected within the component container.

Each Kevoree node/container is in charge of managing the component's execution and adaptation. Following the Models@run.time approach, each node can be sent a new architecture model that corresponds to a system evolution. In this case, the node compares its current configuration with the configuration required by the new architectural model and computes the list of individual adaptations it must perform. Among these adaptations, the node is in charge of downloading all the component packages and their dependencies, and loading them into memory. During this process, Scapegoat provides the existing container with (i) checks to verify that the system has enough resources to manage the new component, (ii) instrumentation for the component's classes in order to add bytecode for the monitoring probes, and iii) communication with a native agent that provide information about heap utilization. Scapegoat uses the components' contracts to check if the new configuration will not exceed the amount of resources available on the device. It also instruments the components' bytecode to monitor object creation (to compute memory usage), to compute each statement (for calculating CPU usage), and to monitor calls to classes that wrap I/O access such as the network or file-system. In addition, Scapegoat provides a mechanism to explore the Java heap and to account for memory consumption with an alternative mechanism.

We provide several instrumentation levels that vary in the information they obtain and in the degree they impact the application's performance:

- **Global monitoring** does not instrument any components, it simply uses information provided directly by the JVM.
- **Memory instrumentation** or memory accounting, which monitors the components' memory usage.
- **Instruction instrumentation** or instruction accounting, which monitors the number of instructions executed by the components.
- **Memory and instruction instrumentation**, which monitors both memory usage and the number of instructions executed.

Probes are synthesized according to the components' contracts. For example, a component whose contract does not specify I/O usage will not be instrumented for I/O resource monitoring. All probes can be dynamically activated or deactivated. Note that due to a technical limitation, one of the two probes implemented to check memory consumption must be always activated. This memory consumption probes, based on bytecode instrumentation must, remain activated to guarantee that all memory usage is properly accounted for, from the component's creation to the component's destruction. Indeed, deactivating this memory probes would cause object allocations to remain unaccounted for. However, probes for CPU, I/O usage and the second probe for memory can be activated on-demand to check for component contract compliance.

We propose two different mechanisms to deal with memory consumption. The first mechanism is based on bytecode instrumentation and accounts for each object created.

As mentioned previously, this mechanism cannot be disabled. The second mechanism is a just-in-time exploration of the JVM heap, performed on demand. These two mechanisms differ in i) when the computation to account for consumption is done, ii) how intensive it is, and iii) in the way the objects are accounted for. Computations in the first mechanism are spread throughout the execution of the application, short and lightweight operations are executed every time a new object instance is created or destroyed. Objects are always accounted to the component that creates them. Computations in the second mechanism occur only on demand but are intensive because they involve traversing the graph of living objects in the heap. The accounting policy follows the paradigm of assigning objects to the component that is holding them and, if an object is reachable from more than one component, it is accounted to either one randomly, as suggested in [37]. In this paper, we call this second mechanism **Heap Exploration**.

We minimize the overhead of the monitoring system by activating selected probes only when a problem is detected at the global level. We estimate the most likely faulty components and then activate the relevant monitoring probes. Following this technique, we only activate fine-grain monitoring on components suspected of misbehavior. After monitoring the subset of suspected components, if any of them are found to be the source of the problem, the monitoring system terminates. However, if the subset of components is determined to be healthy, the system starts monitoring the next most likely faulty subset. This occurs until the faulty component is found. If no components are found to be faulty, we fallback to global monitoring. If the problem still exists the entire process is restarted. This can occur in cases where, for example, the faulty behavior is transient or inconsistent. The monitoring mechanism implemented in ScapeGoat is summarized in listing 3.

```
1  monitor(C: Set<Component>, heuristic : Set<Component>→Set<Component>)
2   init  memory probes (c | c ∈ C ∧ c.memory_contract ≠ ∅)
3  while  container  is  running
4  wait  violation  in  global  monitoring
5  checked = ∅
6   faulty  = ∅
7  while checked ≠ C ∧ faulty = ∅
8  subsetToCheck =  heuristic  ( C \ checked )
9   instrument  for  adding  probes  ( subsetToCheck )
10  faulty  =  fine−grain monitoring( subsetToCheck )
11  instrument  for  removing probes  ( subsetToCheck )
12 checked = checked ∪ subsetToCheck
13  if  faulty  ≠ ∅
14 adapt  the  system  ( faulty ,  C)
15
16  fine−grain monitoring( C :  Set<Component> )
17 wait  few  milliseconds  // to obtain good information
18  faulty  = {c | c ∈ C ∧ c.consumption > c.contract}
19 return  faulty
```

Listing 3: The main monitoring loop implemented in ScapeGoat

As a result, at any given moment, applications must be in one of the following monitoring modes:

- **No monitoring.** The software is executed without any monitoring probes or modifi-

cations.

- **Global monitoring.** Only global resource usage is being monitored, such as the CPU usage and memory usage at the Java Virtual Machine (JVM) level.
- **Full monitoring.** All components are being monitored for all types of resource usage. This is equivalent to current state-of-the-art approaches.
- **Localized monitoring.** Only a subset of the components are monitored.
- **Adaptive monitoring.** The monitoring system changes from Global monitoring to Full or Localized monitoring if a faulty behaviour is detected.

For the rest of this paper we use the term **all components** for the adaptive monitoring policy that indicates that the system changes from *global monitoring* mode to *full monitoring* mode if and when a faulty behaviour is detected.

3.2.1. ScapeGoat's architecture

The Scapegoat framework is built using the Kevoree component framework. Scapegoat extends Kevoree by providing a new Node Type and three new Component Types:

- **Monitored Node.** Handles the admission of new components by storing information about resource availability. Before admission, it checks the security policies and registers components with a contract in the monitoring framework. Moreover, it intercepts and wraps class loading mechanisms to record a component type's loaded classes. Such information is used later to (de)activate the probes.
- **Monitoring Component.** This component type is in charge of checking component contracts. Basically, it implements a complex variant of the algorithm in listing 3. It communicates with other components to identify suspected components.
- **Ranking Component.** This is an abstract Component Type; therefore it is user customizable. It is in charge of implementing the heuristic that ranks the components from the most likely to be faulty to the least likely.
- **Adaptation component.** This component type is in charge of dealing with the adaptation of the application when a contract violation is detected. It is also a customizable component. The adaptation strategy whenever a faulty component is discovered is out os scope of this paper. Nevertheless, several strategies may be implemented in Scapegoat, such as removing faulty components or slowing down communication between components when the failure is due to a violation in the way one component is using another.

3.2.2. Extensibility of the ScapeGoat Framework

The scapegoat framework has been built with the idea of being as generic as possible, thus supporting various extensions and specializations. In this section we discuss the extension points provided by the ScapeGoat framework.

Heuristics used to rank suspected faulty components can be highly specialized and, as we show in section 4, have a remarkable impact on the behavior of ScapeGoat. A new heuristic is created by defining a component that implements an interface to provide a ranking of the suspected components. To do so, a context is sent with each

ranking request on this component. This context is composed of three elements, i) a model that describes the components and links of the deployed application, ii) a history that contains all the models that have been deployed on the platform, and iii) a history of failures composed of metadata regarding what components have failed as well as why and when it happened. In this paper, we present three heuristics. The first heuristic is proposed in section 3.3 and shows how we can leverage the Models@Run.time paradigm to guide the framework in finding the component that is behaving abnormally. Due to their simplicity, the other two heuristics are presented in section 4 where we use them to evaluate the behavior of ScapeGoat.

The mechanism for creating new heuristics is based on the strategy design pattern. Figures 1 and 2 illustrates this extension point.

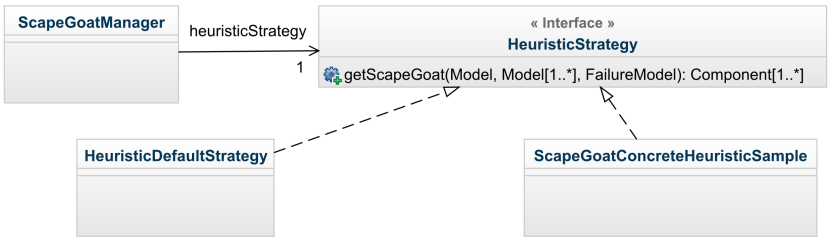

Figure 1: Heuristic extension point in Scapegoat. This illustrates the class diagram.

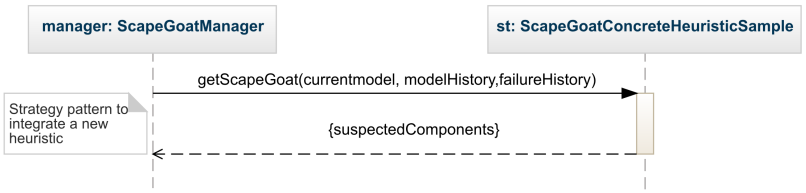

Figure 2: A sequence diagram showing how the extension point to define heuristics in Scapegoat is used.

A second extensible aspect of the framework is the admission control system. The framework provides an API to hook user-defined actions when new components are submitted for deployment. Basic data describing the execution platform in terms of resource availability, information about the already deployed components and the new component's contract are sent to the user-defined admission control system. On each request, the admission control system has to accept or refuse the new component. In this paper, we are using an approach which check the theoretical availability of resources whenever a component is deployed, and accept the new component if the contract can fit in the remaining available resources. ScapeGoat is meant to support other policies as, for instance, overcommitment.

A last element that can be specialized to user needs is the contracts semantic. In section 3.1 we describe how we interpret the contract in this work. However, it is possible to define other contract semantics, for instance, accepting values that are closed to the limit defined in the contract, or using fuzzy values instead of sharp values. It is worth noting that modifying the semantic of the contract would likely involved redefining the domain-specific language to describe contract and also modifying the admission control system.

3.2.3. Implementation strategy

Scapegoat aims at minimizing monitoring overhead when the framework is monitoring the global behavior of the JVM. To achieve this, ScapeGoat uses as few probes as possible when executing in global monitoring mode. Only when it is necessary, ScapeGoat activates the required probes. This features are implemented in the framework in three modules that are in charge of different concerns: a module to activate/deactivate the probes, a module to collect the resource usage, and a module to compute what components should be carefully monitored. In this section we focus on the modules for activating/deactivating probes and for collecting information of resource usage because they required considerable engineering effort. Notice, however, that this module is executed on demand when the framework already decides the monitoring mode to use and what components to monitor.

Module to activate/deactivate probes. In ScapeGoat we use bytecode instrumentation to perform localized monitoring. However, instead of doing as previous approaches that manipulate the bytecode that defines components just when the component's code is executed for the first time, we modify the bytecode many times during components' life. Every time the monitoring mode is changed we either activate or deactivate the probes by simply inserting them in the bytecode or by removing them. Implementing this mechanism at per-component basis requires knowing all the classes that have been loaded for a component. This information is kept using a dictionary in which we treat a component's id as a key and a set of class names as a value. The dictionary is filled using the *traditional* classloader mechanism of Java. In short, when a class is loaded on behalf of a component, we detect the class name and the thread that is loading the class. Using the thread's id we are able to identify the component because we use special naming conventions for each thread executing the initial code of a component. When probes are activated/deactivated on a component, iterating over the set of class names allows the re-instrumentation of each involved class.

The probes perform two actions: collecting data about the local usage of resources (e.g., objects recently allocated, instructions executed in the current basic block, bytes sent through the network), and notifying to the resource consumption monitor about the collected data. Some data we collect is computed statically when the bytecode is loaded. This includes the size of each basic block and the size of each object allocated when the size of each instance of the class is already known. Other data, such as bytes sent through the network or the size of allocated arrays, can only by collected dynamically when the code is running. To notify about the collected data we use simple method calls to a proxy class in charge of forwarding the data to the monitoring module. Probes to detect CPU consumption are inserted at the end of each basic block. These

probes collect the size in number of instructions of its container basic block. Probes for IO throughput and network bandwidth are added in a few selected method defined in classes of the Java Development Kit (JDK). These probes take the needed information from local variables (e.g., number of bytes) and call the proxy class.

Our implementation, which is built using the ASM library [5] for bytecode manipulation and a Java agent to get access to and transform the classes, is based on previous approaches to deal with resource accounting and profiling in Java [10, 38, 39]. As in previous approaches, we compute the length of each basic block to count the number of executed instructions and we try to keep a cache of *known* methods with a single basic block. Moreover, we compute the size of each object once it is allocated and we use *weak* references instead of *finalizers* to deal with deallocation.

Module to collect information regarding resource usage. In Scapegoat, there are two mechanisms to collect information about how components consume resources.

The first mechanism is able to capture the usage of CPU, IO throughput, network bandwidth and memory. Every time a probe that was inserted in the code of a component is executed, the proxy class forwards the local resource usage to the module in charge of collecting the resource usage. Along with the local resource usage, probes also notify the id of the components consuming resource. Such data is then used to aggregate the global consumption of each component. It is worth noting that, when this first mechanism is used to collect memory consumption, on object is always accounted as consumed for the component responsible for its initial allocation. In short, no matter whether the initial component C that allocates the object no longer held a reference to an object O, as long as O remains in memory, C is accounted for its consumption. Moreover, as was already mentioned, using this mechanism is not possible to deactivate the probes related to memory consumption.

On the contrary, the second mechanism is only useful to collect information about memory consumption. The advantages of this method are: we can leverage the proposed optimistic monitoring because it executes only on demand, and it has no impact on the number of objects allocated in memory because no *weak references* are used. However, in this method an object O is consumed not for the component that allocates it but for those components that held references to it. As a consequence, in certain occasions the framework states that an object is being consumed for many components at the same time. We built this solution on top of the JVM Tool Interface (JVMTI) by implementing the algorithm proposed in [37, 40], with the main difference being that our solution works without modifying the garbage collector. In summary, this algorithm simply try to find those objects that are reachable from the references of each component. It does so by traversing the graph of live object using as the component instance and its threads as roots of the traversal. Since our approach does not require a modification to the garbage collector, it is portable and works with different garbage collector implementations.

[5] asm.ow2.org

3.3. Leveraging Models@run.time to build an efficient monitoring framework

As presented in section 3.2, our approach offers a dynamic way to activate and deactivate fine-grain localized monitoring. We use a heuristic to determine which components are more likely to be faulty. Suspected components are the first to be monitored.

Our framework can support different heuristics, which can be application or domain-specific. In this paper we propose a heuristic that leverages the use of the Models@run.time approach to infer the faulty components. The heuristic is based on the assumption that the cause of newly detected misbehavior in an application is likely to come from the most recent changes in the application. This can be better understood as follows:

- recently added or updated components are more likely to be the source of a faulty behaviour;
- components that directly interact with recently added or updated components are also suspected.

We argue that when a problem is detected it is probable that recent changes have led to this problem, or else, it would have likely occurred earlier. If recently changed components are monitored and determined to be healthy, it is probable that the problem comes from direct interactions with those components. Indeed, changes to interactions can reveal dormant issues with the components. The algorithm used for ranking the components is presented in more detail in listing 4. In practice, we leverage the architectural-based history of evolutions of the application, which is provided by the Models@run.time approach.

Listing 4 shows two routines, but only routine *ranker* is public. It can be called by the monitoring system when it is necessary to figure out in what order components must be carefully monitored. After initializing an empty list which will hold the rank, the algorithm starts to iterate in line 4 over the history of models that have been installed in the system. As mentioned, this history contains a sorted set of models that describe what components have been installed in the system. Within each iteration, the algorithm first computes in line 5 the set of components that were installed at such a point in time. Afterwards, these components are added to the result. The next step, executed at lines 8 and 9, is finding those components that are directly connected to components that were added to the application at this point in time. Finally, these *neighbors* are added to the rank after being sorted. Routine *sort* simply sorts a set of components using as criteria the time at which components where installed in the system.

3.4. What kind of failures is Scapegoat able to detect?

It is worth discussing in what contexts developers can leverage the proposed monitoring framework. Scapegoat is a framework implemented at the application level; this has important implications from a security point of view that depend on particularities of the target technology. For instance, using a Java implementation of Scapegoat in a completely open-world environment poses the same challenges as using OSGi in to create an open platform where untrusted stakeholders can deploy OSGi bundles. This is, due to the lack of isolation in the JVM, a malicious stakeholder can corrupt other OSGi bundles or, in the case of Scapegoat, it can corrupt the framework.

```
1  ranker () : list <Component>
2  // used to avoid adding duplicated elements to the list
3  visited = ∅
4  // this list will contain the result of calling the routine
5  ranking = {}
6  for each model M ∈ History
7  // adding components that were added in this model
8  N = {c | c was added in M}
9  ranking.add N\ visited
10 visited = visited ∪ N
11 // finding neighbors
12 Neighbors = ⋃_{c∈N} c.neighbors
13 SortedNeighbors = sort (Neighbors \ visited , History)
14 // adding neighbors
15 ranking.add SortedNeighbors
16 visited = visited ∪ Neighbors
17 // return the built ranking
18 return ranking
19
20 // this routine recursively sort a set of components using the following criteria :
21 // components are sorted by the timestamp that indicates when they were installed
22 private sort (S : Set<Component>, H : History) : list <Component>
23 r = {}
24 if S ≠ ∅
25 choose b | b ∈ S ∧ b is newer with respect to H than any other element in S
26 r.add b, sort (S\{b}, H)
27 return r
```

Listing 4: The ranking algorithm (uses the model history for ranking).

Scapegoat is nevertheless useful in many contexts. For instance, it can detect components with development errors, and components in which the implementation do not follow the specification. In addition, the framework is capable of detecting small differences in the resource consumption of components with similar functional behavior. Hence, it is useful to choose what component to deploy in order to improve the QoS. Finally, Scapegoat can also provide hints of malicious attacks such as denial-of-services attacks because it is able to pinpoint the real source of the excessive resource consumption when several components are interacting. This does not mean that you can fully trust Scapegoat as a tool to avoid malicious attacks. In other words, Scapegoat is susceptible to false negatives (unreported attacks) but we can expect that it will trigger few false positives (reporting nonexistent attacks).

4. ScapeGoat Performance Evaluation

In this section we present a first series of experiments and discuss the usability of our approach. We focus on the following research questions to assess the quality and the efficiency of ScapeGoat:

- **What is the impact of the various levels of instrumentation on the application?** Our approach assumes high overhead for full monitoring and low overhead for a lightweight global monitoring system. The experiments presented in section 4.2 show the overhead for each instrumentation level.

- **What is the performance cost of using instrumentation-based and heap-exploration-based memory monitoring?** Since both mechanisms have by design different features, the experiments in section 4.2 show the overhead each mechanism produces.
- **Does our adaptive monitoring approach have better performance than state-of-the-art monitoring solutions?** The experiment presented in section 4.3 highlights the performances benefits of our approach considering a real-world scenario.
- **What is the impact of using a heuristic in our adaptive monitoring approach?** The experiment presented in section 4.4 highlights the impact of the application and component sizes, and the need of a good heuristic to quickly identify faulty components.

The efficiency of our monitoring solution is evaluated on two dimensions: the overhead on the system and the delay to detect failures. We show there is a trade-off between the two dimensions and that ScapeGoat provides a valuable solution that increases the delay to detect a faulty component but reduces accumulated overhead. This evaluation has been conducted on a Cyber Physical System case study. It corresponds to a concrete application that leverage the Kevoree framework for dyamic adaptation purpose.

We have built several use cases based on a template application from our motivating example in section 2.1. We reused an open-source crisis-management application for firefighters that has been built with Kevoree components. We use two functionalities of the crisis-management application. The first one is for managing firefighters. The equipment given to each firefighter contains a set of sensors that provides data for the firefighter's current location, his heartbeat, his body temperature, his acceleration movements, the environmental temperature, and the concentration of toxic gases. These data are collected and displayed in the crisis-management application, which provides a global-view of the situation. The second functionality uses drones to capture real-time video from an advantageous point-of-view.

Figure 3 shows the set of components that are involved in our use-case, including components for firefighters, drones and the crisis-management application[6]. The components in the crisis-management application are used in our experiments, but the physical devices (drones and sensors) are simulated through the use of mock components. The application presents two components: the first one is a web browser that shows information about each firefighter in the terrain, and the second one allows to watch the video being recorded by any drone in the field. A Redis database is used to store the data that is consumed for the application's GUI.

Every use case we present extends the crisis-management base application by any one of the following possibilities: adding new or redundant components, adding external Java applications with wrapper components (e.g., Weka, DaCapo), or modifying existing components (e.g., to introduce a fault into them). Using this template in the

[6]More information about these components is given in `http://goo.gl/x64wHG`

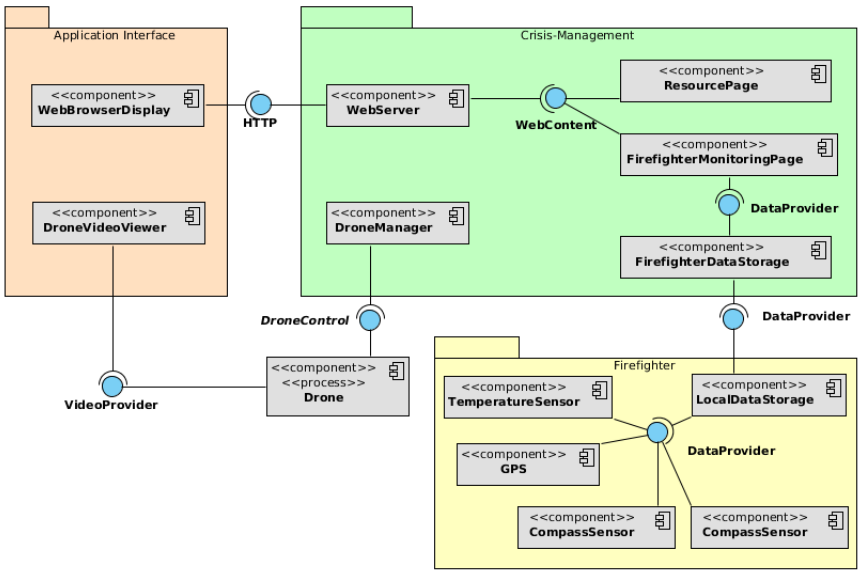

Figure 3: The component configuration for our crisis-management use-case.

experiments allow us to measure the behavior of our proposal in a more realistic environment where many components with different features co-exist.

4.1. Measurement Methodology

To obtain comparable and reproducible results, we used the same hardware across all experiments: a laptop with a 2.90GHz Intel(R) i7-3520M processor, running Fedora 19 with a 64 bit kernel and 8GiB of system memory. We used the HotSpot Java Virtual Machine version 1.7.0_67, and Kevoree framework version 5.0.1. Each measurement presented in the experiment is the average of ten different runs under the same conditions.

The evaluation of our approach is tightly coupled with the quality of the resource consumption contracts attached to each component. We built the contracts following classic profiling techniques. The contracts were built by performing several runs of our use cases, without inserting any faulty components into the execution. Firstly, we executed the use cases in an environment with global monitoring activated to get information for the global contract. Secondly, per-component contracts were created by running the use cases in an environment with full monitoring.

4.2. Overhead of the instrumentation solution

Our first experiment compares the various instrumentation levels to show the overhead of each one. In this section, *Memory instrumentation* refers to the technique for accounting memory which leverage bytecode instrumentation, while *Heap Exploration* refers to the memory accouting technique which leverage on-demand heap exploration. In this experiment, we compare the following instrumentation levels: *No monitoring*,

Global monitoring, *Memory instrumentation*, *Instructions instrumentation*, *Memory and instructions instrumentation* (i.e., Full monitoring). We also evaluate the impact on performance of the two fine-grain memory monitoring approaches we proposed: instrumentation-based and heap-dump-based.

In this set of experiments we used the DaCapo 2006 benchmark suite [41]. We developed a Kevoree component to execute this benchmark [7]. The container was configured to use full monitoring and the parameters in the contract are upper bounds of the real consumption[8].

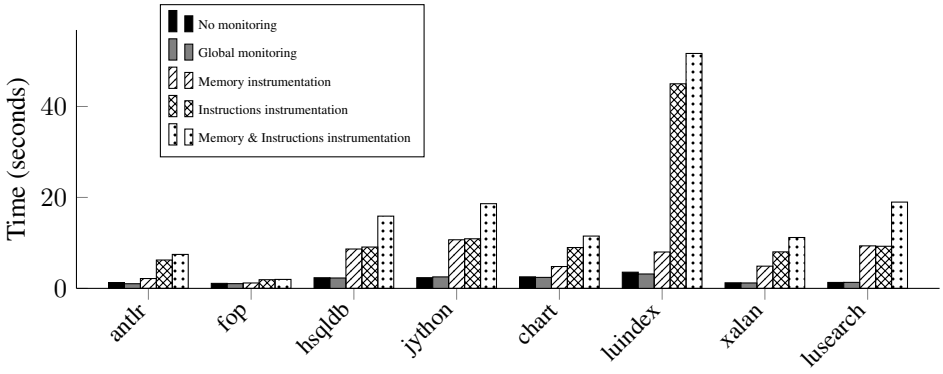

Figure 4: Execution time for tests using the DaCapo Benchmark

Figure 4 shows the execution time of several DaCapo tests under different scenarios when only instrumentation is used to provide fine-grain monitoring. First, we wish to highlight that Global monitoring introduces no overhead compared with the *No monitoring* mode. Second, the overhead due to memory accounting is lower than the overhead due to instruction accounting. This is very important because, as we described in section 3.2, memory probes cannot be deactivated dynamically.

To perform the comparison, we evaluate the overhead produced for each monitoring mode. We calculated the overhead as:

$$overhead = \frac{WithInstrumentation}{GlobalMonitoring}$$

The average overhead due to instruction accounting is 5.62, while the value for memory accounting depends on the monitoring mechanism. If bytecode instrumentation is used, the average overhead is 3.29 which is close to the values reported in [11]. In the case of instruction accounting, these values are not as good as the values reported in [11]; because they obtain a better value between 3.2 and 4.3 for instructions accounting. The performance difference comes from a specific optimization that we chose not to apply. The optimization provides fast access to the execution context by adding a new parameter to each method. Nevertheless, this solution needs to keep a version

[7] `http://goo.gl/V5T6De`

[8] Scripts are generated from those available at `http://goo.gl/FR8LC7`.

of the method without the new parameter because native calls cannot be instrumented like that. We decided to avoid such an optimization because duplication of methods increases the size of the applications, and with it, the memory used by the heap. In short, our solution can reach similar values if we include the mentioned optimization, but at the cost of using more memory. On the other hand, the values we report are far lower than the values reported in [11] for hprof. Hence, we consider that our solution is comparable to state of the art approaches in the literature.

In figure 5 we compare the execution time of the same benchmarks but using different memory monitoring approaches. This comparison is important because, as explained in section 3.2, the two approaches have different CPU footprint. These are controlled experiments where, in order to stress the technique, we demand the execution of a *heap exploration* step every two seconds, which is not the expected usage pattern. On the contrary, the memory instrumentation technique is executed with the expected usage pattern. In comparison to using memory instrumentation where the average execution time is 3.29, the average overhead in execution time decreases to 1.79 if the *Heap Exploration* monitoring mechanism is used. This value is better than the value reported in [11]. These results suggest that this technique has less impact on the behavior of applications being monitored.

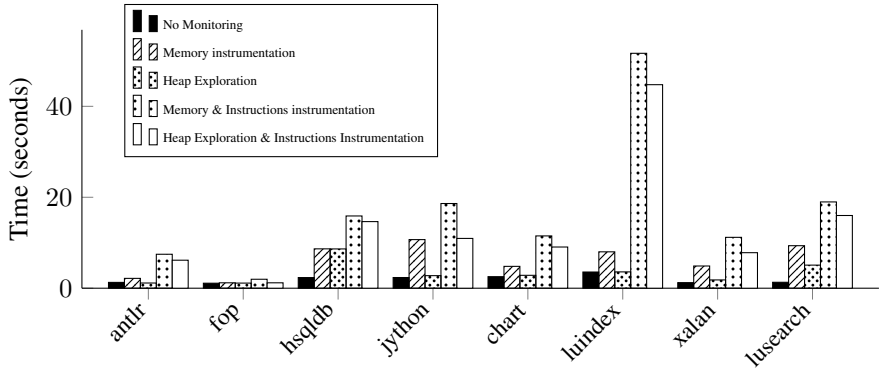

Figure 5: Comparison of execution time for tests using two different memory monitoring techniques

The results of our experiment shown in figures 4 and 5 demonstrate the extensive impact of the *Full monitoring* mode, which uses either *Memory instrumentation* or *CPU instrumentation*, has on the application. Thus, our *Adaptive monitoring* mode, which uses *Global monitoring* and switches to *Full monitoring* or *localized monitoring*, has the potential to reduce this accumulated overhead due to the fact that *Global monitoring* has no appreciable overhead.

In addition, we plan to study alternatives to improve instruction accounting. For example, we plan to study the use of machine learning for monitoring [42]. Based on a machine learning approach, it is possible to train the monitoring system to do the instruction instrumentation. Then, instead of doing normal instruction instrumentation, we might only do, for example, method-calls instrumentation and with the learning data, the monitoring system should be able to infer the CPU usage of each call, whilst

lowering the overhead.

4.3. Overhead of Adaptive Monitoring vs Full Monitoring

The previous experiment highlights the potential of using *Adaptive monitoring*. However, switching from *Global monitoring* to either *Full* or *Localized monitoring* introduces an additional overhead due to having to instrument components and activate monitoring probes. Our second experiment compares the overhead introduced by the adaptive monitoring with the overhead of *Full monitoring* as used in state-of-the-art monitoring approaches.

Table 1 shows the tests we built for the experiment. We developed the tests by extending the template application. Faults were introduced by modifying an existing component to break compliance with its resource consumption contract. We reproduce each execution repetitively; thus, the faulty behaviour is triggered many times during the execution of the application. The application is not restarted.

Table 1: Setting for each use case

Test Name	**Monitored Resource**	**Faulty Resource**	**Heuristic**	**External Task**
UC1	CPU, Memory	CPU	number of failures	Weka, train neural network
UC2	CPU, Memory	CPU	number of failures	dacapo, antlr
UC3	CPU, Memory	CPU	number of failures	dacapo, chart
UC4	CPU	CPU	number of failures	dacapo, xalan
UC5	CPU, Memory	CPU	less number of failures first	dacapo, chart
UC6	Memory	CPU	number of failures	Weka, train neural network

Figure 6 shows the execution time of running the use cases with different scenarios. Each scenario uses a specific monitoring policy (*Full monitoring*, *Adaptive monitoring with All Components*, *Adaptive monitoring with Localized monitoring*, *Global monitoring*). All these scenarios were executed with the heap explorer memory monitoring policy. This figure shows that the overhead differences between *Full monitoring* and *Adaptive monitoring with All Components* is clearly impacted by scenarios that cause the system to transition too frequently between a lightweight Global and a fine-grain Adaptive monitoring. Such is the case for use cases UC3 and UC4 because the faulty component is inserted and never removed. Using *Adaptive monitoring* is beneficial if the overhead of *Global monitoring* plus the overhead of switching back and forth to *All Components monitoring* is less than the overhead of the *Full monitoring* for the same execution period. If the application switches between monitoring modes too often then the benefits of adaptive monitoring are lost.

The overhead of switching from *Global monitoring* to *full components* or *Localized monitoring* comes from the fact that the framework must reload and instrument classes to activate the monitoring probes. Therefore, using *Localized monitoring* reduces the number of classes that must be reloaded. This is shown in the third use-case of figure 6, which uses a heuristic based on the number of failures. Because we execute the faulty component many times, the heuristic is able to select, monitor and identify the faulty component quickly. This reduces overhead by 93%. We use the following equation to calculate overhead:

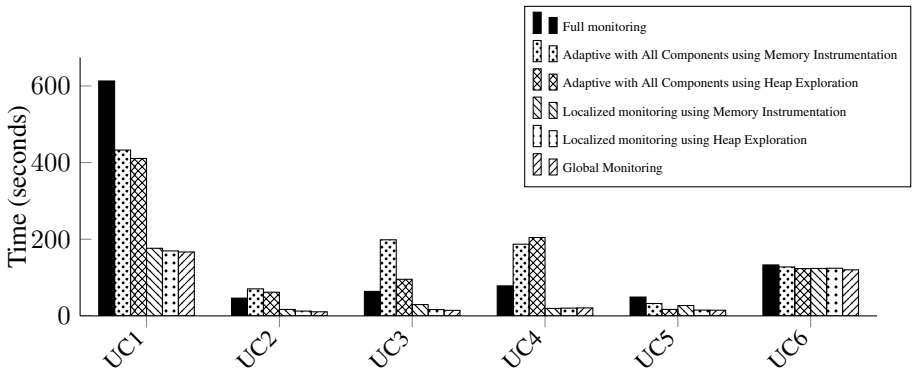

Figure 6: Execution time for some use cases under different monitoring policies.

$$Gain = 100 - \frac{OurApproach - GlobalMonitoring}{FullMonitoring - GlobalMonitoring} * 100$$

We also evaluate the execution time for each use case using the instrumentation-based memory monitoring mode. The average gain in that case is 81.49% and, as shown in previous section, in average it behaves worse than the *Heap Exploration* mechanism. However, it is worth noting that the difference between using memory monitoring based on instrumentation and heap exploration is less remarkable than in the previous experiment. Observe how in test UC4, using a combination of heap exploration and adaptive monitoring with all components behaves worse than using plain instrumentation-based memory monitoring. In this particular test, activating and deactivating the monitoring probes dominate the execution time. Alas, adding a heap exploration step right after the probes are activated, just add some extra overhead. On the contrary, there is no additional step executed when we use instrumentation to measure the memory usage. Apparently, what matter when the all components strategy is guiding the adaptive monitoring is the ratio among the amount of allocations performed by components and the size of those components.

4.4. Overhead from switching monitoring modes, and the need of a good heuristic

As we explain in the previous experiments, even if using *Localized monitoring* is able to reduce the overhead of the monitoring system, the switch between *Global* and *Localized monitoring* introduces additional overhead. If this overhead is too high, the benefits of adaptive monitoring are lost.

In this experiment we show the impact of the application's size, in terms of number of components, and the impact of the component's size, in terms of number of classes, on adaptive monitoring. We also show that the choice of the heuristic to select suspected components for monitoring is important to minimize the overhead caused from repeated instrumentation and probe activation processes.

For the use case, we created two components and we introduced them into the template application separately. Both components perform the same task, which is

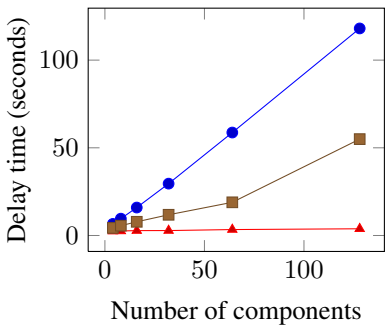

Figure 7: Delay time to detect fault with a component size of 115 classes.

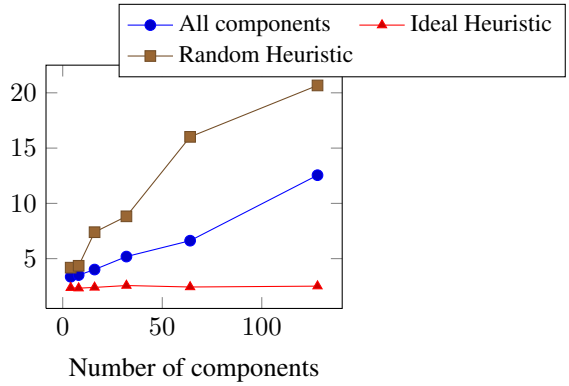

Figure 8: Delay time to detect fault with a component size of four classes.

performing a *primality test* on a random number and sending the number to another component. However, one of the components causes 115 classes to be loaded, while the other only loads 4 classes.

We used the same basic scenario with a varying number of *primality testing's* components and component sizes. In this way, we were able to simulate the two dimensions of application size. The exact settings, leading to 12 experiments, are defined by the composition of the following constraints:

- $N_{comp} = \{4, 8, 16, 32, 64, 128\}$ which defines the number of components for the application
- $Size_{comp} = \{4, 115\}$ which defines the number of classes for a component

With these use cases, we measured the delay to find the faulty component and the execution-time overhead caused by monitoring. Figures 7 and 8 show the delay to detect the faulty component with regards to the size of the application. In the first figure, the component size is 115 classes, and in the second figure, the component size is four classes.

4.4.1. Impact of the application size

Using figures 8 and 10, we see that the size of the application has an impact on the delay to detect faulty components, and also on the monitoring overhead. We also calculated the time needed to find the faulty component with the *All components* mode after its initialization (the time needed to switch from *Global monitoring*). This time is around 2 seconds no matter the size of the application. That is the reason the switch from *Global monitoring* to *All components* has such a large effect on overhead.

These figures also show that using *Localized monitoring* instead of *All components* when switching from *Global monitoring* helps reduce the impact of the application's size by reducing the number of components to monitor and the number of classes to instrument. However, we also see that using a sub-optimal heuristic may have negatively

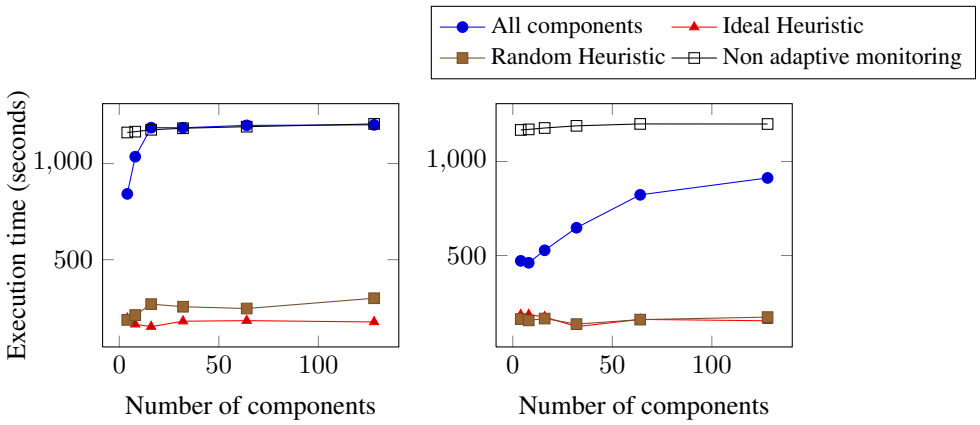

Figure 9: Execution time of main task with a component size of 115 classes.

Figure 10: Execution time of main task with a component size of four classes.

impacted the delay to detect faulty components. This can be explained by the multiple switches that the Random heuristic may often require to locate the faulty component.

4.4.2. Impact of the component size

In figures 7 and 8 we can observe that the component size greatly impact the performance and the delay for ScapeGoat to find the faulty comopnent. Similar to the explanation for the application's size, component size impacts the switch from *Global monitoring* to *Localized monitoring*, beacause of the class reloading and instrumentation. A good heuristic rastically reduces the number of transitions; thus, it has a huge impact on the delay. When the components size increase, the choice of a good heuristics becomes even more important, because the cost of dynamic monitoring probes injection increase with the size of the components.

5. ScapeGoat to spot faulty components in a scalable diverse web application

In this section, we present another application that benefits from the Scapegoat approach. Although the general goal of spotting components that behave abnormally regarding resource consumption remains the same, with this use case we highlight the possibility of using Scapegoat to automatically find buggy components on a scalable modular web application. The section 5.1 presents an introduction to the application use case, while the remainder of the section deals with the experimental setup and the results.

5.1. Use case presentation

We are applying the ScapeGoat approach to check resource consumption contracts on a web application called MdMS.[9] This application offers a web Content Manage-

[9]https://github.com/maxleiko/mdms-ringojs

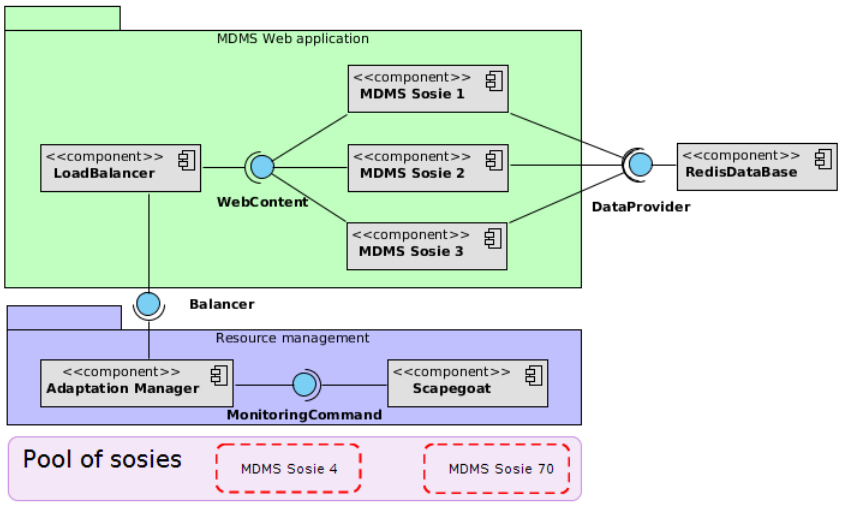

Figure 11: Architecture of MdMS along with Scapegoat and additional components to adapt th system.

ment System based on the Markdown language for editing posts. MdMS uses a typical architecture (as shown in Figure 11) for scalable web applications: a load-balancer connected to a set of workers (called MdMS Sosie in the Figure 11), which are themselves connected to a distributed database to retrieve the application specific content. The worker layer of this application can be duplicated across various machines to support a growing number of clients. The web application is currently online[10].

The main characteristic of MdMS is that all workers are not pure clones but diverse implementations of the MdMS server stack [43]. This proactive diversification of MdMS targets safety [44] and security [45] purposes. In particular, we have used our recent technique for the automatic synthesis of *sosie* programs [46] in order to automatically diversify the workers. A *sosie* is a variant of a program that exhibits the same functionality (passes the same test suite) and a diverse computation (different control or data flow). *Sosie* synthesis is based on the transformation of the original program through statement deletion, addition or replacement.

While the construction of *sosies* focuses on preserving functional correctness, it ignores all the non-functional aspects of the program. Consequently, a *sosie* offers no guarantee regarding its resource consumption and may contain memory leaks or other overhead on resource consumption that can significantly impact the performance of MdMS.

In this experiment, we use ScapeGoat to monitor the resource consumption of the various *sosies* of the MdMS workers. This technique enables us to identify *sosies* in a production environment that do not behave according to the resource consumption contracts, allowing the system to remove these workers and use other *sosies*. Our goal

[10]http://cloud.diversify-project.eu/

in this experiment is to answer the following research question:

- Does ScapeGoat correctly identify the faulty components in a system which includes many variants of the same component?

5.2. Experimental setup

We devised this experiment as a scenario where many clients interact with the web application at the same time by adding and removing articles. The stress produced by these requests increases the resource consumption on the server side which is running on top of Kevoree components. Figure 11 depicts the server side's configuration. Since MdMS is a web application developed on top of RingoJS [11], a JavaScript runtime written in Java, our *sosies* include the RingoJS framework and the application that has been wrapped into Kevoree components.

In this experiment, we deploy many of these components as back-end servers of the web application and we use ScapeGoat to monitor the consumption of each server. Their contracts regarding resource consumption were built using the mechanism described in section 4.1 but with the original MdMS worker as a reference component. The application also contains a component acting as a front-end that evenly distributes the requests among back-end servers. This load balancer implements a plain round robin policy.

To produce a realistic load on the web server we have recorded a set of standard activities on the MdMS web site using Selenium [12]. We then use the Selenium facilities to replay these activities many times in parallel to provide the required work load on the server. Our experimental settings feature 120 clients which are scheduled by a pool of 7 concurrent Selenium workers. Each client adds 10 articles to the database through the Website GUI, which represents 16 requests per article, for a total of 19200 requests to the MdMS workers sent through the load balancer. In this experiment, the Selenium workers are executed on the same physical device as the web server, with the same testing platform described in section 4.1.

The experiment is configured as follows. Using the diversification technique described in [46], we synthesized 20 *sosies* of the MdMS workers. These *sosies* are used to execute the application with a varying number of back-ends (from 4 to 10). One particular *sosie* has been modified by hand to ensure that it violates the original component's contract. We execute all the described components as well as the ScapeGoat components on a single instance of Kevoree.

5.3. Experimentation results

Figure 12 shows the time required on the server side to reply to all the requests sent by Selenium. Although the values might look surprisingly high at first, they are in fact the result of a heavily loaded system. Selenium is actually rendering a couple of web pages for each added article; hence at least 2400 pages are rendered. Moreover, both clients and servers are sharing resources because they run on the same physical device.

[11] https://github.com/ringo/ringojs
[12] http://www.seleniumhq.org/

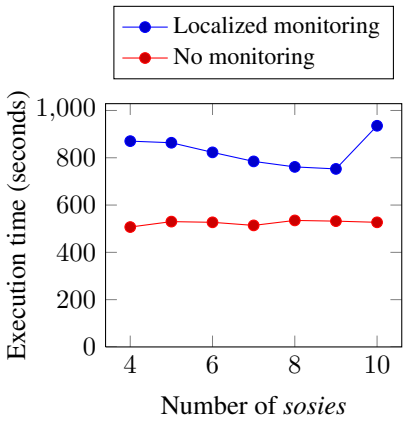

Figure 12: Time to obtain the reply to all requests.

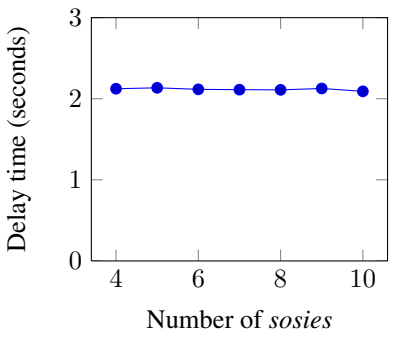

Figure 13: Average delay time to detect a faulty *sosie*.

This leads to very stable execution times when monitoring is not activated because the number of requests does not change between experiments, the load balancer distributes these requests evenly, and we are using the same physical device to execute all back-end servers. In the *local monitoring* series, the global time to execute decreases until reaching 9 *sosies*. Although counterintuitive, it is caused by the effect of having *localized monitoring* and *load balancing* at the same time. For instance, when four *sosies* are used, the monitoring probes are periodically injected into one component out of four, hence roughly a quarter of the requests are handled by a slower *sosie*. However, with eight components the slow execution path is only taken by around 12.5% of the requests. The overhead of *Localized monitoring* when ten *sosies* are deployed increases because the physical machine reaches its limit and begins thrashing. As a consequence, low-level interactions with the hardware (e.g. cache misses), the operating system and the JVM slow down the execution. On average, the overhead due to monitoring with both instruction instrumentation and memory instrumentation is 1.59, which is lower than the values shown in section 4.2 for full monitoring despite only one of the instrumentation mechanisms being enabled in those experiments. The values in this section are better even if we are monitoring both resources because we are using the adaptive approach.

In these experiments, we evaluate the accuracy of the output and its quality in terms of the time needed to find the faulty component. ScapeGoat always spots the correct *sosie*. It does so because it is an iterative process that continues until finding the faulty component. In addition, ScapeGoat does not output false positives during these experiments. The delay to detect faulty components is shown in Figure 13. In this case, the values remain close to 2 seconds no matter the number of *sosies* used nor the execution time. This behavior is consistent with the experiments in section 4 because we are also using a good heuristic for the use case. It shows that ScapeGoat can spot faulty components with an acceptable delay in a real application.

5.4. Discussion of the use case

This use case shows that ScapeGoat is able to provide useful information in real applications. It also highlights how the framework can help select software variants at runtime in the context of software diversity. Or, more generally, in the field of software oriented architectures where many stakeholders may provide the same services, Scapegoat can help to choose services. Moreover, this use case leads to a distributed usage of ScapeGoat, where the policies for admission control and resource consumption monitoring can be coordinated among distributed devices.

Finally, in systems where there are many variants of the same component or service, ScapeGoat provides essential information to drive application reconfiguration. For example, the adaptation component in figure 11 may use Scapegoat's faulty component selection to replace a faulty *sosie* or to modify the scheduling policy in the load balancer.

5.5. Threats to validity

Our experiments show the benefits of using adaptive monitoring instead of state-of-the-art monitoring approaches. As in every experimental protocol, our evaluation has some bias which we have tried to mitigate. All our experiments are based around the same case study. We have tried to mitigate this issue by using an available real case study. We have also used different settings across our experiments, even if all of the experiments are based on the same case study. Thus, our experiments limit the validity of the approach to applications with the same characteristics of the presented case study. New experiments with other use cases are needed to broaden the validation scope of our approach.

The evaluation of the heuristic mainly shows the potential impact of using an ideal heuristic. More case study and experiments are needed to fully validate the value of our Models@run.time based heuristic.

6. Discussion

In our proposal, the novelty lays on defining a set of conditions and devising a technical solution that allow us to safely change the granularity level of monitoring. Thanks to the existence of these conditions and monitoring technologies is possible to implement what we call *optimistic monitoring*. It is then worth discussing whether it is possible to use the general idea of optimistic monitoring to observe applications' properties other than resource consumption. For instance, it would be useful to apply the general idea of our proposal to detect at runtime whether a system meets certain Quality of Service (QoS) requirements. To support the implementation of optimistic monitoring to observe a given property, such a property must meet the following necessary conditions:

- **Many monitoring technologies exist**. There exists more than one technology to monitor the property and they produce different performance overhead on the application execution. For instance, in our framework we provide lightweight

global monitoring based on the Java Management API [13], and a more heavy monitoring method based on bytecode instrumentation.

- **Monitoring technologies are replaceable at runtime**. It is possible to turn on/off a monitoring technology many times during the execution of an application. Moreover, the cost of these operations in term of performance overhead are *acceptable*. For instance, in Java it is possible to activate/deactivate instruction instrumentation and doing so has a reasonable cost as we show in section 4. On the contrary, it would not be possible to adapt the monitoring framework to deal with memory if only memory instrumentation was available.
- **Being optimistic makes sense**. It is reasonable to expect that the monitored property *quite often* have values that we can monitor using a *lightweight* technology. For instance, in the case of resource consumption monitoring we can expect that most misbehaviors would be detected during components' development. Hence, we only care about the specific consumption of each component once we detect a global problem.
- **Being optimistic and lazy does not make the system crash**. We must guarantee that no matter what monitoring technique is being used, we are always able to eventually detect a problem, and the detection happens on time to notify such an event to a manager in charge of fixing the system. For instance, as we show, switching among monitoring techniques might impact the time to detect what are the faulty components. However, our implementation makes the assumption that, once we detect a misuse of the CPU, we are able to switch fast enough to localized monitoring and we can still spot the faulty component consuming excessive CPU cycles.
- **There are likely culprits**. We require a mechanism to *easily* spot faulty components. This mechanism must be based on a sort of heuristic that minimizes the time required to detect the offenders. For instance, we show in section 4 how using a good heuristic greatly reduce the performance overhead and detection time.

This set of conditions might guide the implementation of a framework to monitor properties in reconfigurable software system. They are useful as questions to answer before attempting an implementation.

Properly defining contracts for each component is one of the most complicated requirements of our approach. Since we focus on finding whether a contract is violated at runtime, we consider it as a problem which is orthogonal to our approach, and we just assume the preexistence of such contracts. Nonetheless, we acknowledge than in practice is difficult to find the values to fill the kind of contract we refer to. The challenge when on of such contracts is being written is to figure out a consumption's upper bound which is close enough to the real consumption along the application execution. In this paper, we use a priori experimentation to build the contracts. To use this approach it is necessary to executed the components several time using the monitoring framework

[13] http://docs.oracle.com/javase/7/docs/api/java/lang/management/package-summary.html

with full monitoring mode activated.

Another solution that can be used to infer the contract is the use of static analysis techniques to infer resource usage contract [47–49]. Such techniques can be used only if we are in a white-box component application where we can get access to component source code.

Finally, this problem of contract can be reversed and we can use ScapeGoat to check contract correctness. Indeed, in an open-world system when we integrate existing piece of software together, it is well known that design by contract [50] was an important contribution to avoid software integration issue [51]. In that trend, contracts can also be seen as specification for reuse. Like other specifications, they will often be handwritten by humans. As these contracts will be used to check components assembly correctness, it is required that these contracts are correct. In this context, ScapeGoat can also be used to detect inconsistencies between component implementation and component contracts. When ScapeGoat detects a problem, it can come from an implementation error but it can also highlight a component contract bug.

7. Related work

The Scapegoat framework is related to component monitoring, Models@run.time, component isolation and component performance prediction approaches.

Performance and resource-consumption prediction approaches are complementary to the Scapegoat framework because they can assist in better specifying the component contracts. Some approaches require developers to provide extensive per-component metadata at design-time in order to calculate the application's overall performance or resource consumption [52, 53]. Prediction approaches have been achieved by using combinations of design-time and runtime analyses [54]. However, although many approaches to performance prediction have been proposed, none of them have obtained widespread use [55].

KAMI [56] builds performance models at design-time but uses and continually refines them at runtime. By collecting runtime data, they are able to build performance and resource consumption models that reflect real usage. They are able to adapt the application according to changes in components' behavior, but they do not use nor propose an adaptive monitoring system to minimize overhead.

State of the art monitoring systems [8–10] extract steady data-flows of system parameters, such as, the time spent executing a component, the amount of I/O and memory used, and the number of calls to a component. The overhead that these monitoring systems introduce into applications is high, which makes it unlikely for them to be used in production systems. Maurel et al. [57] propose an adaptive monitoring framework for the OSGi platform. Similar to our approach, they propose a global monitoring system that changes to a localized monitoring system when a problem is detected. However, their work is focused on CPU usage and does not consider other resources, such as memory or I/O. Exploring the Java heap to obtain useful information about resource consumption has been proposed in [37, 40]. As in our work, they account objects to the resource principal being explored (in their case to OSGi bundles) the first time an object is reached. Their solutions modify the garbage collector in order to

reduce overhead, but this causes resource accounting to be tied to, and performed on, each collection cycle. In contrast, our approach can be executed on demand, albeit at the cost of further passes over the heap.

Modifications to existing MRTEs to support lightweight instrumentation-based profilers have been proposed. In [58], an approach to instrument and (de)instrument methods on demand is proposed. The idea is to generate an additional version of each method, which includes instrumentation code. Afterwards, the runtime executes a version based on user interests. This dynamic instrumentation approach shows lower overhead than static instrumentation. Likewise, Arnold et al. [59] propose an approach to reduce the cost of performing instrumentation-based profiling. The insight of this approach is creating an additional version of each method where no instrumentation code is present but it is used to figure out if switching to the instrumented version is necessary. Since the switching condition can change at runtime, this approach is in fact dynamically adjusting the cost and accuracy of profiling. The results show an overhead of 6% during the profiling of CPU usage. Finally, heavy modifications to MRTEs can reduce the effort needed to perform resource accounting. For instance, due to the architecture of MRTEs such as MVM [60] and KaffeOS [61], memory accounting in these systems shows a negligible overhead. The main disadvantage of these approaches is that they are not portable; hence, in practice this prevent their adoption in production environments. Moreover, often these approaches can only deal with global monitoring or with a very specific representation of components on top of the managed runtime environment.

Gama and Donsez [62] propose using virtual machines in separate processes or using MVM isolates [60] to manage trusted and untrusted components. After an evaluation period, untrusted components can be moved to the trusted JVM if no problems are detected. This allows the main application to depend on potentially faulty components without risking severe crashes. We can also 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 [63]. In addition to process virtualization, some operating systems also propose user-space virtualization, which isolates not only the processes but also the memory, the network interface and the file system. Examples of these approaches are Jails[14] for BSD, LXC[15] and CGroups for Linux, and lmctfy[16]. All of these approaches have the drawback of limiting code and instance sharing and introduce additional overhead in cross-boundary component interactions. Furthermore, depending on the complexity of the approach, there is also overhead in having to manage multiple processes.

8. Conclusions and Perspectives

In this paper we presented ScapeGoat, an adaptive monitoring framework to perform lightweight yet efficient monitoring of Component-Based Systems. In Scape-

[14] `http://www.freebsd.org/doc/handbook/jails.html`
[15] `http://lxc.sourceforge.net/`
[16] `https://github.com/google/lmctfy`

Goat, each component is augmented with a contract that specifies its resource usage, such as peak CPU and memory consumption. ScapeGoat uses a global monitoring mode that has low overhead on the system, and an on-demand fine-grained localized monitoring mode that performs extensive checking of the components' contracts. The system switches from the global monitoring mode to the localized monitoring mode whenever a problem is detected at the global level in order to identify the faulty component. Furthermore, we proposed a heuristic that leverages information produced by the Models@run.time approach to quickly predict the faulty components.

ScapeGoat has been implemented on top of the Kevoree component framework which uses the Models@run.time approach to tame the complexity of distributed dynamic adaptations. The evaluation of ScapeGoat shows that the monitoring system's overhead is reduced by up to 93% in comparison with state-of-the-art full monitoring systems. The evaluation also presents the benefits of using a heuristic to predict the faulty component. In our second part of the evaluation, we highlighted the benefit of scapegoat on a classical web server architecture to dynamically determine faulty components. This second usage also exposes the capacity of ScapeGoat to be applied to different application domains and confirms its relatively low overhead on the running system. This paper contributes to the state of the art by providing a monitoring framework which adapts its overhead depending on current execution conditions and leverages the architectural information provided by Models@run.time to drive the search for the faulty component.

Our proposal aims at monitoring a single execution environment, in particular we implement a framework to monitor component-based systems written in Java that are executing atop a single JVM. Nevertheless, it is interesting to observe that it is possible to leverage ScapeGoat to support resource monitoring in distributed applications. This framework is recording the data about resource usage in storage which is local to each execution environment. To monitor the resource usage of a distributed application, it is enough to add an extra management layer in charge of collecting data from the different distributed nodes. This recollection of data can be done by a central agent or we can leverage a hierarchical architecture where a set of nodes are connected to a parent resource manager in order to parallelize data recollection. Afterwards, these distributed resource managers can use the information to perform decision-making on aspect such as system reconfiguration.

The work presented in this paper opens various research perspectives. Scapegoat currently uses code injection at load-time to perform fine-grain monitoring. The adaptive monitoring approach we have presented provides good results, but we believe we can reduce the overhead of CPU and memory monitoring by using a modified JVM and injecting specialized bytecode to cooperate with it. The modified JVM would account for the resources at a low-level, while the instrumentation code could provide application-level information like the component boundaries. This should result in a more efficient solution than calculating resource usage at the application-level only. A second research perspective consists in proposing appropriate reactions when the source of a problem is discovered by ScapeGoat. Indeed, reconfiguration policies when a resource-consumption problem is found could range from resource limitations for faulty components, to a replacement of the component or of part of the application, to degrading the applications functionality. In the context of distributed systems, the

set of possible reconfigurations is larger and can include moving components across the distributed infrastructure. It is necessary to choose how to efficiently reconfigure the system to deal with the discovered fault. Finally, we are interested on other ways to define contracts. In this paper we use a simple definition of contract that is based on the upper bound of resource consumption. It is clear that more accurate and rich contracts would increase the usability of the framework.

Acknowledgement

This work has been supported by the European FP7 Marie Curie Initial Training Network RELATE (Grant Agreement No. 264840). It has been also supported by the ITEA2 MERgE project, and by the French project InfraJVM (Grant Agreement No. ANR-11-INFR-0008).

References

[1] G. Galante, L. C. E. d. Bona, A survey on cloud computing elasticity, in: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, UCC '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 263–270. doi:10.1109/UCC.2012.30.
URL http://dx.doi.org/10.1109/UCC.2012.30

[2] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, U. Scholz, Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments, in: Software engineering for self-adaptive systems, Springer, 2009, pp. 164–182.

[3] V. Poladian, J. P. Sousa, D. Garlan, M. Shaw, Dynamic configuration of resource-aware services, in: Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on, IEEE, 2004, pp. 604–613.

[4] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste, Rainbow: Architecture-based self-adaptation with reusable infrastructure, Computer 37 (10) (2004) 46–54.

[5] The OSGi Alliance, OSGi Service Platform Core Specification, Release 5.0, http://www.osgi.org/Specifications/ (Jun. 2012).

[6] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, J.-M. Jézéquel, A dynamic component model for cyber physical systems, in: V. Grassi, R. Mirandola, N. Medvidovic, M. Larsson (Eds.), CBSE, ACM, 2012, pp. 135–144.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J. Stefani, The FRACTAL Component Model and its Support in Java, Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems 36 (11-12) (2006) 1257–1284.

[8] S. Frénot, D. Stefan, Open-service-platform instrumentation: Jmx management over osgi, in: Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing, UbiMob '04, ACM, New York, NY, USA, 2004, pp. 199–202.

[9] H. Kreger, W. Harold, L. Williamson, Java and JMX: Building Manageable Systems, Addison-Wesley, Boston, MA, 2003.

[10] W. Binder, J. Hulaas, Exact and portable profiling for the {JVM} using bytecode instruction counting, Electronic Notes in Theoretical Computer Science 164 (3) (2006) 45 – 64, proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006).

[11] W. Binder, J. Hulaas, P. Moret, A. Villazón, Platform-independent profiling in a virtual execution environment, Softw. Pract. Exper. 39 (1) (2009) 47–79.

[12] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, D. Watkins, Making components contract aware, Computer 32 (7) (1999) 38–45.

[13] I. Gonzalez-Herrera, J. Bourcier, E. Daubert, W. Rudametkin, O. Barais, F. Fouquet, J.-M. Jézéquel, Scapegoat: an adaptive monitoring framework for component-based systems, in: A. Tang (Ed.), Working IEEE/IFIP Conference on Software Architecture, IEEE/IFIP, Syndey, Australia, 2014.
URL https://hal.inria.fr/hal-00983045

[14] G. S. Blair, N. Bencomo, R. B. France, Models@run.time, IEEE Computer 42 (10) (2009) 22–27.

[15] E. M. Dashofy, A. van der Hoek, R. N. Taylor, An infrastructure for the rapid development of XML-based architecture description languages, in: 24th International Conference on Software Engineering, ICSE '02, ACM, New York, NY, USA, 2002, pp. 266–276.

[16] N. Medvidovic, R. N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, IEEE Trans. Softw. Eng. 26 (2000) 70–93.

[17] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The Koala Component Model for Consumer Electronics Software, Computer 33 (3) (2000) 78–85.

[18] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, K. Pohl, A journey to highly dynamic, self-adaptive service-based applications, Autom. Softw. Eng. 15 (3-4) (2008) 313–341.

[19] T. T. Johnson, S. Bak, S. Drager, Cyber-physical specification mismatch identification with dynamic analysis, in: Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems, ICCPS '15, ACM, New York, NY, USA, 2015, pp. 208–217. doi:10.1145/2735960.2735979.
URL http://doi.acm.org/10.1145/2735960.2735979

[20] V. Panzica La Manna, Local dynamic update for component-based distributed systems, in: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE '12, ACM, New York, NY, USA, 2012, pp. 167–176. doi:10.1145/2304736.2304764.
URL http://doi.acm.org/10.1145/2304736.2304764

[21] J. Zhang, H. J. Goldsby, B. H. Cheng, Modular verification of dynamically adaptive systems, in: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD '09, ACM, New York, NY, USA, 2009, pp. 161–172. doi:10.1145/1509239.1509262.
URL http://doi.acm.org/10.1145/1509239.1509262

[22] V. Grassi, R. Mirandola, N. Medvidovic, M. Larsson (Eds.), Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, part of Comparch '12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012,

ACM, 2012.

[23] L. Baresi, E. Di Nitto, C. Ghezzi, Toward open-world software: Issue and challenges, Computer 39 (10) (2006) 36–43.

[24] M. Caporuscio, M. Funaro, C. Ghezzi, Architectural issues of adaptive pervasive systems, in: G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, B. Westfechtel (Eds.), Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, Vol. 5765 of Lecture Notes in Computer Science, Springer, 2010, pp. 492–511.

[25] D. Perez-Palacin, J. Merseguer, S. Bernardi, Performance aware open-world software in a 3-layer architecture, in: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering, WOSP/SIPEW '10, ACM, New York, NY, USA, 2010, pp. 49–56. doi:10.1145/1712605.1712614.
URL http://doi.acm.org/10.1145/1712605.1712614

[26] B. Porter, Runtime modularity in complex structures: A component model for fine grained runtime adaptation, in: Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering, CBSE '14, ACM, New York, NY, USA, 2014, pp. 29–34. doi:10.1145/2602458.2602471.
URL http://doi.acm.org/10.1145/2602458.2602471

[27] D. Zheng, J. Wang, B. Kerong, Research of context-aware component adaptation model in pervasive environment, in: Proceedings of the 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, DASC '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 496–501. doi:10.1109/DASC.2014.95.
URL http://dx.doi.org/10.1109/DASC.2014.95

[28] F. Irmert, T. Fischer, K. Meyer-Wegener, Runtime adaptation in a service-oriented component model, in: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS '08, ACM, New York, NY, USA, 2008, pp. 97–104. doi:10.1145/1370018.1370036.
URL http://doi.acm.org/10.1145/1370018.1370036

[29] C. Ghezzi, A. Motta, V. Panzica La Manna, G. Tamburrelli, Qos driven dynamic binding in-the-many, in: Proceedings of the 6th International Conference on Quality of Software Architectures: Research into Practice - Reality and Gaps, QoSA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 68–83.

[30] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan, A generic component model for building systems software, ACM Transactions on Computer Systems (TOCS) 26 (1) (2008) 1.

[31] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, J.-B. Stefani, A component-based middleware platform for reconfigurable service-oriented architectures, Software: Practice and Experience 42 (5) (2012) 559–583.

[32] B. Morin, O. Barais, G. Nain, J.-M. Jezequel, Taming dynamically adaptive systems using models and aspects, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 122–132. doi:10.1109/ICSE.2009.5070514.

[33] M. Léger, T. Ledoux, T. Coupaye, Reliable dynamic reconfigurations in a reflective component model, Component-Based Software Engineering (2010) 74–92.

[34] R. Johnson, B. Woolf, The Type Object Pattern (1997).

[35] B. Meyer, Applying "design by contract", Computer 25 (10) (1992) 40–51.

[36] L. Alfaro, T. Henzinger, Interface theories for component-based design, in: T. Henzinger, C. Kirsch (Eds.), Embedded Software, Vol. 2211 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 148–165.

[37] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, B. Folliot, I-jvm: a java virtual machine for component isolation in osgi, in: Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on, 2009, pp. 544–553. doi:10.1109/DSN.2009.5270296.

[38] W. Binder, Portable and accurate sampling profiling for java, Softw. Pract. Exper. 36 (6) (2006) 615–650.

[39] G. Czajkowski, T. von Eicken, JRes: a resource accounting interface for java, in: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '98, ACM, New York, NY, USA, 1998, pp. 21–35.

[40] D. W. Price, A. Rudys, D. S. Wallach, Garbage collector memory accounting in language-based systems, in: Proceedings of the 2003 IEEE Symposium on Security and Privacy, SP '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 263–274.

[41] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006, pp. 169–190.

[42] G. Tesauro, N. K. Jong, R. Das, M. N. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, in: Autonomic Computing, 2006. ICAC'06. IEEE International Conference on, IEEE, 2006, pp. 65–73.

[43] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, M. Tricoire, Multi-tier diversification in internet-based software applications, IEEE Software 99 (PrePrints) (2015) 1. doi:http://doi.ieeecomputersociety.org/10.1109/MS.2014.150.

[44] A. Avizienis, The n-version approach to fault-tolerant software, IEEE Trans. Softw. Eng. 11 (12) (1985) 1491–1501. doi:10.1109/TSE.1985.231893.
URL http://dx.doi.org/10.1109/TSE.1985.231893

[45] S. Forrest, A. Somayaji, D. Ackley, Building diverse computer systems, in: Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), HOTOS '97, IEEE Computer Society, Washington, DC, USA, 1997, pp. 67–.
URL http://dl.acm.org/citation.cfm?id=822075.822408

[46] B. Baudry, S. Allier, M. Monperrus, Tailored source code transformations to synthesize computationally diverse program variants, in: C. S. Pasareanu, D. Marinov (Eds.), International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, ACM, 2014, pp. 149–159. doi:10.1145/2610384.2610415.
URL http://dl.acm.org/citation.cfm?id=2610384

[47] B. K. Pasquale, G. C. Polyzos, A static analysis of i/o characteristics of scientific applications in a production workload, in: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Supercomputing '93, ACM, New York, NY, USA, 1993, pp. 388–397.

[48] M. Hofmann, S. Jost, Static prediction of heap space usage for first-order functional programs, in: ACM SIGPLAN Notices, Vol. 38 of 1, ACM, 2003, pp. 185–197.

[49] S. Jost, K. Hammond, H.-W. Loidl, M. Hofmann, Static determination of quantitative resource usage for higher-order programs, in: ACM Sigplan Notices, Vol. 45 of 1, ACM, 2010, pp. 223–236.

[50] B. Meyer, Applying'design by contract', Computer 25 (10) (1992) 40–51.

[51] J.-M. Jezequel, B. Meyer, Design by contract: The lessons of ariane, Computer 30 (1) (1997) 129–130.

[52] S. Becker, H. Koziolek, R. Reussner, Model-based performance prediction with the palladio component model, in: Proceedings of the 6th international workshop on Software and performance, WOSP '07, ACM, New York, NY, USA, 2007, pp. 54–65.

[53] M. D. Jonge, J. Muskens, M. Chaudron, Scenario-based prediction of run-time resource consumption in component-based software systems, in: In Proceedings of the 6th ICSE Workshop on Component-based Software Engineering (CBSE6, IEEE, 2003, p. pages.

[54] M. Autili, P. D. Benedetto, P. Inverardi, A hybrid approach for resource-based comparison of adaptable java applications, Science of Computer Programming 78 (8) (2013) 987 – 1009.

[55] H. Koziolek, J. Happe, A qos driven development process model for component-based software systems, in: Proceedings of the 9th international conference on Component-Based Software Engineering, CBSE'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 336–343.

[56] C. Ghezzi, G. Tamburrelli, Predicting performance properties for open systems with kami, in: R. Mirandola, I. Gorton, C. Hofmeister (Eds.), Architectures for Adaptive Software Systems, Vol. 5581 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 70–85.

[57] Y. Maurel, A. Bottaro, R. Kopetz, K. Attouchi, Adaptive monitoring of end-user osgi-based home boxes, in: Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering, CBSE '12, ACM, New York, NY, USA, 2012, pp. 157–166.

[58] M. Dmitriev, Profiling java applications using code hotswapping and dynamic call graph revelation, SIGSOFT Softw. Eng. Notes 29 (1) (2004) 139–150.

[59] M. Arnold, B. G. Ryder, A Framework for Reducing the Cost of Instrumented Code, in: SIGPLAN Conference on Programming Language Design and Implementation, 2001, pp. 168–179.
URL `http://citeseer.ist.psu.edu/arnold01framework.html`

[60] G. Czajkowski, L. Daynàs, Multitasking without compromise: a virtual machine evolution, ACM SIGPLAN Notices 47 (4a) (2012) 60–73.

[61] G. Back, W. C. Hsieh, J. Lepreau, Processes in KaffeOS: isolation, resource management, and sharing in java, in: Proceedings of the 4th conference on Sym-

posium on Operating System Design & Implementation - Volume 4, OSDI'00, USENIX Association, Berkeley, CA, USA, 2000, pp. 23–23.
URL http://dl.acm.org/citation.cfm?id=1251229.1251252

[62] K. Gama, D. Donsez, A self-healing component sandbox for untrustworthy third party code execution, in: Proceedings of the 13th international conference on Component-Based Software Engineering, CBSE'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 130–149. doi:10.1007/978-3-642-13238-4_8.

[63] J. Löwy, COM and. NET: Component Services, O'Reilly Media, Inc., 2001.