UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOÃO CLAUDIO RODRIGUES AMÉRICO

A study of the impact of real-time constraints in Java/OSGi applications

Trabalho de Graduação.

Prof. Dr. Cláudio Fernando Resin Geyer

Orientador

Porto Alegre, junho de 2010.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a minha familia, pelo carinho, atenção e apoio desmedidos ao longo não só dos anos de estudo, mas de toda a minha vida.

Agradeço à Universidade Federal do Rio Grande do Sul e ao Instituto de Informática pelo ensino de alta qualidade que me foi conferido durante a minha graduação e pela oportunidade que me foi dada de complementar a minha formação através de um intercâmbio e de um duplo diploma na França.

Agradeço ao professor Claudio Fernando Resin Geyer pela orientação durante a realização deste trabalho e durante os meus estudos de duplo diploma.

Agradeço a Walter Rudametkin pelo seu apoio, conselhos, orientação e amizade durante a minha estadia na França. Agradeço também ao professor Diogo Onofre Gomes de Souza, que me acompanha desde o ensino médio e que muito me apoiou ao longo desses anos.

Gostaria de agradecer aos colegas e amigos que fiz durante os meus anos de estudos na UFRGS pelos momentos de descontração, pelas risadas e pelas partidas de truco. Não obstante, agradeço também aos meus amigos fora da UFRGS, que souberam compreender todas as vezes em que foram trocados por noites de trabalho.

Por fim, agradeço a todos que tenham contribuído de alguma forma para o meu trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

NOTA EXPLICATIVA

Esta monografia é fruto de um trabalho de um ano (fev/2009 - ago/2009 e jan/2010 - jun/2010) desenvolvido durante meu duplo diploma. Ela é constituída de duas partes. A primeira mostra a instrumentação de bytecode para transformação de aplicações escritas com a API standard de Java em aplicações que utilizam a API de tempo-real de Java e o seu impacto no determinismo das aplicações. Este trabalho constituiu o meu projeto de fim de estudos na ENSIMAG e validou o meu primeiro ano de duplo diploma. Entretanto, visto que ele foi desenvolvido durante o meu estágio na Bull, seu conteúdo foi classificado pela empresa como sendo confidencial à escola e à empresa. Mesmo assim, um artigo apresentando a ferramenta que foi construída durante o estágio foi escrito e submetido à 18ª Conferência Internacional em Sistemas de Rede e Tempo-Real (RTNS 2010). O artigo submetido pode ser encontrado nos anexos deste trabalho.

A segunda parte do trabalho, descrita nessa monografia, é uma continuação da primeira. Em suma, a abordagem de instrumentação foi utilizada para transformar um servidor de aplicação Java Enterprise Edition cujo núcleo é constituído de uma plataforma OSGi em tempo-real. Tendo em mente que isto não foi suficiente, a segunda parte do trabalho investiga a questão do dinamismo da plataforma OSGi e o impacto do mesmo sobre restrições temporais. Para resolver esta questão, foi utilizada uma abordagem baseada no congelamento da arquitetura das aplicações hospedadas na plataforma e a utilização de acordo de níveis de serviço para a aceitação das reconfigurações. Este trabalho foi desenvolvido durante o meu estágio em um laboratório de pesquisa especializado em OSGi, validando o meu segundo ano de duplo diploma.

ABSTRACT

Real-time requirements and software runtime adaptation are two needs of today's software. On the one hand, the most important characteristics in real-time applications are their predictable behavior and deterministic execution time. On the other hand, runtime (also called dynamic) adaptive software have as main characteristic the capability of being modified and updated at execution time, what makes it more flexible and robust.

In the context of Java platform, many solutions for dealing with both aspects separately have been developed. Among the real-time solutions, the most popular is the Real-Time Specification for Java (RTSJ) and its implementations, which offers a complete API for the development of real-time applications in Java. Likewise, the OSGi Service Platform is one of the most popular solutions for developing and deploying dynamic adaptive software. One of the reasons of its popularity is the fact that it combines both service-oriented computing and component-based design concepts in a simple service-oriented component model. The OSGi Service Platform has become the *de facto* platform for developing flexible and modular software, and many Java applications are being migrated and developed by means of its component model. However, due to the popularization of real-time solutions, some of these applications may have timing constraints which cannot be respected because of the platform dynamic behavior and the fact that service-oriented component-based applications architectures may change at execution time. The goal of this project is to suggest and evaluate solutions for this issue.

Keywords: Service-oriented architecture, component-based development, service-oriented component models, dynamic adaptive software, real-time, RTSJ, Java, OSGi.

RESUMO

Restrições de tempo-real e adaptação de software em tempo de execução são duas necessidades frequentes nos sistemas modernos. De um lado, as características mais importante dos sistemas de tempo-real são a sua preditibilidade e o seu tempo de execução determinista. De outro lado, aplicações adaptáveis em tempo de execução tem como principal característica a capacidade de serem modificáveis e atualizáveis em tempo de execução, o que as torna flexíveis e robustas.

No contexto da plataforma Java, muitas soluções lidando separadamente com estes dois aspectos foram desenvolvidas. Entre as soluções de tempo-real, a mais popular é a Especificação de Tempo-Real para Java (RTSJ) e suas implementações, que oferecem uma API completa para o desenvolvimento de aplicações de tempo-real em Java. Da mesma forma, a plataforma de serviços OSGi é uma das soluções mais populares para o desenvolvimento e implementação de software dinamicamente adaptável. Um dos motivos para sua popularidade é o fato de que ela combina conceitos de ambas abordagens orientada a serviços e baseada em componentes. A plataforma de serviços OSGi tornou-se o padrão de facto para o desenvolvimento de sistemas flexíveis e modulares, e muitas aplicações tem sido migradas e desenvolvidas utilizando o seu modelo de componentes. Entretando, com a popularização das soluções de tempo-real, algumas destas aplicações podem apresentar restrições temporais que não poderão ser respeitadas devido ao comportamento dinâmico da plataforma e ao fato de que a arquitetura das aplicações abrigadas na plataforma OSGi podem mudar ao longo de sua execução. O objetivo deste trabalho é de avaliar este problema e sugerir soluções para o mesmo.

Palavras-chave: Arquiteturas orientadas a serviços, desenvolvimento baseado em componentes, modelos de componentes orientados a serviços, tempo-real, RTSJ, Java, OSGi, software dinâmicamente adaptável.

1 INTRODUCTION

1.1 *Context*

Dynamic adaptive behavior and real-time requirements are common needs of today's software. While the former primes for flexibility and unforeseen modifications in the environment at runtime, the latter concerns predictability and determinism of application's response times. Many solutions for dealing with both aspects separately have been fairly recently developed for the Java platform. One of the most adopted real-time solutions for Java is the Real-Time Specification for Java (RTSJ) and its implementations, which offers a complete API for the development of real-time applications in Java. At the same, the popularization of component-based design and service-oriented computing concepts for the development of flexible and modular applications in Java are responsible for the creation of service-oriented component models and the specification of service platforms.

One of the most popular service platforms is the OSGi Service Platform. Its original intention was to become an open specification to develop and deploy services in home gateways, but it has become the *de facto* standard for developing general-purpose Java applications in a modular and flexible way. Its popularization in several domains is due to, among many other things, its adoption by the Eclipse Foundation for developing plug-ins for their IDE. In the OSGi framework, we can create service-oriented and component-oriented applications. Nowadays, the OSGi Specification is in its 4th release. This specification, that in former days addressed embedded systems, was extended to cover many other domains, such as mobile phones, industrial supervision, automobiles and more recently a whole set of Java Enterprise Edition application servers.

However, some of the Java applications which are being migrated or developed with service-oriented component models present among their requirements the need for predictable response times. Although we can assure real-time behavior by means of real-time Java

solutions, predictability is a hard-to-guarantee property due to the fact that service-oriented component-based application architectures are dynamic and can evolve during application execution.

1.2 *Objectives*

This work aims to analyze the effects and impacts of the dynamicity provided by service-oriented component models over the predictability needed by Java applications with real-time requirements. The OSGi Service Platform is used as a concrete example for these issues. Solutions to these issues are proposed and a prototype is presented to give them a tangible form and to demonstrate their feasibility.

1.3 *Document outline*

This document is structured as follows. In this first chapter, the context in which this work is inserted was presented, along with our general objectives. The second chapter of this document shows the state of the art of the principal related domains: Real-time Java applications, runtime software adaptability and the OSGi Service Platform, the platform chosen to exemplify the studies. In the third chapter, we identify the issues in dynamic real-time adaptive applications and instantiate the problem in the OSGi Service Platform. In the fourth chapter, a proposition for such problems is presented. The fifth chapter presents an implementation to experiment and validate our proposition. Finally, the sixth chapter presents the conclusions and perspectives for future research created by this study.

2 STATE OF THE ART

2.1 *Introduction*

The increasing software complexity and the need for dynamism in its execution led developers to look for new ways of designing, constructing and maintaining applications. In order to address this problem, new domains in software engineering have emerged such as dynamic software architectures. Software architecture is a structural representation of a system, in terms of components and interactions. Dynamic architectures are software architectures which can evolve and adapt at run time, thus increasing flexibility and availability. Most of the approaches used to implement runtime evolutions are based on modularity and the use of proxies to intercept execution flow.

The OSGi Service Platform is a service platform which addresses the lack of support for modularity in Java applications [HAL10]. OSGi components interact through the *publish, find* and *bind* service interaction pattern: service providers publish their services into a registry, while service clients query the registry to find available services to use. Since modules can be installed, updated and uninstalled at any time, services can appear and disappear dynamically. The lack of modularity is not the only problem we find in the Java platform. The unpredictability introduced by its weak mechanisms for handling priority-based scheduling and automatic garbage collection makes Java unsafe for designing real-time applications [NILSS02]. For this purpose, the Real-Time Expert Group created the Real-time Specification for Java (RTSJ) [BOL00], whose implementations provide additional mechanisms for building deterministic Java applications.

Gradually, more and more Java applications have been migrating to the OSGi framework, due to its flexibility and dynamism. However, due to the popularization of RTSJ in the world of real-time programming, some of these applications have real-time requirements and the dynamism offered by the OSGi platform is a factor that may

compromise the deterministic behavior in the applications' response times. The aim of this chapter is to present an overview of Real-time Java, runtime software evolution and OSGi Service platform in order to elucidate the challenges we can meet once we deal with dynamically adaptive real-time applications.

2.2 *Real-Time Java*

Java [ARN00] has become one of the most popular general purpose languages. This popularity is in part due to its portability, reusability, security features, ease of use, robustness, rich API set and automatic memory management. Java has many advantages over traditional languages for programming, such as C and C++ [TYM98]. In addition, nowadays it is arguably easier to find programmers with Java skills than those experienced with Ada or C. However, the same Garbage Collector that eases development is one of the main reasons why Java was not used to design critical, embedded and real-time applications. Indeed, garbage collection introduces unpredictable execution times [BACO03]. As a result, many different solutions were designed to improve the determinism of conventional Java.

In the next sections, we introduce Java and its real-time solutions. In section 2.2.1, we give the basic concepts of real-time computing. Section 2.2.2 contextualizes real-time concepts in the Java platform, explaining the principal shortcomings found for using Java to design real-time systems. In section 2.2.3, we present solutions developed to overcome the difficulties discussed in the last section and implement real-time systems in Java.

2.2.1 Real-time Computing

Before presenting Java and its real-time extension, we present in this section a brief overview of some important aspects in real-time computing. In the first subsection, we introduce some basic concepts and definitions. The second and third subsections detail two requirements for real-time systems: predictability and determinism. The concept of real-time operating system is presented in the fourth subsection, while the fifth and sixth subsections discuss important concerns in real-time scheduling and real-time programming languages.

2.2.1.1 Definitions and Concepts

Real-time systems[1] differ from other information systems in the fact that their correctness depends on both functional and temporal aspects [STA92]. **Timing correctness** requirements proceeds from the impact of a real-time system upon the real world. These

[1] Also known as "reactive systems"

requirements, in turn, may be expressed in the form of timing constraints for the set of cooperating tasks which compose the system. Depending on the tasks arrival pattern, timing constrains may be *periodic*, *sporadic* or *aperiodic***,** as shown in Figure 1. Periodic tasks are invocated within regular time intervals, while arrival times in sporadic and aperiodic tasks are unknown; however, the time interval between two releases of an aperiodic task is only known to be greater or equal to zero, while sporadic tasks have a time interval between two releases always greater than or equal to a constant [ISO00, BRUNO09]. Ideally, temporal constraints are explicitly specified for each task by the system designer. In order to satisfy different timing constraints, services and algorithms used by real-time systems must be executed in bounded time.

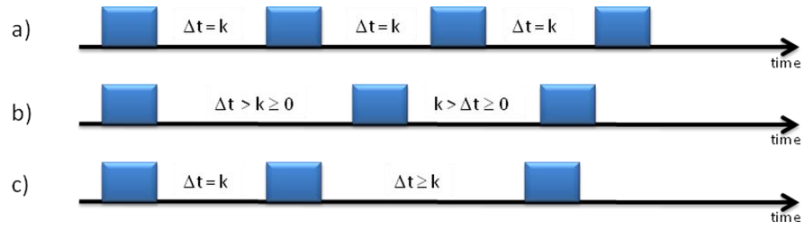

Figure 1. Periodic (a), aperiodic (b) and sporadic (c) tasks.

Those timing constrains, also known as **deadlines**, can be relative to an event or absolute, precising a point in time for a task to complete its execution. Depending on the enforcement of deadlines, real-time systems may be divided into **hard, firm** and **soft real-time systems** [SHI94]. In hard real-time systems, all deadlines must be strictly enforced to avoid safety issues (e.g., weapon systems, nuclear power plants, automated transport systems) and to ensure system correctness [BRUNO09]. Firm real-time systems are those in which results produced as soon as the deadline expires become useless for the application, but consequences are not very severe. In soft real-time systems, the need for strict deadlines is more or less replaced by the need for homogeneous response times in order to ensure acceptable levels of service, i.e., minimize response-time deviations. Missed deadlines are interpreted as degraded service quality, and should be avoided; nevertheless the system continues to operate. Besides temporal constraints, real-time processes may have other types of constraints, such as resource, performance and availability constraints, which can also be found in non real-time applications.

The concepts described above highlight the most important characteristics in real-time applications: they do not necessarily have to be fast, but they must be **predictable** and **deterministic**.

2.2.1.2 Predictability in Real-Time Systems

In fact, the notion of predictability may vary from one application to another. For some applications, a predictable system is one where it is possible to mathematically demonstrate, at design time, that all timing constraints will be met. This requires one to know all the tasks and their characteristics. Another possible definition is that in predictable systems, timing constraints are guaranteed for critical tasks; other tasks may offer probabilistic or run-time guarantees, thus one cannot predict at design time if a task will meet its deadlines. Most of the cases, worst case values are assumed in order to provide deadline guarantees [STA90]. Two different types of techniques are used to ensure predictability [RICHT03]:

Schedulability analysis: Given a set of tasks, their priorities, temporal constraints and worst case execution time (WCET), the aim of this technique is to identify a schedule which satisfies all the constraints for all the tasks. Many scheduling analysis techniques are popular in the real-time systems domain. Some examples are rate-monotonic scheduling, earliest deadline first, Round-Robin and static order scheduling [LIU73, RASM08, SRI02].

Formal verification: In this technique, the system and its properties are formalized into logic statements or timed automata and the timing constraints are formally verified through model checking or theorem proving techniques [BER81, LAR95].

However, both solutions are not enough for designing predictable real-time systems [HUA05].

2.2.1.3 Determinism in Real-time Systems

Determinism and predictability are closely related, because one results in the other. A deterministic system has the ability of ensuring the execution of an application despite external factors that can unpredictably cause a perturbation (and thus alter the functionality, performance and response time) [Bruno09]. Application behavior is then more or less fixed, in such a way that all deadlines can be met and predictability is achieved.

Real-time systems are not all about deadlines. Two additional metrics are related to determinism:

- **Latency:** Latency is the time between an event and a system response to that event. Usually, developers focus on minimizing system latency. However, in real-time applications the aim is to normalize it, that is, to make the latency of a system a known

and predictable quantity. Measuring latency includes finding and measuring all the sources of latency in a system, which is often far from being an easy task. Assuming that we know the time it takes to process an event, another way to measure latency is by measuring response time[2].

- **Jitter:** In the context of real-time systems, detects unsteadiness in system latency. Simply averaging latency or measuring it for one event does not guarantee that a system is deterministic. The distribution and standard deviation of latency responses are common ways of measuring jitter.

Depending on the application, having a normalized latency and a small jitter may be more important than deadline enforcement.

2.2.1.4 Real-time Operating Systems

All aspects of a system must be taken into account in order to design a real-time system. Initially, real-time systems were implemented for specific use with dedicated hardware. Nowadays, hardware support is still required, but due to the advances in modern computer hardware, even general-purpose systems can be used to solve real-time problems. Today, real-time concerns are concentrated in the software layer, more specifically in the operating system software. Some of the features that the underlying operating system must provide in order to support real-time applications are real-time task scheduling, priorities, resource management, high-resolution clocks and low-latency interrupts [STA04].

Real-time operating systems (RTOS) are operating systems that support applications with timing constraints, providing a deterministic environment while maintaining logical correctness in its results [CED07]. Some basic paradigms found in traditional operating systems cannot be applied to RTOS's. For instance, it is not important for a RTOS to have support for security or file systems. However, predictable interrupt handling and scheduling with timing and dependability constraints is required. Real-time behavior for firm and soft real-time applications can be achieved by enhanced conventional operating systems with some real-time features, but for hard real-time applications a RTOS is necessary.

The IEEE Portable Operating System Interface for Computer Environments (POSIX 1003.1b) [IEE96] defines a list of basic services required by a RTOS. Some of these are asynchronous and synchronous input/output (I/O), memory locking, semaphores, shared

[2] That is, latency plus the event processing time.

memory, execution scheduling, timers, interprocess communication (IPC), real-time files and real-time threads. Other basic requirements [BAS05] are preemptability, multi-task support, deterministic synchronization mechanisms, real-time priority levels, dynamic deadline identification and predefined latencies for task switching and interrupt mechanisms.

2.2.1.5 Real-Time Scheduling

Scheduling problems are present in many computer science domains (e.g., parallel processing), but constraints in real-time systems make the problem considerably different. Instead of minimizing the total time required to execute all tasks, real-time schedulers must focus on respecting all deadlines. Besides timing constraints, resource (to provide to the tasks what is needed to a successful execution) and precedence (to ensure proper system behavior) constraints must also be considered when scheduling real-time tasks [SHI94].

A scheduling algorithm is a set of rules which determines the task to be executed at a particular moment and the duration of its execution [LIU73]. Real-time scheduling algorithms have as finality guaranteeing that heedless of system charge critical tasks will meet their deadlines. They can be classified along several dimensions depending on tasks characteristics [STA92, BRUNO09]. The most important distinctions made are:

- **Preemptive/Non-preemptive Scheduling:** Most operating systems allow assigning a priority to a task. Thus, higher-priority tasks have precedence over lower-priority tasks. Preemptive algorithms allow interrupting the execution of a lower-priority task to execute a higher-priority one, while in non-preemptive algorithms a thread executes until it completes its tasks. Non-preemptive algorithms have the advantage of avoiding dispatch latency and thrashing, but they may not respect precedence constraints [MUN70].
- **Static/Dynamic Scheduling:** In static scheduled systems, tasks assignment is determined a priori to processors [PEN89]. Thus, priority and other scheduling parameters are determined when tasks first enter the system. Dynamic scheduling allows tasks to be dispatched as the system is running, based on the system state and on scheduling parameters that may change over time.
- **Periodic tasks-oriented/Aperiodic tasks-oriented Scheduling:** Some algorithms may only deal with periodic tasks, while others handle only aperiodic tasks. Dealing with periodic tasks is obviously easier than dealing with aperiodic tasks, once the

latter may produce unforeseen events. There are some schedulers which can support both types of tasks.

- **Guarantee-Based/Best-Effort Based Scheduling:** As the name suggests, guarantee-based algorithms are pessimist algorithms which ensure that all tasks will meet their timing constraints. Tasks which can disrupt the system are not allowed to execute. On the contrary, best-effort based algorithms are optimistic and, once a new task arrives into the system, they do their best to ensure that all threads will respect their deadlines or will be completed close to them. The former is best-suited for time-critical systems, and the latter is preferred for soft real-time systems.
- **Optimal/Feasible (Heuristic) Scheduling:** A feasible schedule is a schedule where all the tasks reach the end of their execution on or before their deadlines. Feasible scheduling algorithms search for feasible schedules to guarantee the system's real-time behavior, while optimal scheduling algorithms always find a feasible scheduling (if one exists).

A scheduling algorithm may fall into many of these categories. Some examples are First-In-First-Out (FIFO, dynamic-priority), Earliest-Deadline-First (EDF, dynamic preemptive), Shortest-Execution-Time-First (SETF, dynamic non-preemptive), Least-Slack-Time (LST, dynamic-priority), Latest-Release time-First (LRT, static time-driven), Rate-Monotonic (RM, static fixed-priority preemptive) and Deadline Monotonic (DM, static fixed-priority preemptive) [BRUC98, BRUNO09].

2.2.1.6 Real-time Programming Languages

Using traditional technologies and methodologies in real-time development is costly and difficult [NILSE96]. Thus, since the early days of computer programming field, many programming languages have been used to develop real-time applications. These languages support the expression of timing constraints and deterministic behavior in at least one of three different ways:

- Eliminating constructs with indeterminate execution times,
- Extending existing languages, or
- Being constructed jointly with an operating system.

The most important requirement for real-time programming languages is the guarantee of predictable, reliable and timely operation. For this purpose, every software activity must be

expressible in the language through time-bounded constructs; hence, its execution timing constraints can be analyzable. In addition, a real-time language should be reliable and robust, what implies in strong typing mechanisms and modularity. Modularity also eases a "programming-in-the-large" approach, in view of the fact that many real-time systems are large systems used in military and finance domains. Process definition and synchronization, interfaces to access hardware, interrupt handling mechanisms and error handling facility are also desirable features for real-time languages [STO92].

Assembly, procedural and object-oriented languages are the most common general-purposed languages used for developing real-time systems. Despite of the lacking of most the high-level language features (such as portability, modularity and high-level abstractions), assembly language provides direct access to hardware and an economic execution. Procedural languages, such as C and FORTRAN, BASIC, Ada and Modula extensions, offer desirable properties of real-time software, like versatile parameter passing mechanisms, dynamic memory allocation, strong typing, abstract data typing, exception handling and modularity. C++ and real-time extensions for Java are examples of object-oriented languages used in real-time development, which benefits from some procedural languages advantages and adds higher level programming abstractions. Even though these abstractions increase developers' efficiency and code reuse, mechanisms underlying them may introduce unpredictability and inefficiency into real-time systems [LAP06]. Besides real-time extensions for general-purposed languages, many highly-specialized or research-only languages for real-time applications were also created along the last 40 years. These include Eiffel, Pearl, LUSTRE, MACH, MARUTI and ESTEREEL, among others [SCHW95, LAP06].

2.2.2 Java versus Real-Time Systems

This section presents a brief overview of the Java technology. In the first subsection, we introduce Java and its main features. Then, in a second subsection, we show why it is standard form is not suitable for real-time applications.

2.2.2.1 An Overview of Java

Java technology was designed by Sun Microsystems in 1995 and consists of the Java language definition, a definition of the standard library and the definition of an intermediate instruction set, along with an accompanying execution environment. Originally, Java was created to facilitate the development of networked devices small embedded systems, but due to its portable and flexible capabilities, Sun released it to the general public for Internet and

high-level interface applications development. Though the syntax of the Java's programming language is based on C/C++, Java was designed to eliminate some error-prone features of these languages, such as pointer arithmetic, unions, *goto* statements and multiple inheritance, improving developers productivity.

Java introduced a different execution model. First, Java programs are translated into a machine-independent byte-code representation. Then, this byte-code can run in any device which implements the Java Virtual Machine (JVM), a software system which understands and executes byte-code instructions (See figure 2). In the first implementations of the Java virtual machine, those instructions were interpreted, but for performance issues other translation techniques, including ahead-of-time and just-in-time (JIT) compilation, were included into later implementations.

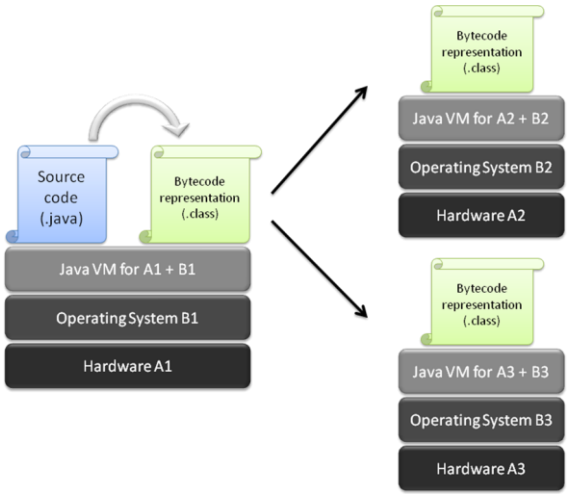

Figure 2. Platform independence in Java

Besides platform independence, other remarkable features in Java are:

- It is simple and easy to learn and to use;
- Robustness due to automatic garbage collection, type safety, byte-code analysis and run-time checks;
- Built-in multithread and multitask support;
- Easy access to remote sources;
- And lazy class loading (classes are loaded dynamically are they are needed).

2.2.2.2 Java Issues in Real-Time Applications

Java has several features which would be desirable for developing real-time applications; however, in its standard form, Java is not well-suited for it [NILSS02]. Some of the reasons why Java is inadequate for the development of real-time software are:

- **Memory footprint:** Standard JVMs needs at least tens of megabytes in memory, what is not adequate for embedded systems. Solutions addressing this issue are, for example, the Java 2 Micro Edition [J2ME], IVM [IVE02] and JVM hardware implementations [IAB00, HAR01]. The formers are significantly limited compared to the standard API, while the latter is a platform-specific solution.
- **Performance and execution model:** Byte-code interpretation reduces the overall performance of Java applications [KAZ00]. In order to solve this issue, JIT compilers were designed to compile Java byte-code into native code at run-time. However, running a compiler at runtime, besides requiring a considerable amount of memory, raises scheduling issues, what implies in latency and lack of determinism.
- **Scheduling:** Java defines a very loose behavior of threads and scheduling. Threads with higher priority are executed in preference to threads with lower priority. However, low priority threads can preempt high priority threads. Although this protects from starvation in general purpose applications, it violates the precedence property required for real-time applications, and may introduce indeterminism in execution time. In addition, the wakeup of a single thread (through the method `notify()`) is not precisely defined.
- **Synchronization:** Synchronized code uses monitors to protect critical code sections from multiple simultaneous accesses. Even though Java implements mutual exclusion, it does not prevent unbounded priority inversions, an unacceptable condition for real-time systems.
- **Garbage Collection:** Automatic memory management simplifies programming and avoids programming errors. At the same time, traditional garbage collection implies in pauses at indeterminate times impose delays of unbounded duration.
- **Worst Case Execution Time:** Key concepts for object-oriented programming support in Java are method overriding and the use of interfaces for multiple inheritance.

However it usually requires a search on the class hierarchy or dynamic selection of functions at runtime, what complicates WCET analysis.

- **Dynamic Class Loading:** In order to dynamically load classes, they must be resolved and verified. This is a complex and memory-consuming task, which may introduce an unforeseen delay in execution time depending on factors as the speed of the medium and the classes' size.

As we may see, standard Java implementations do not provide mechanisms for the reliable and deterministic execution of real-time applications. However, most of these issues do not come from the language, but from the Java execution environment.

2.2.3 Real-time Solutions for Java

The advantages of Java over languages traditionally used to design real-time systems resulted in several efforts in late 1990s in extending the language. We present in the next sections the main solutions developed to make Java more appropriate for real-time applications.

2.2.3.1 Early Work in Real-Time Java

The simplest extension proposed for supporting real-time applications in Java was Real-Time Java Threads [MIY97], in 1997. However, this support was very rudimentary. A more sophisticated and complete solution was proposed by Nilsen [NILSE98] providing both high-level abstractions for real-time systems and low-level abstractions for hardware access, the Portable Executive for Reliable Control (PERC). Another approach, based on CSP Algebra, Occam2 and the Transputer microprocessor, was proposed by Hilderink [HIL98]. Other attempts include hardware implementations of the JVM [BACK98] or integrating it to the operating system [MCGH98].

Unfortunately, much of this work was fragmented and did not have a clear direction. Thus, the US National Institute of Standards and Technology (NIST) reunited several companies to general guidelines and requirements for real-time extensions to Java. The NIST requirements resulted in two initiatives: The Real-Time Specification for Java (RTSJ) [BOL00], backed by Sun and IBM; and the Real-Time Core Extension for the Java Platform (RT Core) [JCO00], backed by the J Consortium[3], based on the PERC system. However,

[3] Supported by groups such as Microsoft, HP, Siemens and Newmonics.

contrarily to the RTSJ, the RT Core proposed modifications to the Java language syntax, which was not well-accepted by the Java community.

2.2.3.2 The Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) defines real-time behavior in the Java Platform by means of a collection of classes, constraints to the behavior of the virtual machine, an API and additional semantics. Seven areas were identified as requiring enhancements to enable the creation, analysis, execution and management of real-time tasks:

- **Thread Scheduling and Dispatching:** RTSJ introduces the concept of schedulable objects (real-time threads, asynchronous event handlers and their subclasses), objects which the base scheduler manages. The RTSJ's base scheduler is priority-based, preemptive, with at least 28 unique priorities[4], run-to-block[5] and can perform feasibility analysis for a schedule. Schedulable objects have parameters classes bound to it, representing resource-demand (scheduling, memory or release) characteristics.
- **Memory Management:** RTSJ provides extensions to the garbage collected model memory, supporting memory management without interfering with real-time code deterministic behavior. It allows the allocation of short and long-lived objects in memory areas that are not garbage collected. Besides the traditional heap memory, where objects lifetime is defined by their visibility, and the JVM stack, which allocates a private stack for each created thread, three memory areas were included to the Java programming model: scoped memory, which manages objects short-lived objects whose lifetime is defined by a scope; physical memory, allowing objects to be allocated in a specific physical memory region; and immortal memory, an area containing objects which may be referenced by any schedulable object. Scoped and Immortal memories are not garbage-collected.
- **Synchronization and Resource Sharing:** RTSJ requires priority inversion avoidance algorithms for implementing the Java keyword synchronized[6]. In addition, it

[4] In addition to the values 1 to 10 defined by conventional Java Threads, but with higher execution eligibility.

[5] It means that a schedulable object in execution will continue running until it either blocks or is preempted by a higher-priority schedulable object

[6] Commonly used to share serialized resources

introduces wait-free queues to allow the communication between schedulable objects and objects subject to garbage-collection.

- **Asynchronous Event Handling:** To allow a closer interaction with the real-world and its inherent asynchrony, RTSJ allows the creation of asynchronous events as well as handlers for these events. These handlers are scheduled and dispatched, just like threads. Timer class represents events whose occurrence is time-driven and is a specific form of asynchronous events. These timers are based on Clock objects, which represent the system clocks, as uniformly and accurately as allowed by the underlying hardware.
- **Asynchronous Transfer of Control (ATC):** RTSJ allows the asynchronous transfer of the current point of logic execution. This mechanism also allows the execution of iterative algorithms, which refines gradually the result precision, transmitting the results at the expiration of a precise time bound.
- **Asynchronous Real-time Thread Termination:** RTSJ provides a safe mechanism for abnormally stopping threads and transferring control, contrarily to the deprecated stop and destroy methods in class Thread, which could leave shared objects in inconsistent states or lead to deadlocks.
- **Physical Memory Access:** RTSJ defines classes allowing to directly byte-level access the physical memory and create objects in physical memory. In addition, it provides manager classes to appropriately access and create objects with specific characteristics.

New exceptions were also included, along with new treatments surrounding ATC and memory allocation. The RTSJ implementations are based on its version 1.0.2. Besides the requirements defined by the RTSJ itself, additional requirements for implementations were defined by the Mackinac team [BOL05]. Nowadays, a new JSR[7] was created addressing the RTSJ version 1.1, the JSR 282.

2.2.3.3 Implementations of the RTSJ

Since the official release of the RTSJ in 2002, several implementations of the specification were already developed. We list some them in Table 1.

[7] Java Specification Request, documents proposing technologies for additions to the Java platform.

Implementation	Developer	Certification TCK JSR-001[8]	Implementation type	Platform compatibility	Java Compatibility
RTSJ-RI[9]	Timesys	Yes	Reference	RTLinux[10]/x86	JRE 6.0
Java Real-Time System[11]	Sun Microsystems/ Oracle	Yes	Commercial	Solaris, RTLinux/x86; Solaris/SPARC	JRE 5.0
Websphere Real-Time[12]	IBM	Yes	Commercial	RTLinux/x86	JRE 6.0
Real-Time JRE[13]	Apogee	Yes	Commercial	RTLinux /x86	JRE 5.0/J2ME
Jamaïca Virtual Machine[14]	Aicas	No	Commercial	RTLinux, SunOS, Solaris/x86	JRE 5.0
J-Rate (Java Real-Time Extension)[15]	University of California	No	Open Source (GPL)	RTLinux/x86, PowerPC	JRE 1.4-5.0
OVM (Open Virtual Machine)[16]	Purdue University	No	Open Source (BSD)	RTLinux/x86, PowerPC; OS X/PowerPC	JRE 1.4-5.0

Table 1. RTSJ implementations

2.2.3.4 Other Real-Time Solutions for Java

Not all real-time solutions for Java are RTSJ-based. Indeed, some solutions claim that RTSJ's region-based allocation mechanism takes away the simplicity of the base Java, being error-prone and incurring non-trivial runtime overheads due to dynamic memory access checks [PIZ08]. In addition, it is not well suitable for hard real-time applications due to performance issues [PIZ10]. In order to overcome those issues, many independent solutions were already proposed. We list some of them in the next paragraphs.

- **Oracle JRockit Real Time**[17] provides a Java-based soft real-time computing infrastructure. It contains a deterministic garbage collector, which ensures short pause times. JRockit Real Time supports Java applications running on Java SE 6 and J2SE 5.0 runtime environments. *Oracle WebLogic Real-Time* is a version of Oracle JRockit Real-Time offered with *Oracle WebLogic Suite*. Supported platforms include commercial distributions of Linux (Oracle Enterprise, Novell SUSE, Red Hat

[8] RTSJ was registered at JCP as JSR-000001. TCK stands for Technology Compatibility Kit, a suit of tests to verify if an implementation is compliant to a given JSR.

[9] http://www.timesys.com/java/. Although classes can be compiled by a JDK 6.0 compiler, they must remain 1.3-compatibles. This implementation is based on J2ME, thus some J2SE classes are not present.

[10] RTLinux is a set of adaptations made in Linux kernels for supporting real-time. Numerous commercial and free versions are available.

[11] http://java.sun.com/javase/technologies/realtime/index.jsp

[12] http://www-01.ibm.com/software/webservers/realtime/

[13] http://www.apogee.com/products/rtjre. This solution is based on IBM's J9 Virtual Machine.

[14] http://www.aicas.com/jamaica.html. Generated classes must be 1.4-compatibles.

[15] http://jrate.sourceforge.net/. This is a GCJ-based solution, so there is not an exact correspondence with a J2SE version.

[16] http://www.cs.purdue.edu/homes/jv/soft/ovm/index.html. Actually, OVM is a framework for generating customizable virtual machines. This framework is also based on GCJ Compiler.

[17] http://www.oracle.com/technology/products/jrockit/jrrt/index.html

Enterprise and Red Flag AS) and Microsoft Windows over x86 architectures, and Sun Solaris over SPARC architectures.

- **SimpleRTJ**[18] is an implementation of the Java virtual machine optimized to run on devices with limited amount of memory and without RTOS support. It requires only 18-24 KB of memory to run. Despite of its small size, it includes core features like multithreading, interfaces, garbage collection and exception handling. SimpleRTJ uses pre-linked applications and classes, which reduces start-up times and delays to resolve symbolic references. It was designed to run on 8, 16 and 32 bit microcontrollers.
- **Fiji VM**[19] [PIZ10] is a virtual machine implementation which compiles Java 1.6 byte-code ahead-of-time directly to ANSI C. This virtual machine consists of a compiler, a runtime library and open-source class libraries. The runtime system contains an on-the-fly concurrent real-time garbage collector. As well as in RTSJ-based implementations, region-based memory allocation is also supported.
- Despite the fact that **Aonix PERC**[20] is based on RTSJ libraries, it defines its own class hierarchy. PERC integrates a static analysis system to verify scope safety and resource requirements for hard real-time systems.
- **JOP** [SCHO07] is a hardware implementation for the Java virtual machine. In introduces a processor architecture which simplifies WCET analysis. Java byte-code is translated to a stack-based instruction set (called microcode) which can be executed in a 3-stage pipeline. Byte-code translation and interrupt handling are also pipelined, increasing time predictability.
- **Juice** [COR03] is an interpreted J2ME virtual machine designed for real-time embedded systems running on the NUXI [SAN02] operating systems. In Juice, heap memory is divided into pre-fixed size blocks. Free memory blocks are organized into a linked list, while blocks allocated by Java objects are connected through a hierarchical structure. Thus, object allocation and deallocation depends only on the object size. Memory is garbage-collected only when new objects have to be allocated, and the collector's execution time is proportional to the size of the object to be allocated.

[18] http://www.rtjcom.com/
[19] http://www.fiji-systems.com/
[20] http://www.aonix.com/perc.html

As we have seen in this section, real-time software requires reliability and predictability. However, many current and future real-time applications are *dynamic*, that is, external conditions may require modifications and adaptations at runtime. In the next section, we present an overview of dynamic software adaptation.

2.3 *Dynamic Software Adaptation*

Complexity issues have been present in computer science since its early days. The first programming complexity problems were solved through the use of data structures, the development of algorithms and scope separation. Most of what we know now as software engineering nowadays comes as result of this problem. The first works about the importance of structuring software systems were led by Dijkstra [DIJ68] and Parnas [PAR72], in the late 1960s. These were the basis for a software engineering discipline called Software Architecture. Software architecture studies ways of structuring software systems, by representing its software components, their interconnections and the rules concerning their design and evolution over time [GAR93]. Many aspects of a system can be addressed in its architectural description, such as its properties, functional and non-functional requirements and different configurations.

Dynamic software architectures are architectures in which the composition of interacting components changes during system's execution. This behavior is known as runtime evolution or adaptation [TAY09]. Advances in this field have been boosted by the emergence of ubiquitous computing [WEIZ93] and the growing demand for autonomic computing [KEP03]. The main motivations for runtime adaptive software are the risks, costs and inconveniences presented by the downtime of software-intensive systems of environment changes [ORE08].

In this section, we explore concepts and techniques used for dynamic software adaptation. The subsection 2.3.1 discusses dynamic software architectures. Subsection 2.3.2 introduces definitions for software adaptation. In the subsection 2.3.3, we introduce dynamic adaptive systems and list some enabling technologies for designing them. Subsection 2.3.4 lists some adaptive frameworks. Finally, we present solutions for real-time adaptive software in the subsection 2.3.5.

2.3.1 Dynamic Software Architectures

Structuring systems as interacting components is the result of years of research in software engineering and one of the solutions proposed in order to deal with scalability,

evolution and complexity issues in software. Jointly with compositional techniques, it eases the system's design, analysis and construction process, by providing a higher level of abstraction.

Dynamic software modification is a useful capability which can be applied in many domain applications. An example is dynamic software update, in which the application is able to update itself to fix bugs and add new features without requiring a stop and a restart. Nonstop and critical systems, such as air-traffic control systems, enterprise and financial applications, which must provide continuous service, are examples of applications in which dynamic update is required [MAG96].

However, this flexibility has a cost: safety. Although we can perform modifications which were not planned during the design phase, we cannot anticipate the effects of a dynamic modification. It can affect predictability, which is inadmissible for safety-critical systems; it can download modules and require more space on disk, unacceptable for constrained and embedded systems; it can add unsafe modules, which can make the whole platform crash; or it can bring the application to a wrong state after applying changes. Figure 3 shows a dynamic software architecture in which a runtime modification was performed. Generally, even though a system architecture may look static at compile time, runtime updates may imply in modified and additional modules.

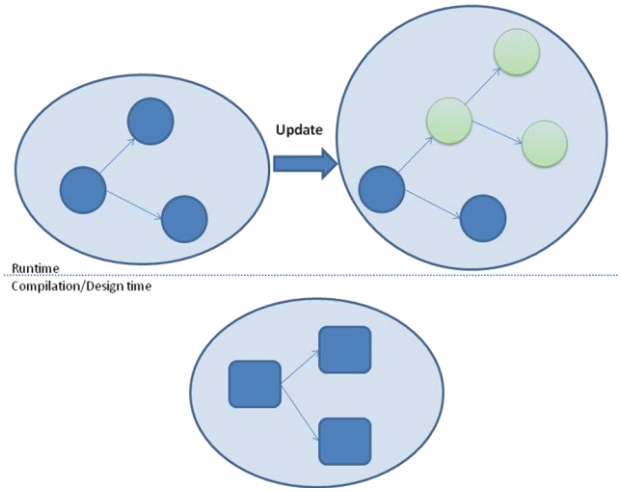

Figure 3. Dynamic software evolution

Transparency is an important property for software adaptation frameworks. It decreases the burden on the developers, which do not have to anticipate the whole set of possible evolutions of a software system. In order to do that, frameworks must perform every

necessary modification in the binary code, injecting code or intercepting calls. We present in the next subsections concepts and approaches used by these dynamic adaptive frameworks.

2.3.2 Definitions and Concepts

According to the standard glossary of software engineering systems [IEE90], *adaptability* is defined as "the ease with which a system or component can be modified for use in applications or environments other than those for which it was designed". Adaptability differs from *adaptiveness* in that the first defines the ability of the software to be reconfigured, while the second designates the ability of the software to reconfigure itself [AKK07].

Two approaches are generally used to implement software adaptation: *parametric adaptation*, in which system variables are modified in order to change system behavior; and *compositional adaptation*, in which the system components are added or replaced to better adapt a program to its environment. Parametric adaptation allows tuning application parameters, but it offers a limited adaptation mechanism, since it is not possible to add behaviors in the software system. In addition, compositional adaption permits an application to be recomposed dynamically during execution. This dynamic recomposition is called dynamic (or runtime) adaptation, which is different from static (or build time) adaptation, where the modifications are made before the system is running (e.g., in the source code or in the requirements).

Other possible classifications are *manual/automatic adaptations*, based on the way in which the adaptation is managed, and *functional/technical adaptations*, based on the properties that are going to be modified [CAN06].

2.3.3 Approaches for Dynamic Software Adaptation

We call *dynamic compositional adaptive software* the software which is able to adapt itself and its components at run time to handle resource variability and other operational environment changes. Most approaches implementing dynamic compositional adaptation are based on dynamically linking and unlinking components or indirectly intercepting and redirecting interactions among software entities [FOX09]. Various techniques may be used to achieve this, such as, manipulating function pointers, aspect weaving, proxies or middleware interception [MCK04]. In the next subsections we list some of the approaches which allow designing and constructing dynamic adaptive software.

2.3.3.1 Separation of Concerns

Separation of concerns is a software engineering principle which emphasizes the separation of the application logic from crosscutting concerns (such as quality of service, synchronization, security and fault tolerance) at conceptual and implementation levels. It allows for simplifying development and maintenance, making software easier to be reused [HUR95].

Nowadays, one of the most used approaches for separating concerns is Aspect-Oriented Programming (AOP). This programming paradigm is based on an entity called *aspect*. An aspect is a technical consideration from a crosscutting concern in an application. Even though AOP is language-independent, it requires a special compiler, called *aspect weaver*. Summarizing, the weaver can insert aspect code in specific code locations, known as *join points*. Aspects code may contain *advices*[21] and *intertype declarations*[22]. In order to select join points to insert aspects code, we create *point cuts*.

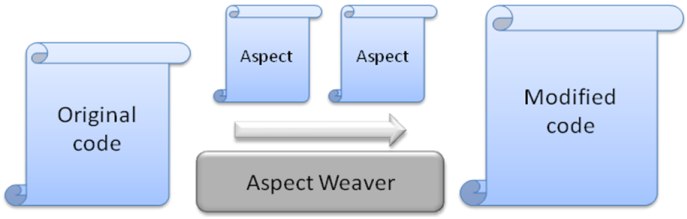

Figure 4. Aspect Weaving

Aspect weaving can be performed at run time (dynamic) as well as at compile time (static), even though static strategy is more popular.

2.3.3.2 Computational Reflection

Computational reflection is a programming language technique which allows a system to keep information about itself (introspection) and use this information to adapt its behavior (intercession). Based on what can be modified, we can distinguish two types of reflection: structural and behavioral (or computational) reflection. In the former, the system structure can be dynamically modified, while in the latter only the system computational semantics can be modified [MAE87].

[21] A piece of code that can be activated and inserted into system join points

[22] Used to add members in a module

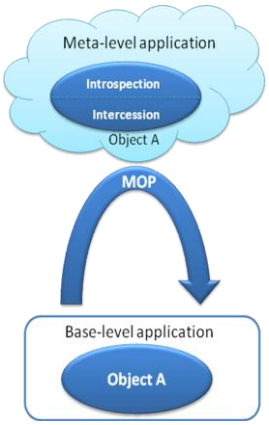

Figure 5. Meta-object protocol for computational reflection

Most runtime reflective systems are based on Meta-Object Protocols (MOP). These protocols specify the way a base-level application[23] may access its meta-level[24], in order to dynamically adapt its structure and behavior.

Some programming languages, such as Common Lisp Object System (CLOS) and Python, have native reflection mechanisms.

2.3.3.3 Dynamic Service-Oriented Architectures

Service-oriented architecture (SOA) is an architectural style and a programming model based on the service concept. The main principles in SOA are loose coupling, abstraction, reusability and composition. A service is a software unit whose functionalities and properties are declaratively described in a service descriptor. Services can be composed and orchestrated to create more complex services. Lazy binding and encapsulation mechanisms allow services to have a loose coupling between the implementation and its interface [PAP03].

SOAs provide a framework for guiding the design, development, integration and reuse of applications. Figure 7a shows the interaction among the different actors in SOAs. Service providers register the description of its services in a service register. Service consumers query the service register to discover and select services. Then, after negotiating and according service usage terms, the service consumer is bound to the service provider. Service certifiers can be used to monitor if both parts respect the accords.

[23] Application expressed by a programming language
[24] The computational object model implementation at the execution environment

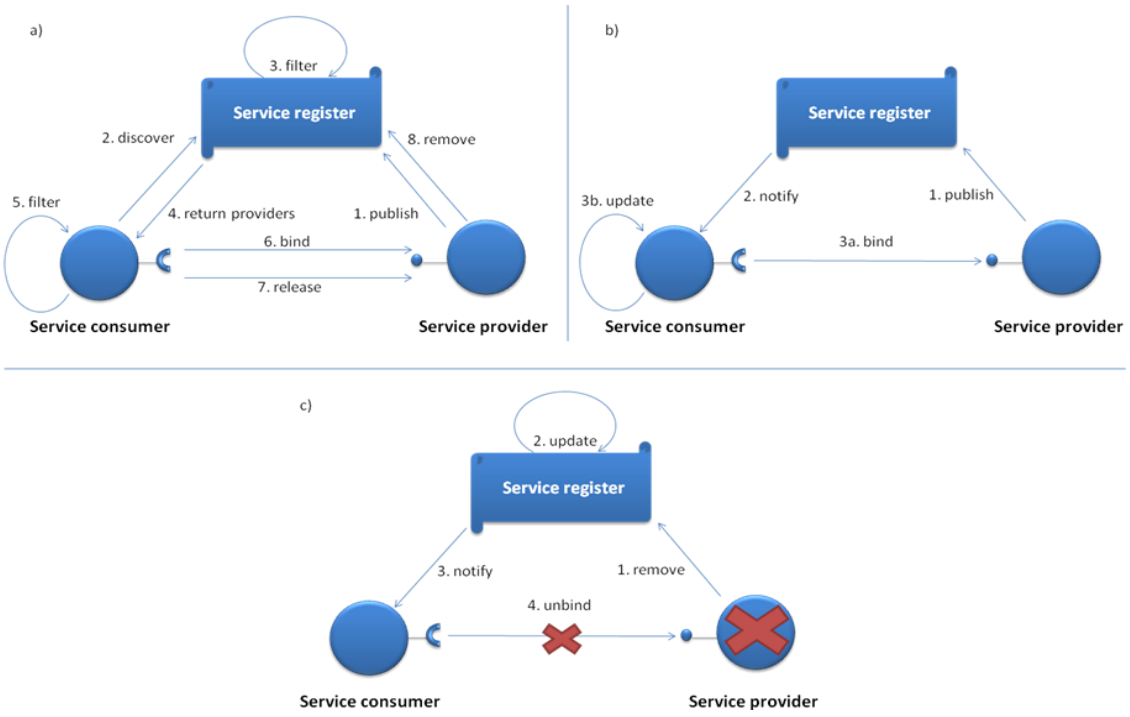

Figure 6. a) Publish-find-bind service interaction pattern b) Service update and service publish in DSOA c) Service removal in DSOA

Dynamic SOA (D-SOA) adds the dynamism to SOA. This dynamism can be depicted in two different concepts: dynamic availability [CER04a], which refers to the ability of the service to be available or unavailable at any moment; and dynamic properties modification, which designates the fact that service properties (thus, service description) can be modified at run time. Dynamic availability allows systems to evolve without downtime and dynamic properties modification may be useful in dynamic context adaptation or negotiation. Figures 7b and 7c show respectively the dynamic service publish/update and removal. In both cases, the service consumer must be notified of the context changes [RED02].

2.3.3.4 Component-Based Design

Component-based design uses components as the underlying software abstraction. Software components are software units, which are composed in order to build a complete system, with contractually specified interfaces and explicit context dependencies. These units can be independently developed and deployed. Composition can be static or dynamic, depending on when the developer is able to add, remove or reconfigure components (compile time or run time, respectively). Dynamic adaptation can be performed using late binding mechanisms, which allows coupling components at runtime through well-defined interfaces. This architectural style also promotes software reuse, reduces production cost (because software systems are built from existing code) and shortens time to market [CLE95].

The foundation of a component-based methodology lies on its software component model, which defines what components are, how they can be constructed, assembled, deployed, etc. Examples of component models are Architecture Description Languages (ADL), Web services and JavaBeans [LAU07].

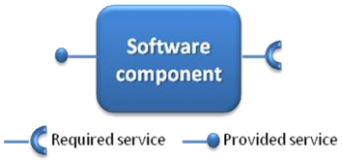

Figure 7. Common representation of a component

Even though SOA and component-based are main focused in different actors (while the component-oriented approach focus on the provider's view, easing the deployment of new functionalities, SOA focuses on the consumer's view, to supply functions to consumers which do not care about service implementation), SOA is considered as an evolution of component-based design, introducing abstract business model concepts such as contract, service provider and service consumer. In addition, SOA introduces dynamism and substitutability into static component-based design. Thus, both approaches are often combined in service-oriented component models [ROU08].

2.3.3.5 Other Factors

Many other approaches are also used to provide dynamic software adaptation. Many of them are based on middleware - layers of services separating applications from operating systems and network protocols. Most adaptive middleware works by intercepting and modifying messages.

Other technologies used to adapt software architectures at run-time are P2P[25], software design patterns, agent-oriented programming and generative programming [MCK04].

2.3.4 Dynamic Adaptive Frameworks

In the following paragraphs, we list some frameworks with dynamic adaptive features [FOX09]. This is far from being an exhaustive list; there are many other frameworks that are not listed here. Our selection was based on choosing frameworks which present the characteristics outlined in the last sections.

[25] Peer-to-peer, a distributed network architecture where participants make a portion of their resources directly available to other participants, without central coordination.

Intense research has been developed along the last years in the component models domain. Among the works in dynamic adaptive frameworks are based in component models. One of the most important component models in the literature is the **CORBA Component Model (CCM)** [OMG], which an OMG extension to the specification CORBA 2.0, incorporating ideas from component-based development into CORBA[26]. CCM components are abstractions for dynamically loadable packages containing CORBA interfaces which can be easily linked together. Components can be created, assembled and deployed by means of an extension to CORBA's Interface Definition Language. At runtime, component instances are managed and created by containers. Components can include *facets* (provided interfaces for use by other components), *receptacles* (required interfaces from other components), *event source* (logical data channel on which components publish events) and *event sinks* (logical channel for event consuming). Other component models commonly referenced are **Fractal** [BRUNE02b], a hierarchically-structured component model, and **Koala** [OMM00], a component model developed by Philips Research mainly used to develop electronic products software. Both use ADLs in order to specify the software high-level structure. While Fractal supports dynamic architectural reconfiguration by means of computational reflection, Koala is restricted to switching between statically-defined components.

Recent component models introduce new features in order to provide more flexibility. For instance, **iPOJO** [ESC07] is a runtime service-oriented component model which can be used to develop applications over the OSGi service platform. iPOJO injects POJOs[27] at runtime, through the management of service providing and dependencies. iPOJO provides component containers with manage all service interaction and allows adding non-functional properties, such as persistency, security and autonomic management. Component dependencies and non-functional properties are handled by handlers, which are specified in the component type metadata and are plugged on the component instance at runtime. iPOJO also manages the lifecycle of the instances, which are considered as valid if all its plugged handlers are valid, or invalid otherwise. Similarly, **Mobility and Adaption Enabling Middleware (MADAM)** [GEI07] is a component model which has incorporated special features for adaptation. One important concept for this framework is the realization plan, a composition plan which contains combination of components specified by the designer.

[26] Common Object Request Broker Architecture, software architecture to develop components or ORBs.
[27] Plain Old Java Object, ordinary Java objects

MADAM provides an adaptation manager and a middleware framework for runtime adaptation.

With regard to separation of concerns, **AspectJ** [KIC01] is the most popular implementation of AOP concepts for Java. It extends Java language by adding constructions to create and model aspects. AspectJ has features to influence the system behavior at runtime by means of its dynamic join point model. Code can be inserted at method calls, method call reception and method execution, field access, exception handler invocation and object or class initialization. In addition, AspectJ can statically add new members to the class.

Without a doubt, **Web Services** [PAP03] were the responsible for popularizing the service-oriented approach. They allow the interoperable machine-to-machine communication over a network. A web service is a service, identified by a URI[28] whose service description (which is made using WSDL[29], a XML[30]-based language) and transport (services interact by means of SOAP[31] calls carrying XML data) is performed using open Internet standards. Service discovery uses a UDDI[32] protocol to locate candidate services and their properties. Due to the interoperability provided by Web services, the latter have been used to implement cross-enterprise transactions and message flows. Even though web services enable dynamic software architectures, it does not allow self management. Another adaptive framework which uses the service-oriented approach is **Jini** [ARN99], a service platform developed by Sun Microsystems which provides a federated infrastructure for deploying services dynamically in a network. Services are defined by Java interfaces or classes. They must be published in registries, which actually are search services. When entering a Jini architecture, service providers and consumers broadcast an announcement, which is received by these search services and answered, in order to make the new member to know the registries. Service consumers are notified about the availability of the services they are using. Once registries are not entities but services, there are no registry delegation mechanisms.

DynamicTAO [SCHM02] presents an approach different from the ones which were cited in this chapter. It emerged of an object-oriented approach. It is an extension to TAO

[28] Uniform Resource Identifier
[29] Web Services Description Language
[30] Extensible Markup Language
[31] Simple Object Access Protocol
[32] Universal Description, Discovery and Integration

("The ACE ORB[33]"), a standard CORBA Object Request Broker. ORB components can be remotely linked, reconfigured and replaced and code can be uploaded to substitute a component implementation. TAO was extended by means of reflective middleware techniques.

2.3.5 Real-time Dynamic Adaptive Systems

As said before, the correctness of real-time systems is related to real world timing constraints. That is because real-time systems usually interact with entities in the real world. However, the real world is extremely dynamic. Real world entities appear and disappear, combine and separate. Thus, real-time software entities must also be capable of adapting itself to these changes, which may not be possible to specify during the system design. At the same time, software must ensure predictable real-time behavior under both normal and abnormal operating conditions [BIH92]. Real-time adaptive systems may be used to implement real-time systems which need flexibility, adaptive systems whose interactions with other software entities must meet real-time requirements, or systems which present both characteristics.

One approach to real-time adaptive systems is the Real-time Service-Oriented Architecture (RT-SOA), an extension of SOA which aims to include timing constraints in many SOA aspects, such as modeling, composition, orchestration, deployment, policy, enforcement and management [TSA06]. The need for RT-SOA comes from enterprises, many of whom have already adopted SOA for many of their systems, but cannot do the same for their real-time applications due to the lack of strict predictability in current SOA solutions. Many research works are dedicated to this subject [TSA06, PAN09, CUC09], but IT companies such as IBM, Microsoft and HP are also interested in real-time solutions for enterprises. Another useful application for RT-SOA is the support of remote critical care [MCGR08]. It is also worth to mention the IRMOS European Project[34], which investigates on the use of real-time technologies and SOAs for networking, computing and storage levels.

Real-time services are particularly important in contexts where performance requirements demands are not only fast, but predictable operations. A RT-SOA framework must provide real-time communication infrastructure, consider real-time properties (like maximal response time, service capacity and maximal degree of concurrency) in service

[33] Object Request Broker, middleware software which allows programs to make calls in a program from a machine to another via a network

[34] More information at www.irmosproject.eu

specifications, provide dynamic service composition, real-time service deployment, real-time policy engine and dynamic real-time scheduling for application services and framework operations. RT-SOA applications may use Service Level Agreements[35] to express their goals and Quality-of-Service (QoS) constraints. Moreover, the real-time middleware framework must be build at the top of a real-time operating system in order to benefit of a fully preemptive kernel.

Another common approach to the development of real-time dynamically adaptive applications is the use of component-based design. The most important principle considered when building component-based real-time software is the *principle of composability*, in which validated properties (such as timeliness and testability) must not be affected by the system integration [KOP98]. So far, many different approaches were used in the component-based software engineering literature in order to introduce real-time requirements in component models.

- The **Quality Objects** (QuO)[36] framework is a QoS adaptive layer which runs on existing middleware such as Java RMI and CORBA and supports the specification and the implementation of QoS requirements, system elements to measure and provide QoS and the behavior for dynamic adaptation of QoS. QuO allows the developer to use aspect-oriented software development techniques to separate QoS concerns from application logic. Even though it is possible to specify runtime variations of QoS, we cannot compose and configure complex adaptive behaviors. In addition, this framework cannot be deployed using standard configuration tools and description languages. In order to bypass these limitations, Sharma et al [SHA04] proposed the use of components, called *qoskets components*, to encapsulate and re-use adaptive QoS systemic behaviors.
- Kramer and Magee described in [KRA90] a process called *freezing of application components*, in which the whole component activity is stopped. Wermelinger in [WER97] improved the algorithm by blocking only involved components. This way, interruption time is minimized, because only affected connectors must be blocked. Rasche and Polze [RASC05], based on both works, presented a technique for dynamic

[35] A negotiated agreement between the service customer and the service provider, which defines the level of service being delivered [VER99]. SLAs are presented in the chapter 4.

[36] More information in http://quo.bbn.com

reconfiguration of component-based real-time software, in which the application is blocked during a bounded time and the loading of new components and removal of old components is performed before and after the interruption time.

- Stewart described in [Stewart1997] a dynamically reconfigurable real-time software framework by means of port-based objects. However, in order to keep the framework simple, the approach assumes that each object correspond to an independent process[37], a very limiting assumption. Processes obtain information through input ports and send information through output ports. Processes have no knowledge as to the origin or the destiny of the information obtained or sent through these ports. In addition, processes have resource ports, which connect to sensors and actuators, via I/O device drivers. Object communicate by means of state variables stored in global and local tables. The global table is stored in shared memory.
- Many research works focus on component models for building RTSJ-compliant applications [ETI06, PLS08, DVO04, HU07]. Most of them provide higher-level abstractions for creating real-time threads and/or for real-time memory management, in order to alleviate the development process. However, dynamic adaptation issues are only treated by [Plšek2008].
- **RTComposer**, a framework described in [ALU08], is also built atop of RTSJ, but is based on formal specification of scheduling constraints with automata. Components are scheduled in a flexible way, which may vary according to dynamic conditions, such as varying load, platform capabilities and components configuration. Another programming environment in Java for creating real-time components is the Exotasks project [AUE07], which focus mainly on memory isolation and fast garbage collection.
- **MyCCM-HI** [BOR09], **SOFA-HI** [PRO08] and **Blue-ArX** [WEIC04] are component-based frameworks which support dynamism via modes, i.e., applications can have many possible architectures that can be switched among each other at well-defined points and primarily target embedded real-time applications [HOS10]. SOFA-

[37] "An independent process does not need to communicate or synchronize with any other component in the system" [Stewart1997]

HI is still under development, while MyCCM-HI and BlueArX are ready for use. BlueArX is already used by Bosch in the automotive control domain.

- The **Component-Integrated ACE ORB** (CIAO) [WAN03] and **Cardamom**[38] are implementations of the Corba Component Model which supports real-time and other QoS aspects. CIAO applies aspect-oriented techniques to decouple QoS aspects from application components (separation of concerns) and allows the composition of real-time behaviors. In its turn, Cardamom addresses safety-critical systems.

Other techniques include the use of concurrent classloaders [PFE04], agents [ZHA00, BRE02a] and function blocks [BRE02b].

Another framework which is gradually becoming a de-facto standard for developing dynamic adaptive software and therefore was chosen to elucidate the concepts presented in this work is the *OSGi Service Platform*, which will be briefly presented in the next section.

2.4 *OSGi Service Platform*

Modularity is the main approach in order to deal with the increasing software complexity issues. Among the several benefits enabled by breaking the system under smaller and highly cohesive parts are *reuse, abstraction, division of labour* and *ease of maintenance* [BAR08].

Java is currently one of the most popular and used programming languages available. We have seen in the first sections of this document that it offers many flexible advantages. However, it does not support modularity natively. First of all, its access modifiers do not address logical system partitioning. In order to call for code in another package, the latter code must be declared as being public, which makes it visible to everyone else. It could be avoided by putting the class with the dependency in the same package as the one with the required code, but if those classes are logically unrelated, this would impair the application's logical structure. Second, Java's class path ignores code versions and does not allow for explicit dependencies. Classloader manipulation could be used to address this issue, but this is error-prone and low level [HAL10].

[38] http://cardamom.ow2.org/

The whole problem lies in the fact that Java uses *Jar* files as deployment units. Inasmuch as they do not have a corresponding runtime concept, their content is concatenated in the class path without the possibility of declaring explicit dependences and without a versioning mechanism. This forces developers to merge unrelated code, which implies low cohesion. In addition, the information hiding problem generates tightly coupled modules, due to the fact that the public modifier allows us to access internal implementation details.

The OSGi service platform is a module system for Java which adds a middleware layer over the platform in an effort to fill this gap and provide additional capabilities. We will discuss the approach used by the OSGi service platform in the next subsections. Subsection 2.4.1 introduces some concepts and definitions in OSGi. The three OSGi framework layers (module, lifecycle and service) are presented in subsections 2.4.2, 2.4.3 and 2.4.4 respectively. In subsection 2.4.5 we list some OSGi implementations. To conclude this section, we present works which integrate real-time and OSGi in section 2.4.6.

2.4.1 Definitions and Concepts

The OSGi service platform is a Java-based specification defined by the OSGi Alliance, a consortium of around forty companies founded in 1999. The role of this group is to define new releases and certify the implementations of the specification. The first releases of the OSGi specification were oriented to residential gateways. However, nowadays the OSGi platform is used in many different domains, like mobile telecommunications, enterprise application servers and plug-in-oriented applications.

The OSGi specification defines a way to create true modules (*bundles*, in OSGi terminology) and to make them interact at runtime. Bundles are actually Jar files with metadata specifying their symbolic name, version and dependencies.

The central idea of OSGi modularization is that each bundle has its own classloader, and consequently, its own class path. In order to allow interactions among bundles, OSGi uses a mechanism of explicit package imports and exports. Class requests are delegated among classloaders based on the dependency relationship between bundles. The matching between imported and exported packages is implemented by the OSGi platform. The explicit import/export mechanism also allows for package versioning and information hiding (all classes are bundle-private by default). In addition, the OSGi platform allows bundles to be dynamically installed, updated and uninstalled, without requiring the platform to stop and

restart. Besides the deployment mechanisms, the specification defines a Java non-distributed service platform, which allows services to be dynamically published and consumed.

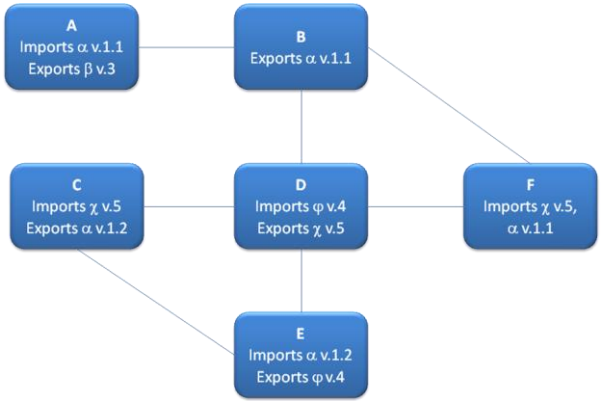

Figure 8. OSGi bundles and dependency resolution.

The OSGi Service Platform specification is divided into two parts: *OSGi framework* and *OSGi Standard Services*. While the first is the runtime which provides the functionality of the OSGi platform, the second defines APIs for common tasks. In turn, the framework is divided into three layers: *Module Layer*, concerned about code sharing and packaging; *Lifecycle Layer*, which focus on the runtime module management; and *Service Layer*, which deals with modules interaction and communication. We will discuss about these layers in the next subsections.

2.4.2 Module Layer

The module layer is the responsible for the bundle management. As said before, bundles are the unit of modularization unit of the OSGi platform, in the form of a JAR file with resources and additional metadata on its manifest file. This additional information includes human-readable information, bundle identification[39] and code visibility[40], which will be used to perform bundle dependency resolution (See figure 8). Nonetheless, unlike JAR files which are just physical containments for classes, bundles combine both the logical and the physical aspects of modularity.

Each bundle has its own classloader, providing code isolation to the platform. This classloader is responsible to load bundle's resources and classes and resolving imported classes, performing runtime verifications according to visibility rules and ensuring class loading happens in a predictable and consistent way. Besides code isolation, the module layer

[39] Symbolic name, version and manifest version.
[40] Bundle class path, imported and exported packages.

provides logical boundary enforcement, version verification, reuse improvement, configuration flexibility and configuration verification.

2.4.3 Lifecycle Layer

On the top of the Module Layer there is the Lifecycle Layer. It deals with the execution time aspects of the modularity provided by the OSGi framework, providing a management API and a lifecycle for OSGi bundles. Lifecycle operations defined by this layer allow dynamic applications evolution and management by means of changing the composition of bundles and interacting with the OSGi platform through their execution context. Bundles can be dynamically installed, started, updated, stopped and uninstalled to flexibly customize applications. Figure 9 shows a state diagram containing all possible state during the lifetime of a bundle.

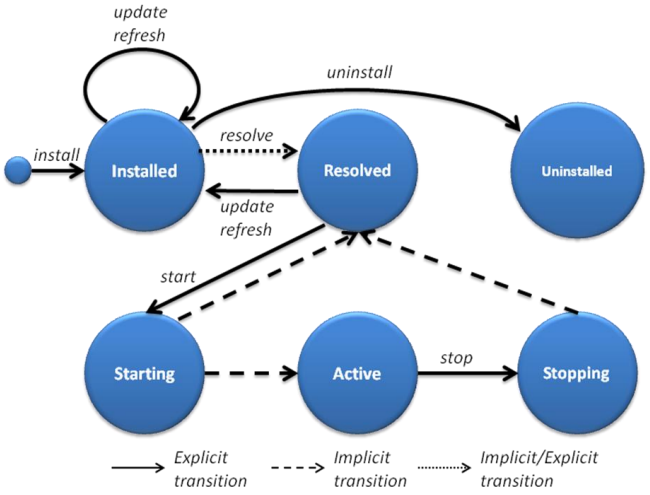

Figure 9. State diagram representation of OSGi bundle lifecycle [OSG05]

First of all, the bundle lifecycle starts with its installation, which is performed through the *install* operation. Installation is performed by passing to the platform the URL of the bundle JAR file. The bundle then is created in the Installed state. Next, the framework must ensure that all the bundle dependencies are satisfied before it can be used. This guarantee is represented by the transition from Installed to Resolved[41]. A bundle is in the Resolved state can be started when executing the *start* command, what leads it to the Starting state. The framework then looks for the Activator class of the bundle informed on its metadata and

[41] This transition is usually implicit and made automatically, but it is also possible to do it explicitly.

executes its `start()` method[42]. If the method executes successfully the bundles transitions to Active state, else it returns to Resolved. An Active bundle can be stopped by means of executing the *stop* command. It transitions then to the Stopping state, where the method `stop()` in the Activator class of the bundle is executed. To the extent that its dependencies were already resolved, the bundle returns to the Resolved state[43]. The framework can be forced to resolve bundle dependencies again by executing the *refresh* or *update* commands. Bundles in the Installed state can be uninstalled by the *uninstall* command, transitioning to the Uninstalled[44,45].

The Module and Lifecycle layers have a very close relationship, in that the Lifecycle layer controls which bundles are installed into the framework, what influences the bundles dependency resolution in the Module layer.

2.4.4 Service Layer

The Service layer builds on top of the Lifecycle and Module layers and defines a model for providing and consuming services as in SOA. Bundles can publish and discover services through the medium of a shared and centralized service catalogue, the *OSGi Service Registry*. This catalogue is accessible through a `BundleContext` object, which is used by OSGi bundles to access the OSGi framework facilities.

In the OSGi specification, services are POJOs with associated Java interfaces (contracts) and meta-information which are published in the OSGi Service Registry. Whenever a bundle needs a service, it would use the `BundleContext` interface to access the service catalogue and ask for a given interface. Filtering parameters may also be provided by the service consumer under the form of LDAP queries to refine the results. In case that the register finds services which match with the interface and the filtering parameters, the registry returns a set of `ServiceReference`s, that is, the information and the indirect reference[46] for

---

[42] This method can be used to provide a bootstrap behavior to a bundle, such as allocating resources and registering its services in the OSGi Service Registry.

[43] Stopped bundles have their services automatically removed from the Service Registry, but the stop method must contain the code to release all resources taken along the bundle's execution.

[44] Since the version 4.2 of the OSGi specification, bundles in the Active state will be automatically stopped, transition to the Resolved state and then to the Installed state, before uninstalling it.

[45] The lifecycle API generates as well synchronous and asynchronous notifications at runtime for bundle and framework events.

[46] Indirect references are used in order to allow service usage track, laziness support and removal notifications.

the corresponding services providers. After, consumers use this reference and the `BundleContext` to bind to the object, which represents the actual service implementation.

On service registration, modification or unregistration, the OSGi framework can send events to notify special objects placed on the service requesters, namely *service listeners* and *service trackers*. Events can be filtered for these objects through LDAP filters. Listeners and trackers in a bundle are automatically removed when the latter stops.

In addition, the OSGi Alliance has specified services which are offered by the platform for common performed tasks. They are divided into *framework services*, which are services that are part or direct the operation of the framework, such as Package Admin, Permission Admin and URL Handler; *System services*, which are necessary functions for every system, such as the Log Service, Event Admin and Component Runtime; *Protocol services*, which map external protocols to OSGi services, like the HTTP service and the UPnP Device Service; and other *miscellaneous services*, such as Wire Admin and XML Parser.

2.4.5 OSGi Applications

Since its first release, many implementations for the OSGi specification have been developed. In 2004, the first open sources projects implementing the OSGi specification arose. Apache Felix, Eclipse Equinox and Knoplerfish are well-known examples. Currently, the OSGi specification is in its fourth release (R4). We list some OSGi implementations in the Table 2.

Implementation	Developer	Certification R4	License
Felix 2.0[47]	Apache	No	Apache License v2.0
Equinox 3.2[48]	Eclipse	Yes	Apache License v2.0
Knoplerfish 2.0[49]	Makewave	Yes	BSD/Commercial
mBedded Server (mBS) 6.0[50]	Prosyst Software	Yes	Eclipse Public License/ Commercial
OSGi R4 Solution[51]	Samsung	Yes	Commercial
SuperJ Engine Framework[52]	HitachiSoft	Yes	Commercial

Table 2. Implementations of the OSGi R4 Specification

[47] http://felix.apache.org

[48] http://www.eclipse.org/equinox

[49] http://www.knopflerfish.org / http://www.makewave.com/site.en/products/knopflerfish_pro_osgi.shtml. Only the commercial version is R4-Certified

[50] http://www.prosyst.com/index.php/de/html/content/97/Products-OSGi-Implementation. Open source version is based on Eclipse Equinox, while commercial version uses Prosyst Framework.

[51] http://www.samsung.com/osgi_patent_pledge/index.htm

[52] http://hitachisoft.jp/products/superj/

In addition, many enterprises have started adopting OSGi technology in their solutions, principally for rich client platform applications. One of the first applications of the OSGi technology in the industry was in the Eclipse IDE[53], under the form of Eclipse plug-ins. OSGi implementations were also incorporated to application servers such as JOnAS[54], JBossAS[55], Oracle/BEA WebLogic Application Servers[56], Oracle/Sun Glassfish v3[57] and IBM Websphere Application Server[58] in order to provide a runtime environment for the application server's modules. IBM has also built Lotus Expeditor[59], a client middleware that enables connection, delivery and management of applications and services, on the OSGi framework. Lotus Expeditor is the foundation for other IBM Lotus applications, such as Sametime and Notes.

Ricoh Company Ltd., a Japanese company ranked among the top worldwide with a 30% percent market share in the United States[60], is another OSGi platform adopter and has included Knoplerfish onto its Embedded Software Architecture device platform[61] for client customization on multi-functional printers. Cisco is another giant company which included ProSyst's mBS as optional add-on on its Application Extension Platform (AXP)[62]. Cisco AXP allows the integration of applications with Cisco's Integrated Services Router (ISR). Other OSGi applications in the industry include SIP communicators[63] and the Service Creation Environment in Alcatel-Lucent's IP Multimedia Subsystem Application Server[64].

2.4.6 Real-Time OSGi

The OSGi Service Platform has been a widely adopted technology for home automation, pervasive environments and even business contexts, due to its dynamic service component model, flexible remote management and its continuous deployment support.

[53] http://www.eclipse.org/osgi/

[54] http://wiki.jonas.ow2.org/xwiki/bin/download/Main/Documentation/JOnAS5_WP.pdf

[55] http://jbossosgi.blogspot.com/2009/06/jboss-osgi-runtime-as-integration.html

[56] http://download.oracle.com/docs/cd/E12524_01/doc.1013/e14481/products.htm

[57] http://docs.sun.com/app/docs/doc/820-7688/abppa?a=view

[58] http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/osgi/index.html

[59] http://www.ibm.com/developerworks/lotus/library/expeditor-osgi-services/

[60] http://www.ricoh-usa.com/about/awards/industryawards.asp

[61] http://www.makewave.com/site.en/showroom/ricoh.shtml

[62] http://www.cisco.com/en/US/prod/collateral/routers/ps9701/data_sheet_c02_459075.html

[63] SIP stands for Session Initiation Protocol, a protocol which is likely to be employed by next generation mobile networks. SIP was originally designed for Voice over IP session management, but it became popular in other applications. A SIP communication may be found in http://www.sip-communicator.org

[64] http://www.alcatel-lucent.com/wps/DocumentStreamerServlet?LMSG_CABINET=Docs_and_Resource_Ctr&LMSG_CONTENT_FILE=Brochures/5400_IMS_Application_Server_Bro.pdf

However, it lacks support for real-time applications, which restricts its application to environments where real-time requirements do not have to be guaranteed. Indeed, the continuous deployment support allows bundles to be installed, started, stopped and uninstalled at anytime, thus the static system configuration assumption is not valid, because the system will evolve during the whole application lifecycle.

The dynamic reconfiguration feature in OSGi is useful in real-time systems for allowing the evolution of real-time systems at run-time and for facilitating the maintenance of software components. Furthermore, it is also useful for managing resources, ensuring that only necessary components are installed in the platform, and minimizing the number of components in order to save memory. Another helpful feature for real-time software deployed in dangerous environments and for mass production control systems is that OSGi allows bundles to be controlled remotely.

Few works have been dedicated to provisioning real-time support in OSGi. [GUI08] presents a descriptive approach for real-time support in the OSGi framework, where the real-time guarantee is implicitly provided by the container runtime environment. In this approach, a real-time contract is specified in the component's metadata. A service called Declarative Real-time Component Executive is responsible for solving the constraints between real-time components at execution time. A hybrid real-time component was used instead of a pure real-time component model to separate the adaptation logic from the real-time component code: while the management parts run in a conventional non-real-time environment, implemented in line with the OSGi specification, an independent concurrent process containing the predictable code[65] runs directly in the real-time operating system layer.

Richardson et al. in [RICHA09] analyzed ways to provide temporal isolation (that is, preventing the timing misbehavior in one thread from affecting the timing constraints of other independent threads) in the OSGi platform at thread and component levels in order to enable the development of component-based RTSJ applications.

Another proposal for real-time OSGi was presented in [COA07]. It suggests the addition of more metadata information to real-time bundles, the isolation of bundles by means of real-time partitioning and a layered architecture for the OSGi Service Platform, with three

[65] The real-time tasks in this approach are written using native code.

distinct profiles models which run atop of the OSGi core: OSGi Enterprise, OSGi Soft Real-Time and OSGi Hard Real-time.

OSGi is already being used in applications for real-time applications, such as in the core of Oracle's (formerly BEA) WebLogic Real-Time[66]. This is a low-latency Java-based middleware framework for event-driven applications which process event streams in real-time. The OSGi framework is the base for BEA's microService Architecture (mSA), an infrastructure based on SOA principles of separation of concern and substitutability. The mSA is event-driven and notification services are used to publish and discover components and microServices [MAH07]. Aonix and ProSyst have been working in the integration of PERC and mBS to establish a reference implementation for prototyping activities in the automotive industry. However, due to the fact that their solutions are proprietary, very little information about is available.

Issues raised by the consideration of real-time requirements in the OSGi Service Platform will be discussed in the next chapter.

2.5 *Summary*

Real-time systems are software systems whose correctness depends on both logical and temporal aspects. The most important properties of real-time systems are their predictability and the determinism of their execution time. In the beginning, dedicated hardware was created for real-time systems; however, with the advances in computer hardware, real-time concerns concentrated on the software layer. Real-time operating systems were designed to offer deterministic hardware access time; real-time scheduling algorithms were elaborated to avoid deadliness missal; and programming languages were designed or extended in order to allow programmers to develop real-time systems with high-level abstractions. Java is a programming technology which was extended in order to provide the timeliness required by real-time systems. In its standard form, Java presents several shortcomings which prevented its use in the design of real-time systems, such as garbage collection, dynamic lazy loading and loose scheduling mechanisms. The Real-Time Specification for Java is an extension to Java which adds real-time programming constructs and constraints to the Java environment.

[66] http://download-llnw.oracle.com/docs/cd/E13221_01/wlrt/docs20/index.html.

Due to its interaction with the real-world, some real-time systems must be dynamically adaptive, that is, they must be capable of being modified and updated at runtime. Software runtime adaptations may be parametric or compositional; the former means modifications in the system variables, while the latter specify addition or removal of system components. Many techniques have been developed to support software runtime adaptation, such as aspect-oriented programming and computational reflection. Service-Oriented Architectures (SOA) and Component-Based Software Engineering (CBSE) are another two paradigms for the development of dynamic adaptive systems which are becoming very popular. The need to separate SOA mechanisms from business or functional code originated the concept of service-oriented component models, a programming model where components are used to implement services. Some works have been developed in the application of dynamic adaptive systems techniques in real-time software. Most of them are concentrated on Real-time SOA or in Real-time CBSE, but very few try to deal with service-oriented component models.

In this study, we focus on the conflicts between real-time requirements and the dynamism provided by the service-oriented component models. These conflicts are discussed in the next chapter. The OSGi service platform, a platform for the dynamic deployment of services, is used as the main object of our study. The OSGi framework was chosen due to its simple component lifecycle and its growing popularity and adoption into large companies' solutions, such as Cisco, Ricoh and IBM.

3 REAL-TIME CONSTRAINTS IN THE OSGI DYNAMIC PLATFORM

Component-based software engineering and service-oriented architectures are becoming widely-adopted effective ways of developing dynamic and flexible software. The emergence of these development paradigms lead to the introduction of SOA concepts into component models and execution environments. In *service-oriented component models*, applications are decomposed into a collection of interacting services, being capable of autonomous runtime adaptation accordingly to the availability of its required services. In addition, the component-oriented programming concepts are used to separate the code responsible for service management mechanisms from the logical implementation of the service functionality. However, it is generally accepted that this dynamicity limits the use of service-oriented component models in applications with real-time requirements, where predictable behavior is a fundamental issue. The fact that services may become available or unavailable at any time during the execution of an application may imply in unbounded execution times and unforeseen delays.

In this chapter, we discuss the issues generated by the consideration of real-time requirements in service-oriented component models and instantiate those problems in the OSGi framework. Section 3.1 presents the conflicts between real-time requirements and the need for dynamic adaptation in service-oriented applications. In section 3.2, we discuss about the lack of real-time support in the OSGi framework. To conclude the chapter, a scenario is presented to exemplify the problem of dealing with dynamic architectures in real-time applications in section 3.3.

3.1 *Real-Time Issues in Dynamic Service-Oriented Component Models*

Service-oriented component models were originated from the need of explicit support for dynamic availability into component models [CER04b]. This property, however, is on the basis of the main shortcomings for using service-oriented component models in real-time applications. These runtime changes may influence other real-time components and compromise the whole application determinism due to the architectural-level modifications. Furthermore, it requires component instances to monitor context changes and listen to component arrivals or departures, which imposes an extra burden for the real-time system developers and additional overhead for the overall system. Departing services may affect availability of hard and critical real-time applications. Dynamic availability also influences WCET analysis due to the fact that runtime updates of service implementations means a change in the WCET of all threads that require this service. On top of that, resource reservation, commonly performed in real-time systems to ensure timing requirements, may cause overload situations because we cannot predict the number of components present in the system[67].

Moreover, most of service-oriented component models do not provide temporal isolation of components. Dynamic availability may lead the service framework to states where there are more installed components than is possible to guarantee resources for. Consequently, threads can miss their deadlines and, without temporal isolation, they may affect timing requirements of other independent threads and components. In addition, this lack of temporal isolation may be a source to carry out a Denial-of-Service attack [NEE93] on OSGi, depleting system resources and preventing other components from obtaining their guarantees.

Another important issue when considering real-time requirements in service-oriented component models is the lack of a global view of the real-time context. Actually, this problem originates from the fact that in the component-based software engineering approach, components are developed independently of the system and other components. Thus, without a global knowledge of the system, it is hard to guarantee timeliness requirements for each component. Global context knowledge is particularly important for priority assignment. By

[67] Consequently, resource usage cannot be predicted.

means of scheduling analysis we can correctly assign priorities within components, but not across components. This can cause problems like deadlocks, starvation and missed deadlines, aside from possibly incorrect results due to the violation of the precedence among computations.

When considering specific service-oriented platforms, we may find many other issues for real-time systems. Next section discuss about the lack of real-time support in the OSGi Service Platform.

3.2 *Real-Time Issues in the OSGi Platform*

Besides the problems listed in the precedent section, the OSGi platform itself presents issues which may compromise the predictability required by real-time applications.

First of all, the OSGi service platform was not conceived as a real-time application. So far, all of its implementations were written in standard Java and its classes will need to interact with real-time components, possibly written in the RTSJ or another real-time Java technology. Considering RTSJ specifically, this may lead to potential memory assignment issues, due to its rules which prevent dangling memory references, i.e. objects from one scoped memory area being referenced in another memory area. Furthermore, the fact that RTSJ class objects are stored automatically in the immortal memory and that the developer may also explicitly allocate objects into it may complicate class unloading and generate memory leaks.

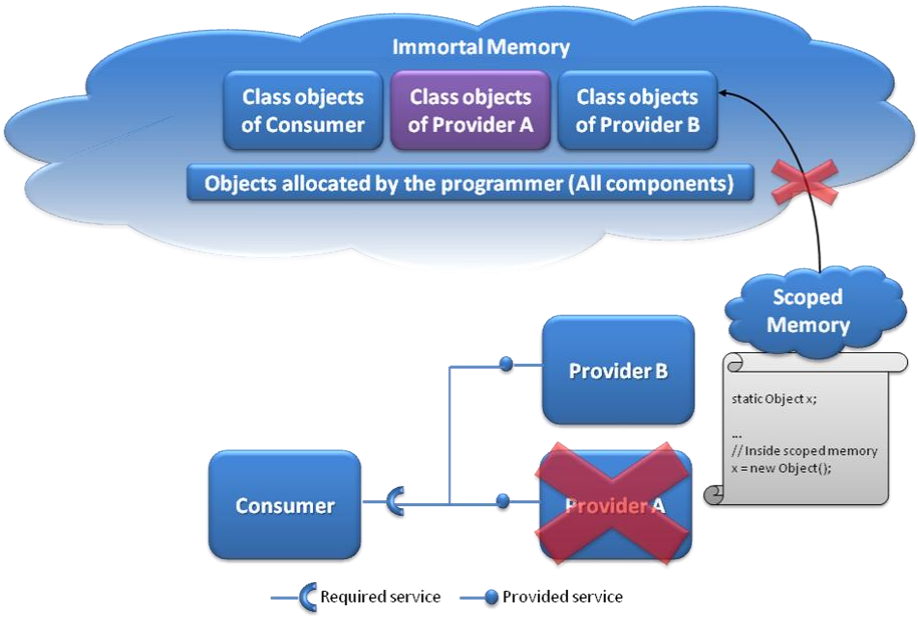

Figure 10. OSGi and RTSJ memory areas

Figure 10 shows the illegal memory assignment and memory leak problems. As said before, in RTSJ class objects are allocated in immortal memory, jointly with objects explicitly allocated by the programmer. In the OSGi specification, each component has its classloader. Class objects hold a reference to its class loader object, and class instances hold reference to its class object. The memory used by a classloader then is reclaimed when it is no longer referenced by a class object. However, even if "Provider A" component is removed, its class objects remain because immortal memory in RTSJ never is garbage-collected, leading to a memory leak in the OSGi framework. Moreover, "Provider B" has a method which executes inside a scoped memory and stores a new object in a static field called *x*. In RTSJ this leads to a memory assignment error, because static fields are also stored in immortal memory and immortal memory cannot hold references to scoped memory areas.

The OSGi framework is based in standard Java. Consequently, it uses ordinary Java threads. RTSJ components have real-time priorities, which are higher than the ordinary ones. Thus, due to the fact that RTSJ's scheduler uses a run-to-block scheduling policy, real-time component threads may lockout system threads and keep the administrator from issuing commands to the framework. In addition, the framework must ensure the safe termination of threads from components which have been uninstalled, currently a developer's concern.

In general, real-time applications have complex and application specific requirements. However, the OSGi specification mechanisms for the composition of modules are strongly based on the import and export of Java packages resolved by an LDAP filter[68,69]. This coupling with Java language prevents us from specifying more complex relationships among real-time components. With regard to resource management, there is no specific mechanism for global resource management. Thus, due to the fact that resources are globally shared, real-time developers must find a way to provide global resource budget enforcement for components and ensure component real-time contracts.

In this study we will focus mainly on ways to deal with real-time constraints in service-oriented component frameworks such as the OSGi service platform where dynamic availability is a present property.

---

[68] Lightweight Directory Access Protocol, an application layer protocol for querying and modifying data over TCP/IP by means of systems which store, organize and access information in a directory.

[69] The OSGi R4 specification introduced the concept of *declarative services* to support dynamic composition at service level; however it is still tightly coupled with the Java language [GUI08].

3.3 *Scenario: Video monitoring application*

An example of application with soft real-time requirements and which has dynamic adaptations on its architecture is a motion detection monitoring system. In this application, the motion detection system is connected to several cameras which provide an image frame service. The number of cameras to which the motion system is connected is unknown at design time; at runtime, once a camera component is installed to the system, it is automatically connected to the motion detection module, which will process its frames in order to detect human presence. Indeed, cameras in this system have the dynamic availability property, being able to appear and disappear at any time.

An important concern in this system is image processing time. If we assume that frames are sent regularly to the motion detection module, image processing time must be bounded in order to allow the system to react as soon as possible to a human presence. For instance, if we assume that image processing takes 200 ms and cameras send an image frame each 1s, we are only able to process four other frames before receiving a new round of image frames to process. Excessive retard of their processing could create security issues, such as the delayed detection of a thief, hours after his attack. In addition, we must consider that the motion detection module thread is a task which is periodically scheduled to run, besides other threads which might be running in the system, such as framework system threads.

Figure 11 illustrates the architecture of the system described above. Four points must be observed concerning the impact of dynamic availability in the architecture of the real-time system:

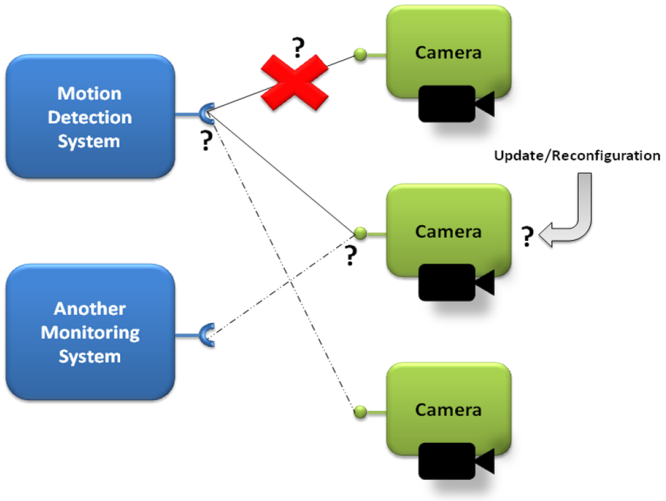

Figure 11. Dynamic availability in the motion detection system

- **Binding to a new camera component:** The system underlying mechanisms which perform the binding between components may carry an unpredictable delay in the image processing time. Moreover, depending on the number of cameras to which the motion detection system is connected, processing time may become longer than the time attributed to the execution of the motion detection system;
- **Removal of a camera component:** In the same way that binding mechanisms, unbinding mechanisms may introduce an unpredictable delay in image processing time. Furthermore, we must ensure that the removed camera component is not currently being used by the motion detection system.
- **Update/reconfiguration of a camera component:** Updating a camera component in the OSGi platform makes the component stop and return to the Installed state. Then, component dependences are resolved and afterwards the component is restarted. The duration of this process cannot be predicted. In addition, the new camera component may have different properties. Thereby, feasibility analysis which may have been performed before are not valid anymore and therefore must be redone.
- **Binding a camera component to another component:** Another factor that must be considered is the case where the camera component provides the image frame service to more than one consumer. Sending the frame to one of the components while the other waits may introduce an unpredictable wait time to the former component.

To summarize, besides the real-time issues inherent to the Java Platform, the OSGi Service Platform present several other shortcomings which make it inappropriate for deploying and developing real-time applications. These shortcomings may have two different reasons. The first one is the fact that the OSGi platform was not conceived for being used in the context of real-time applications. The other one comes from the dynamism provided by its service-oriented component model. Indeed, component dynamic availability feature allows components to appear and disappear unexpectedly at runtime. In this study, we will focus on the dynamism aspect.

In the next chapter, we present some propositions which take into account the issues listed in this chapter. Our scenario will be revisited in chapter five, for the validation of the propositions.

4 PROPOSITION

In the last chapter, we presented the issues raised by the consideration of real-time requirements in service-oriented component models and in the OSGi framework. The aim of this chapter is to propose solutions to these problems. Some of the solutions proposed target service-oriented component models in general; others are more specific to the OSGi platform, but they can be generalized and adapted to other dynamic service platforms. This distinction comes from the fact that conflicts identified in chapter three come from different levels; while the architecture-related issues may be treated in a higher-level of abstraction, problems concerning component lifecycle, component model and the platform-specific questions are more likely to be treated in the OSGi framework level.

We propose in this work an architecture freezing policy when the platform is in a real-time processing state, holding all reconfigurations until the end of execution of the critical code. Then, modifications may be performed if they respect an agreement established between the service consumer and the service provider. Section 4.1 introduces the architectural freezing approach. Section 4.2 presents the extension of a Service Level Agreement (SLA) model in order to take account real-time requirements. Finally, the implementation of those approaches in the OSGi Service Platform and some other modifications in order to make it real-time aware are suggested in the section 4.3.

4.1 *Architectural Freezing*

Since one of the main problems that arises when considering real-time requirements in dynamic platforms is due to dynamic availability, that is, the capability of adding, updating and removing a component, a solution that comes naturally is to forbid those interactions at execution time. This is a radical solution which compromises the dynamicity of the applications and disables runtime software evolution, albeit the most frequently used option

currently. An ideal solution should be less strict, allowing applications to be modified at run-time without interfering with its deterministic behavior.

We consider that in a real-time application every component is able to perform its task within the real-time requirements of the application. Thus, the issues lie in the bindings between components and how they change across time. Suppose that an application is represented by a set of states. Each state corresponds to a given architecture of the application, and transitions between states correspond to the arrival, the departure or the update of a component in the application during runtime. In consequence, in order to respect the application timing constraints, we must define rules for the transitions between states.

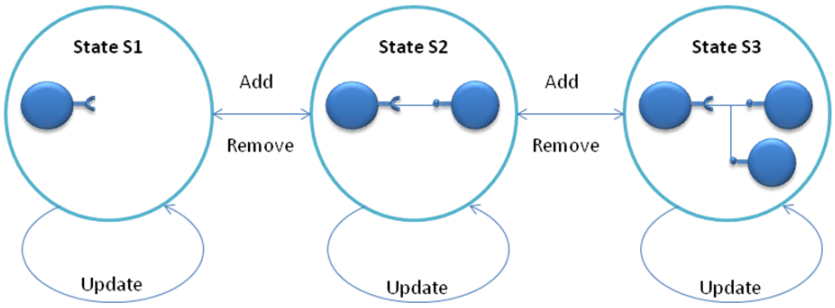

Figure 12. Machine state representation of system architectures

Figure 12 shows a machine state representation of a system's architecture. A system in the state S1, in the presence of a component which satisfies the component's dependencies transits to the state S2. In the OSGi platform, for example, this transition is performed automatically. If this component is removed, the system returns to the state S1. We assume that the service registry in this service platform is unique and centralized, so one architecture modification is performed at a time. We present in figure 13 a modification to this machine state representation, adding *real-time states*. Once a system enters a real-time state, no modifications are performed until it returns to the corresponding non-real-time state, in order to ensure that real-time requirements will be met. We call this approach *architecture freezing*, due to the fact that the system holds all the architecture changes until the system quits the critical code area. This solution addresses mainly systems whose most part of the code is non-real-time but with some critical pieces of code which are executed sporadically.

In platforms where binding and unbinding management is executed by only one thread but several applications may run concurrently, blocking this manager may cause other real-time applications to violate their timing constraints.

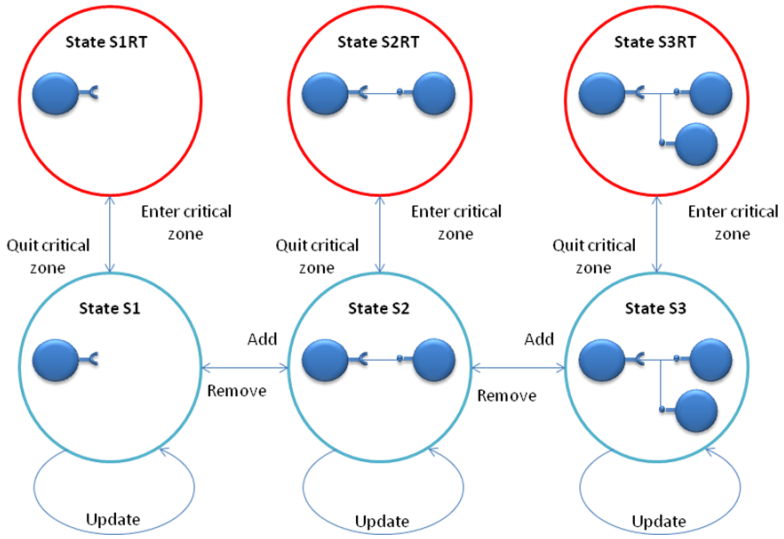

Figure 13. Machine state representation of a system architecture with architectural freezing

In figure 14 we show two pseudocodes for implementing architectural freezing in a platform. Figure 14a shows the application side of the architectural freezing, where the developer specifies that the platform must enter a real-time state in order to ensure that components' dynamic behavior will not introduce unpredictability into the real-time code that will be executed. Variations of this code could include the duration of the real-time state instead of calling a method to quit the real-time state. In this case, when two components enter a real-time state at the same time, and component C_a wants to freeze the system architecture during t_a units of time and component C_b needs to freeze the system architecture during t_b units of time, the system must keep the architecture frozen during $max(t_a, t_b)$ in order to ensure the real-time behavior of both components, where *max(a, b)* is *a*, if $a \geq b$, or *b* otherwise. This variation is possible due to the fact that in a real-time state, all the code executed is bounded in time. Real-time and non-real-time may designate respectively real-time and non real-time bundles (deployment units) instead of simple methods or lines of code.

Figure 14b shows the platform side, more specifically a method which is responsible for the architecture modification. Structure may differ among applications, so that the code responsible for effectuating modifications in the architecture may be distributed among different classes. But in all cases, the platform must verify whether it is in a real-time state or not before starting to perform modifications in the architecture of applications which are being executed.

```
a)  public void toto() {
        … // non-real-time code
        Enter-real-time-state();
        // Platform in real-time state
        … // real-time code
        Leave-real-time-state();
       // Platform in normal state
    }
```

```
b)   public void method-to-modify-arch(){
       if (arrival || departure || update) {
         while platform.isOnRTState() {
           // wait until RTState is over
         }
       // modify architecture
       }
     }
```

Figure 14. a) Pseudocode for freezing a system architecture b) Pseudocode responsible for freezing the system architecture in a platform

Architectural freezing can be extended to a system of scheduling and reservation of architectural modifications in the system. For instance, components may specify at which time the architecture must be frozen, being up to the system to make a schedule analysis and find an available time slot for performing architecture modifications.

Next section details the extension of a Service Level Agreement model in order to take into account real-time requirements. This agreement will be used by a manager which will verify before each architecture reconfiguration if component addition, removal or update do not violate the timing constraints of the applications hosted by the OSGi platform.

4.2 *Real-time Dynamic Service Level Agreement*

Service Level Agreement (SLA) [VER99] is a negotiated part of the contract established between the service provider and the service consumer which formally defines the level of service and the penalties applied when commitments are not met by either party. These commitments are specified in order to reach a given quality of service. A SLA contains information such as the parts engaged in the agreement, the service provided, service utilization time, service availability, service reliability, service utilization price and dates for renegotiating the agreement. SLAs are monitored by *Service Level Management* (SLM) modules, which are also responsible for applying the penalty policies in case of non commitment of the agreement. In order to avoid equity issues, generally a third part (the service certifier) chosen by the service provider and consumer is present to take measures periodically in order to verify if the contract clauses are violated.

An extension to SLAs in order to handle dynamic availability and service disruption is proposed by [TOU08] and [TOU10], the *D-SLA*. Service disruptions are characterized by three additional metadata: *maximum service disruption time*, *maximum accumulated service disruption on a sliding time-window* and *time between two service disruptions*. By using past activities and recorded histories of service provider contracts, we can also consider service providers which are not present in the platform at selection time. We propose in the next

subsections an extension to D-SLA in order to take into account real-time requirements, RTD-SLA.

4.2.1 RTD-SLA Content

Besides the content described by [TOU08] and [TOU10], a Real-time Dynamic Service Level Agreement contains the following data:

- **Task type:** The task type specifies if a given module has periodic, aperiodic or sporadic behavior. This information is important for scheduling analysis performed by the Service Level Management module before selecting or not a service. For instance, depending on how critical the tasks which are being executed are, the manager may block the admission of aperiodic or sporadic tasks.
- **Period time**: In the context of periodic tasks, service provider and service consumer period times are important information for scheduling analysis. For example, the manager may detect that it is not possible for a consumer to use a service because their periods do not match.
- **Worst Case Execution time**: WCET is also important information for scheduling analysis. The manager is able to predict if a consumer can execute within its period time and if service invocations will not interfere on its timing constraints. If that is the case, the manager may act according to the policies specified in the RTD-SLA.
- **Resource Usage**: Resource usage information may stop components from blocking due to resource waiting times. Thus, for common resources such as CPU and RAM, components must specify their usage. In addition the sum of the usage of the services which are being consumed should not exceed the budget allowed for the service consumer to execute.
- **Priority**: In applications where task precedence is important to ensure proper system behavior, it might be interesting for the manager to consider the priority of each task when admitting new tasks in the system. This parameter may also be used to choose between the admission of two components, where only one is possible.
- **Policy**: In case of violation of the contract, policies specify what the modules which perform the service level management must do. These policies may go from a simple substitution of the service to a decrease in the reliability rate in presence of trust

indicators. However, all possible actions must have their execution time bounded in time.

Figure 15 presents a UML diagram representing the structure of our RTD-SLA. For the sake of simplicity, parameters related to D-SLA (maximum service disruption time, maximum accumulated service disruption on a sliding time-window and time between two service disruptions) are not represented in the class diagram.

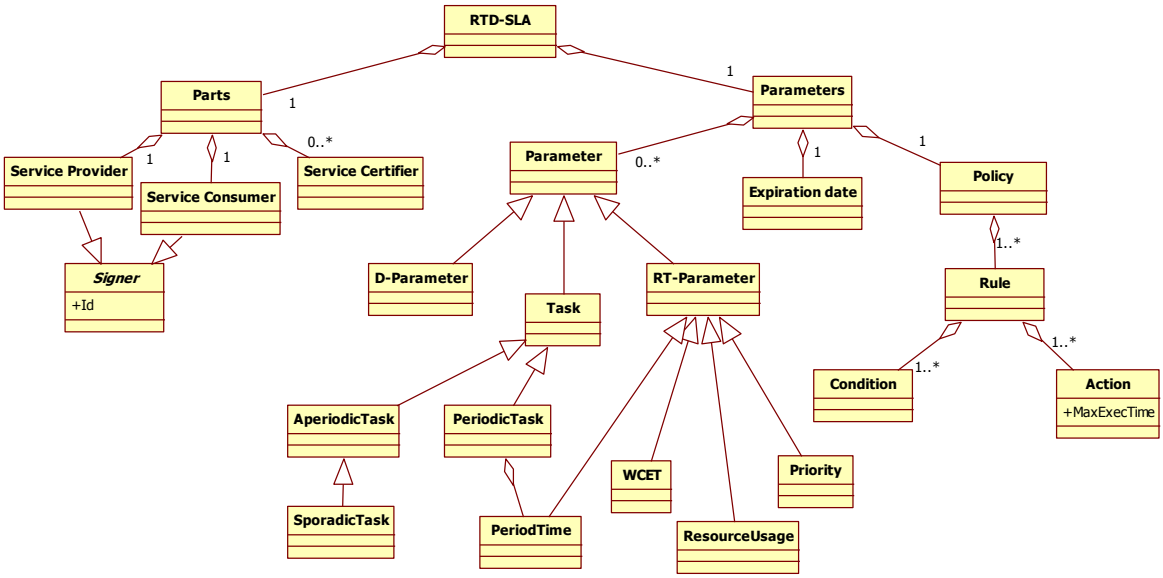

Figure 15. UML diagram of RTD-SLA

4.2.2 Service Level Management

Every dynamic arrival and departure produces changes in the service registry state, the module responsible for the service level management must have listeners to these events. We may have one or several modules which listen to events corresponding to one specific service, all the services or a specific subset of services. With D-SLA, in case of service disruption, the service usage is suspended until the return of the service provider. If the disruption time exceeds the maximum service disruption time specified in the SLA or if the overall downtime goes beyond the maximum accumulated disruption time, actions are taken according to the policy specified in the agreement.

The SLM also must act in the service selection in order to allow the consumer to only bind to providers which will not interfere with its deterministic behavior. In addition, the SLM is responsible for maintaining a global context of the system and using this information to perform schedulability analysis and ensure predictability to all components in the system.

Actually, when the application is in a non-real-time state, the SLM will control all the platform reconfigurations.

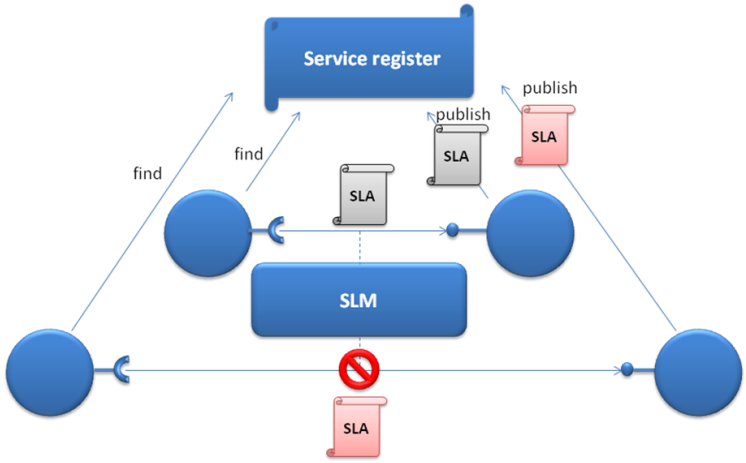

Figure 16. Real-time SLA and SLM

Figure 16 exemplifies the intervention of the SLM on the binding of a consumer to a provider. Even though the provider has the needed functional properties, its QoS properties do not match with the real-time requirements of the system. Thus, the SLM module blocks the interaction between the consumer and the provider. Depending on the policy adopted by the SLM, it may hold the binding until the system decreases its charge or simply reject it.

The fact that we have a central manager may create a bottleneck in the application. In order to bypass this shortcoming, platforms may have multiple SLM modules and use known algorithms in the distributed systems domain in order to keep a coherent global state of the system.

Solutions presented in the sections 4.1 and 4.2 are both platform-independent. However, in the real-time domain often solutions are optimized to a given platform in order to reach a more deterministic execution time. Next section discusses about modifications specifically for the OSGi Platform.

4.3 *Real-time Aware OSGi Platform*

Although the OSGi Service Platform was meant to be used primarily for embedded systems, it does not have support for real-time applications. Real-time requirements cannot be expressed in the platform and its classes are written in standard Java and interact with Java's

standard API. We list in this section some approaches which are possible in order to create a real-time aware OSGi Service Platform.

4.3.1 OSGi Real-time Core and Code Instrumentation

One of the problems cited in the last chapter was the fact that OSGi was not written with real-time code. Therefore, its underlying mechanisms may introduce unpredictable behavior to real-time applications, if not errors due to the interaction between real-time and non-real-time code. Ideally, the OSGi specification should be revisited in order to create a real-time version of its core, where all the operations executed automatically and implicitly by the framework are bounded in time. Performing changes in the specification level would allow groups to freely create compatible implementations.

However, the core of the last OSGi specification has around 300 pages. Reviewing it could take a long time. For this reason, one possible approach is to create a real-time aware version of the OSGi framework by means of code instrumentation. Instrumentation refers to a computer science technique in which code is added to a program in order to gather data to be used by measurement and monitoring tools. Java 5 provides services to instrument Java code, adding byte-code to methods and classes. This addition is performed by an instrumentation library that may be written in Java, which is connected to an application and runs embedded in the virtual machine, intercepting the class loading process. Thus, in the context of OSGi, we may replace standard Java code by RTSJ-compliant code [AME09, AME10].

This approach has the advantage that it can be performed transparently to the OSGi framework for any of the implementations of the service platform. Furthermore, since bundles may be installed at runtime, a dynamic approach is necessary in order to ensure that all the bundles in the platform which may be used by the real-time application will also exhibit a real-time behavior. However, the instrumentation process is not real-time and may exhibit unpredictable behavior, besides the overhead incurred by the code interception and modification. Moreover, it requires a revision of the whole OSGi specification in order to define the precedence and priority of each task for the framework.

4.3.2 Real-time SLA in OSGi

A bundle's manifest contains metadata for the management of bundles in the OSGi framework. A way of inserting the RTD-SLA concepts in OSGi is through the extension of this metadata and the implementation of a module for the management of real-time contracts. However, the OSGi framework deals with bundles, which are deployment units, which do not

correspond precisely to components. For instance, many bundles may be used to compose a service oriented component or bundles may be shared among different components, as shown in figure 15.

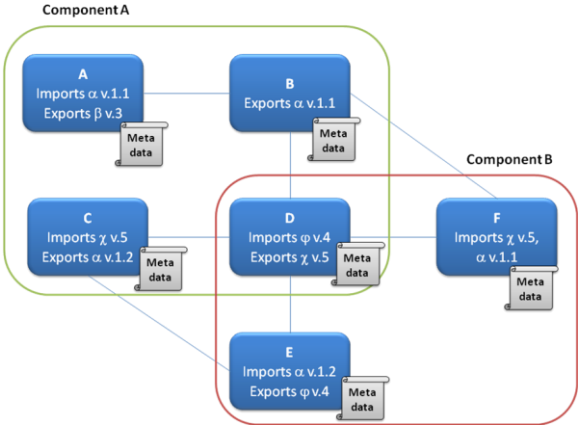

Figure 17. Component vs. Bundle metadata

In the latest release of the OSGi specification, the Declarative Services specification is responsible for allowing developers to specify dependencies between components under the form of required and provided services in an XML-based metadata file. This XML schema is used by a container called Service Component Runtime (SCR), which resides inside bundles and uses OSGi's service requests and service listeners to automatically resolve dependencies and manage component binding, unbinding and service registration. By means of Declarative Services, we may specify the real-time metadata for a component. An alternative to Declarative Services is the use of component models over OSGi, such as iPOJO [ESC07], SCA [BIE07] and Spring[70]. Many developers use component models in order to ease the development and management of components. Generally, their usage requires specifying metadata for components. In iPOJO specifically, we may define handlers to interact with the OSGi platform and enable management of real-time SLAs.

4.3.3 Architectural Lock in OSGi

We can apply the concepts of architectural freezing to the OSGi platform by suspending the automatic binding of components. By means of Declarative Services, component's metadata can be used to specify methods to be implicitly called by the platform when it binds a new component or unbinds an old one. A possible approach is to intercept

[70] http://www.springsource.org/osgi

these method invocations and hold their execution while the application is in a real-time state. Several techniques may be used to perform these modifications:

- **Static byte-code instrumentation:** Since we know which methods will be called for binding/unbinding components, we may use the Java 5's instrumentation package to perform offline instrumentation, loading the class-files from the disk, modifying their byte-codes and saving a new version of the corresponding byte-code by means of tools like ASM [BRUNE02a] and BCEL [DAH99], which use the Visitor design pattern to perform byte-code manipulation. We can put the thread responsible for executing the bind/unbind method to sleep until the end of execution of the critical code. We may ensure that the thread will wake up again because a real-time critical code will always have its execution bounded in time.
- **Aspect-oriented instrumentation**: An alternative to static instrumentation is to use aspects for the byte-code injection. It offers the advantage that code does not have to be written at byte-code level. However, it is less flexible due to the fact that we must specify the correct pointcut to insert the code to suspend the thread before the bind/unbind is performed, while at byte-code level we can insert it anywhere.
- **Modification of component models:** Components models often have access to the bind/unbind mechanisms. Thus, instead of modifying the application code, we may perform these modifications directly on the component model code, making the architectural freezing transparent to the application.

Based on the ideas presented in this chapter, we created a prototype which implements architectural freezing for the OSGi platform. Next chapter presents our implementation with more details.

5 IMPLEMENTATION

In the last chapter we presented approaches for considering real-time requirements in service-oriented component models and in the OSGi Service Platform.

This chapter elucidates some of the solutions proposed by means of an architectural freezing iPOJO-based implementation. Section 5.1 presents the reasons that led us to choose an architectural freezing-based implementation instead of the others in order to validate our proposition. In section 5.2, we discuss the platform and the environment used as base to the implementation. Finally, details of the implementation are given in the section 5.3.

5.1 *Choice of the Approach*

As said in the last chapter, freezing the architecture during real-time states is an approach which may be adapted to any service-oriented platform and component model. This approach does not address platform-specific problems and will not solve all of the issues specified in chapter three. However, it addresses the unpredictability introduced by dynamic availability features, which is a recurrent problem in adaptive dynamic platforms. Thus, validating an implementation which uses this approach would produce a more generic result which may be valid for all dynamic platforms, instead of a specific one.

At the same time, architectural freezing is an approach implementation that is quite simple: we must identify the methods which are responsible for binding, unbinding and modifying components and block them, by means of suspending the execution thread, when the application is in a real-time state. We might have problems if this thread is unique for all applications, as one real-time application may block other real-time applications. Although this is not the case in OSGi, which is inherently multi threaded. The implementation and representation of a real-time state is application-dependent.

Architectural freezing may be mixed with Real-time Dynamic SLAs in order to provide a more complete solution. Only using architectural freezing avoids interference with dynamism issues, but the platform may add components to an application which may degrade the real-time behavior of the latter. Since the main reason to use service-oriented component models is to add dynamism to an application, we chose to focus our implementation on this aspect and not on policy admission schemes.

5.2 *Choice of the platform*

Even though the OSGi Platform enhances the modularization of applications in Java, some functional and non-functional aspects are still mixed in the application logic. The use of service-oriented component models over the OSGi platform helps to separate non-functional aspects such as the dynamism management from the business code in the application. In addition, the OSGi specification has several implementations. Even though they all follow the same specification, it would require platform-specific modifications in order to test our solution. Once again, service-oriented component models add an abstraction layer, and may be executed without any modification in different OSGi framework implementations.

Among the several service-oriented component models for the OSGi framework, we chose iPOJO as the base of our implementation. The iPOJO framework [ESC07] manages dynamic bindings automatically by means of dependency injection and injects code to deal with non-functional aspects. It also provides an extensible container which manages all service-oriented computing aspects, so that the developer may focus on the application logic and the configuration of the container. The possibility of using handlers (i.e., container extensions in iPOJO) to extend the service component model increases the flexibility of the chosen solution. Thus, due to these capabilities and simplicity of development, we decided to use iPOJO as base platform for implementing a prototype.

5.3 *Prototype Implementation*

The architecture freezing approach was implemented by means of an iPOJO Dependency Handler, a type of handler which provides data on service dependencies. These handlers may intercept modifications in the application's lifecycle. The handler is connected to a Real-time State Manager and verifies the platform state before executing a modification on its architecture. If the platform is in a real-time state, the thread responsible for changing

the application's architecture is blocked until the platform reaches a non real-time state. Figure 18 shows the architecture of our solution.

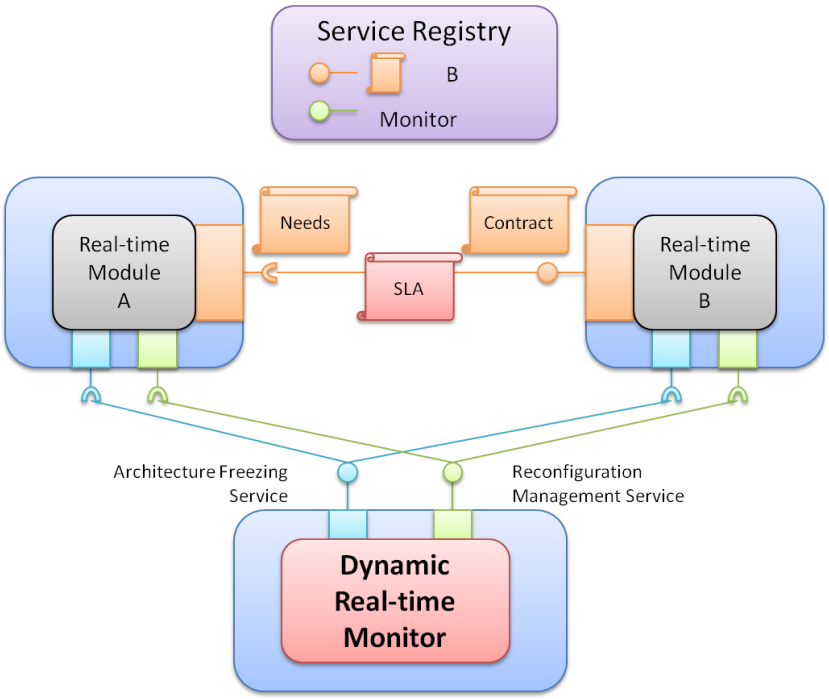

Figure 18. Real-Time State Manager and Solution Architecture

All the iPOJO components in the platform have Dependency Handlers connected to the Real-time Manager, even those which are not real-time. This way, we have a global context of the platform and no component may interfere with the architecture, since the handlers verify the platform with the Real-time Manager before adding, modifying or removing a service dependency. In addition, real-time components are directly connected to the Real-time Manager in order to switch the platform state between real-time and non real-time.

5.4 *Validation*

In order to validate our implementation, we developed a test application based on the scenario described in the section 3.3. This application validates our approach in the context of soft real-time applications. In our scenario, a `WebcamProducer` module an image frame each time its `getImageFrame()` method is called. A `MotionDetection` module is bond to the `WebcamProducer` file and perform the `getImageFrame()` calls every 1 seconds, processing the video stream afterwards. The `WebcamConsumer` is also attached to the `RTManager` module, which plays the role of Real-time Manager and puts the platform in a real-time state during the `getImageFrame()` method call. This way, we may ensure that webcam snap

capture and image transfer are executed in a bounded time. Furthermore, we also ensure that the service will not be removed from the platform during its usage.

In order to perform the validation tests, we installed the Real-time Manager in an *Apache Felix 2.0.5* environment. Then, we installed the *iPOJO 1.6.0* core with the real-time aware Dependency handler that we implemented. The binding between the iPOJO core and our handler was performed by a small modification in iPOJO's metadata. Next, we installed a bundle containing the *Java Media Framework API 2.2.1*, which was used to get access to hardware media devices in Java. A `WebcamProducer` bundle was then launched, followed by the installation of the `MotionDetection` bundle. At the validation of the `MotionDetection` instance, it starts calling `WebcamProducer`'s `getImageFrame()` method. The first time this method is called, it configures the device for capturing video snaps. When we stop the `MotionDetection` instance, iPOJO calls the `WebcamProducer`'s `stopData()` method, which is responsible for deallocating all the taken resources (notably the media devices).

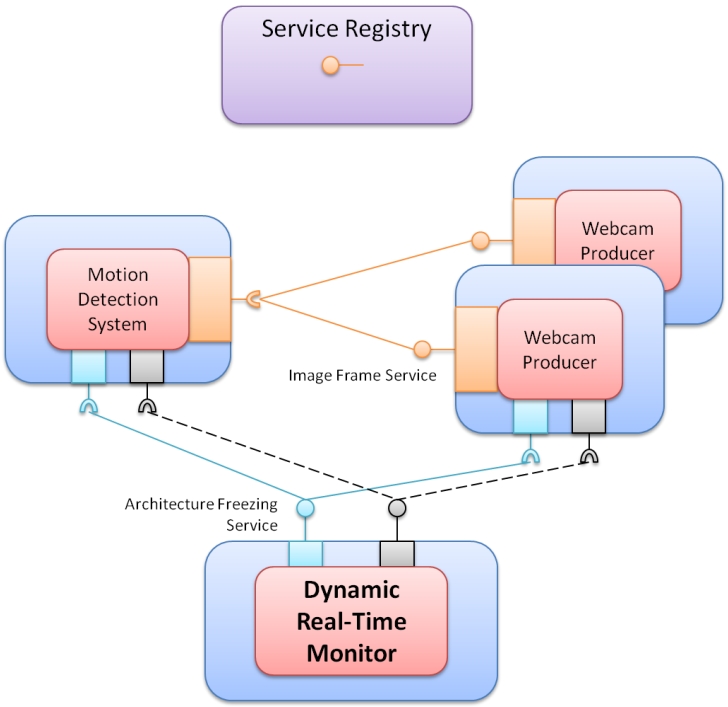

Figure 19. Architecture of the validation test

As we expected, our prototype retains system architecture while the platform is in a real-time state, avoiding component bind, unbind and update. Thus, although we have not taken any metrics yet, we feel that we have validated our approach.

6 CONCLUSIONS AND PERSPECTIVES

Real-time applications must present a predictable behavior, with deterministic response times. This predictability is generally ensured at compile time, by means of schedulability analysis, formal verification and statistics based on the behavior of the several modules which may compose the application. Consequently, most real-time applications are not able to adapt their architecture to unforeseen environment modifications at run-time, hence they are static applications. Conversely, dynamic adaptive software requires flexibility capabilities in order to take into account possible changes on its environment, sometimes without powering off the whole system. Although the approaches for providing runtime adaptation compensate the inconveniences of software systems downtime, including, updating or removing software components at runtime, this may occasionally compromise application safety due to unanticipated effects. Indeed, in general, runtime adaptive software and real-time software are disjoint sets due to the conflict between predictability and flexibility.

However, two factors motivate us to find solutions to the growing number of systems which may be in the intersection of both application classes: first, the fact that even critical real-time software, which cannot have its execution interrupted, must be updated due to environment changes or maintenance; and secondly, the increasing popularity of service-oriented and component-based approaches, which leads industries and developers to migrate their applications to service and component frameworks. An example is the inclusion of the OSGi framework in the core of several application servers, such as JOnAS and BEA's WebLogic Real-Time.

This study concentrated on:

- Identifying the issues raised by software architecture dynamic modifications in real-time applications in Java;

- Suggesting approaches for avoiding the introduction of unpredictability in real-time software hosted on the OSGi Platform.

A prototype implementing an architectural freezing approach for the OSGi platform was provided by means of an iPOJO Dependency Handler, which blocked the addition, update and removal of components when the platform was in a real-time state. This chapter concludes our study by drawing the conclusions obtained, and presenting the perspectives for future work.

6.1 *Contribution*

In this work, we proposed solutions at different levels for dealing with dynamic availability and adaptation of software components. First, we have proposed the introduction of real-time states and the concept of architectural freezing, which locks the system architecture in these states. In this approach, before executing real-time code, the platform enters a real-time state by explicit calls to a real-time manager, which holds every component addition, update or removal until the platform returns to a non real-time state. This way, no modifications are performed in the system architecture and the deterministic behavior of real-time code is ensured. One natural limitation of this approach is that when dealing with physical devices or over the network services, keeping their references when they have already left or are unavailable, may lead to application errors and inconsistent states. This could be solved by extending our model to consider these un-suspendable components and require their declaration beforehand. In this report we have also suggested modifications to the OSGi platform in order to support real-time application requirements.

We also proposed that when the application is not in a real-time state, reconfigurations may be performed if they respect an agreement between the service consumer and the service provider, in which the deterministic behavior of the applications hosted in the platform is maintained. This agreement is an extension of a service level agreement model which takes into account dynamic service-oriented architecture concepts in order to integrate real-time requirements. This SLA model, called RTD-SLA, jointly with service level management modules, provides a solution to avoid the introduction of components which may interfere in the deterministic behavior of real-time applications. However, although this solution limits the admission of unsafe components, it does not avoid the misbehavior of already admitted components, only applying the penalty policies after the constraints have been violated.

An implementation of the architectural freezing approach was developed by means of iPOJO handlers. We chose iPOJO handlers as the base of our implementation in order to increase the portability of our solution and avoid modifications to the standardized OSGi platform and iPOJO core's corresponding source code. The prototype worked as expected: when components entered the real-time state, the real-time manager component held all the dynamic component modifications, performing them once the execution of real-time code had terminated. We see our proposed solutions and our prototype as a first step towards the development of real-time extensions for OSGi's component model and other real-time service-oriented component models.

6.2 *Future Work*

In future works, we intend to clarify some aspects which were not treated in this study, notably:

- **Integration of RTD-SLA in the prototype:** In order to provide a more complete solution, we plan to integrate the RTD-SLA model to our prototype and extend the Real-time Manager module to execute the Service Level Management functions. It is important to test the impact of both solutions in the application's execution time, since monitoring all the components binding may introduce an overhead to the platform.
- **Proposition of a real-time aware core for the OSGi Service Platform:** It would be important to correctly adapt the OSGi platform in order to support real-time requirements. Real-time applications demand a well thought application design, modularization and priority assignment, especially when dealing with RTSJ, where a wrong priority assignment may block the whole platform. Each component of the OSGi core must be taken into account and refactored.
- **Validation of the approaches with different component models and environments:** The first prototype was developed by means of iPOJO Handlers. Thus, though all components bond to the real-time manager may put the application in a real-time state, only iPOJO components arrival, departure and update are considered and only their threads can be blocked by the manager. In order to better evaluate our approach, we plan to test it with other component models and their impact in real-time environments.

- **APIs for measuring resource usage in Java:** Resource usage monitoring is a central factor in real-time applications, since one of the most used techniques for ensuring predictable behavior of real-time applications is the anticipated resource reservation. Our proposed RTD-SLA also uses resource usage as one of the criteria for the admission of real-time tasks. Thus, resource consumption monitoring is a task to be performed by the SLM modules. However, standard Java API does not provide functions for obtaining resource usage at runtime [GUID02, JSR284]. We intend in future works to provide solutions which take into account this issue.

7 REFERENCES

[AKK07] Faisal Akkawi, Atef Bader, Daryl Fletcher, Kayed Akkawi, Moussa Ayyash, and Khaled Alzoubi. "Software adaptation: A conscious design for oblivious programmers". *Aerospace Conference*, 2007, p. 1–12.

[ALU08] R. Alur and G. Weiss. "RTComposer: A Framework for Real-Time Components with Scheduling Interfaces". *Proceedings of the 7th ACM international conference on Embedded software*, 2008, p. 159-168.

[AME09] J. C. Américo. "Prototypage d'un serveur d'application Java EE Jonas 5 temps-réel". Engineer degree final project report, ENSIMAG, 2009.

[AME10] J. C. Américo, W. Rudametkin, and D. Donsez. "RealtimeizeMe: A tool for automatic transformation from Java legacy to Java Real-time code". *Submitted to the 18th Intl Conference on Real-Time Networks and Systems*, 2010.

[ARN00] K. Arnold, J. Gosling and D. Holmes. "The Java Programming Language". Addison-Wesley, 2000.

[ARN99] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo and A. Wollrath. "The Jini Specification". Addison-Wesley, 1999.

[AUE07] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. T. Rajan, H. Roeck and R. Trummer. "Java takes flight: time-portable real-time programming with exotasks". *Proceedings of the conferences on languages, compilers, and tools for embedded systems*, 2007, p. 51-62.

[BACK98] G. Back, P. Tullmann, L. Stoller, W.C. Hsieh, and J. Lepreau. "Java operating systems: Design and implementation". Technical report, Department of Computer Science, University of Utah, 1998.

[BACO03] D. F. Bacon, P. Cheng and V. T. Rajan. "A real-time garbage collector with low overhead and consistent utilization". *Proceedings of the Thirtieth Annual ACM Symposium on the Principles of Programming Languages*, 2003, p. 285–294.

[BAR08] N. Bartlett. "OSGi in practice". CC-E-books, 2008.

[BAS05] S. Baskiyar, "A survey of contemporary real-time operating systems". *Informatica* 2005, 29, p. 233-240.

[BER81] Bernstein, A. and Harter, P. K. "Proving real-time properties of programs with temporal logic". *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, 1981, p. 1-11.

[BIE07] M. Beisiegel, H. Blohm, D. Booz, M. Edwards, O. Hurley, S. Ielceanu, A. Miller, A. Karmarkar, A. Malhotra, J. Marino, M. Nally, E. Newcomer, S. Patil, G. Pavlik, M. Raepple, M. Rowley, K. Tam, S. Vorthmann, P. Walker, and L. Waterman. "SCA Service Component Architecture - Assembly Model Specification", version 1.0. Technical report, *Open Service Oriented Architecture collaboration (OSOA)*, 2007.

[BIH92] T. Bihari, P. Gopinath, A. Technol, and O. Columbus. "Object-oriented real-time systems: concepts and examples". *Computer*, vol. 25, 1992, p. 25–32.

[BOL00] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Fun; M. Turnbull. "The Real-Time Specification for Java", Addison-Wesley, 2000.

[BOL05] G. Bollella, B. Delsart, R. Guider, C. Lizzi, F. Parain. "Mackinac: Making HotSpotTM Real-Time". *ISORC'2005*, 2005, p. 45-54.

[BOR09] E. Borde, F. Gilliers, G. Haïk, J. Hugue, and L. Pautet. "MyCCM-HI: un framework à composants mettant en œuvre une approche d'ingénierie dirigée par les modèles". *Génie logiciel*, 2009, p. 6-12.

[BRE02a] R. Brennan, M. Fletcher, D. Norrie. "An agent-based approach to reconfiguration of real-time

distributed control systems". *IEEE Transactions in Robotics and Automation*, vol. 18, issue 4, 2002, p. 444–451.

[BRE02b] R. Brennan, X. Zhang, Y. Xu, and D. Norrie. "A reconfigurable concurrent function block model and its implementation in real-time Java". *Integrated Computer-Aided Engineering*, vol. 9, 2002, p. 263–279.

[BRUC98] P. Brucker, "Scheduling Algorithms", 2nd edn, Springer-Verlag, Berlin.

[BRUNE02a] E. Bruneton, R. Lenglet and T. Coupaye. "ASM: A Code Manipulation Tool to Implement Adaptable Systems". *Proceedings of adaptable and extensible component systems*, 2002.

[BRUNE02b] E. Bruneton, T. Coupaye, J-B. Stefani. "Recursive and Dynamic Software Composition with Sharing". *7th International Workshop on Component-Oriented Programming (WCOP02)*, 2002.

[BRUNO09] E. Bruno and G. Bollella. "Real-Time Java Programming with Java RTS". Addison-Wesley, 2009.

[CAN06] C. Canal, J.M. Murillo, and P. Poizat. "Software Adaptation". *L'objet*, vol. 12, 2006, p. 9-31.

[CED07] W. Cedeño and P. Laplante. "An Overview of Real-time Operating Systems". *Journal of the Association for Laboratory Automation*, vol. 12, 2007, p. 40-45.

[CER04a] H. Cervantes and R.S. Hall. "A Framework for Constructing Adaptive Component- Based Applications: Concepts and Experiences". *Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE7)*, 2004, p. 130-137.

[CER04b] H. Cervantes and R. Hall. "Autonomous adaptation to dynamic availability using a service-oriented component model". *Proceedings of the 26th International Conference on Software Engineering*, 2004, p. 623-632.

[CLE95] P. Clements. "From subroutines to subsystems: Component-based software development". *American Programmer*, vol. 8, 1995, p. 31–31.

[COA07] G. Coates, "Real-Time OSGi", http://www.osgi.org/wiki/uploads/VEG/Aonix-RT-OSGi.ppt, 2007.

[COR03] A. Corsaro, C. Santoro. "A C++ Native Interface for Interpreted JVMs". *1st Intl. JTRES Workshop (JTRES'03)*. LNCS 2889, Springer, 2003.

[CUC09] T. Cucinotta, A. Mancina, G.F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusina. "A Real-Time Service-Oriented Architecture for Industrial Automation". *IEEE Transactions on Industrial Informatics*, vol. 5, 2009, p. 267-277.

[DAH99] M. Dahm. "Byte Code Engineering". *Proceedings JIT'99*, Springer, 1999.

[DIJ68] E. W. Dijkstra. "The structure of the 'T.H.E.' multiprogramming system". *CACM*, vol. 11, no. 5, 1968, p. 453-457.

[DVO04] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. "Project Golden Gate: towards real-time Java in space missions". *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004, p. 15–22.

[ESC07] C. Escoffier, R. Hall, and P. Lalanda. "iPOJO: An extensible service-oriented component framework". *IEEE International Conference on Services Computing*, 2007, p. 474–481.

[ETI06] J. Etienne, J. Cordry, and S. Bouzefrane. "Applying the CBSE paradigm in the real time specification for Java". *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, 2006, p. 218-226.

[FOX09] J. Fox and S. Clarke. "Exploring approaches to dynamic adaptation". *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, 2009, p.19-24.

[GAR93] D. Garlan, and M. Shaw. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific, 1993.

[GEI07] K. Geihs. "Selbst-adaptive software". *Informatik-Spektrum*, vol. 31, issue 2, 2008, p. 133–145.

[GUI08] N. Gui, V. de Flori, H. Sun, and C. Blondia. "A framework for adaptive real-time applications: the declarative real-time OSGi component model". *Proceedings of the 7th workshop on Reflective and adaptive middleware*, 2008, p. 35–40.

[GUID02] F. Guidec, and N. Le Sommer. "Towards Resource Consumption Accounting and Control in Java: a Practical Experience". *Workshop on Resource Management for Safe Language, ECOOP 2002*, 2002.

[HAL10] R. Hall, K. Pauls, S. Mc Culloch, and D. Savage. "OSGi In Action: Creating Modular Applications in Java" (1st ed.), Manning Publications, 2010.

[HAR01] D. Hardin. "Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the Java Virtual Machine". *4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'01)*, 2001, p. 53–59.

[HIL98] G. Hilderink. "A new Java thread model for concurrent programming of real-time systems". *Real-*

Time Magazine, 1, 1998.
[HOS10] P. Hošek, T. Pop, T. Bureš, P. Hnetynka, and M. Malohlava. "Comparison of Component Frameworks for Real-time Embedded Systems". *Federated Events on Component-Based Software Engineering and Software Architecture*, 2010.
[HU07] J. Hu, S. Gorappa, J. A. Colmenares, and R. Klefstad. "Compadres: A lightweight component middleware framework for composing distributed, real-time, embedded systems with real-time Java". *Proc. ACM/IFIP/USENIX 8th Int'l Middleware Conference*, 2007, vol. 4834, p. 41-59.
[HUA05] Jinfeng Huang. "Predictability in Real-Time System Design". PhD thesis, Technische Universiteit Eindhoven, The Netherlands, September 2005.
[HUR95] W. Hürsch and C. Lopes. "Separation of concerns". Northeastern University, Technical report NU-CCS-95-03, 1995.
[IAB00] Imsys AB. "The Cjip java microprocessor". http://www.imsys.se, May 2000.
[IEE90] Institute of Electrical and Electronics Engineers. "IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries", 1990.
[IEE96] Institute of Electrical and Electronics Engineers. "Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface (API) [C Language]", 1996.
[ISO00] D. Isovic and G. Fohler. "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints". Proceedings 21st IEEE Real-Time Systems Symposium, 2000, p. 207-216.
[IVE02] A. Ive. "Implementation of an embedded real-time java virtual machine prototype". Licentiate thesis, Department of Computer Science, Lund Institute of Technology, 2002.
[J2ME] Java 2 platform micro edition (j2me) technology for creating mobile devices. http://www.java.sun.com, May 2000. Sun Microsystems Inc. White Paper.
[JCO00] J Consortium. "Real-time core extensions for the Java platform". Revision 1.0.10, J Consortium, 2000.
[JSR284] JSR 284. "Resource Consumption Management API". http://jcp.org/en/jsr/detail?id=284
[KAZ00] I. H. Kazi, H. H. Chen, B. Stanley and D. J. Lilja. "Techniques for obtaining high performance in Java programs". *ACM Comput. Surveys*, vol. 32, 2000, p. 213–240.
[KEP03] J. O. Kephart and D.M. Chess. "The Vision of Autonomic Computing". *Computer*, 2003, p. 41-50.
[KIC01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. "An overview of AspectJ". *ECOOP 2001*, 2001.
[KOP98] H. Kopetz. "Component-based design of large distributed real-time systems". *Control Engineering Practice*, vol. 6, 1998, p. 53-60.
[KRA90] J. Kramer and J. Magee. "The evolving philosophers problem: Dynamic change management". *IEEE Transactions on Software Engineering*, vol. 16, issue 11, 1990, p. 1293-1306.
[LAP06] P. Laplante. "Real-Time Systems Design & Analysis". 3rd Edition, Wiley India Pvt. Ltd., 2006.
[LAR95] K. G. Larsen, P. Pettersson, and Wang Yi. "Model-checking for realtime systems". *Proceedings of 10th International Fundamentals of Computation Theory*, vol. 965 LNCS, 1995, p. 62– 88.
[LAU07] K. Lau and Z. Wang. "Software component models". *IEEE Transactions on Software Engineering*, vol. 33, 2007, p. 709–724.
[LIU73] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment". *Journal of the ACM (JACM)*, vol. 20, 1973, p. 46–61.
[MAE87] P. Maes. "Concepts and experiments in computational reflection". *Conference proceedings on Object-oriented programming systems, languages and applications*, 1987, p. 147–155.
[MAG96] J. Magee, and J. Kramer. "Dynamic Structure in Software Architectures". *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1996, p. 3-14.
[MAH07] Z. Mahmood, "Service-Oriented Architecture: Tools and Technologies", *Proceedings of the 11th WSEAS International Conference on Computers*, 2007, pp. 485-490.
[MCGH98] H. McGhan and M. O'Connor. "picoJava: a direct execution engine for Java bytecode". *IEEE Computer*, vol. 31, issue 10, 1998, p. 22-30.
[MCGR08] C. McGregor and J.M. Eklund, "Real-Time Service-Oriented Architectures to Support Remote Critical Care: Trends and Challenges". *32nd Annual IEEE International Computer Software and Applications Conference*, 2008, p. 1199-1204.
[MCK04] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. "Composing adaptive software". *Computer*, vol. 37, 2004, p. 56-64.
[MIY97] A. Miyoshi, H. Tokuda, and Kitayama T. "Implementation and evaluation of real-time Java threads". *Proceedings of the IEEE Real-Time Systems Symposium*, 1997, p. 166–175.
[MUN70] R. Muntz and E. Coffman Jr. "Preemptive scheduling of real-time tasks on multiprocessor

systems". *Journal of the ACM (JACM)*, vol. 17, 1970, p. 324-338.
[NEE93] R. M. Needham. "Denial of Service". *ACM Conf. on Computer and Communications Security*, 1993, pp. 151-153.
[NILSE96] K. Nilsen. "Issues in the design and implementation of real-time Java" *Java Developer's Journal*, vol. 1, 1996, p. 44.
[NILSE98] K. Nilsen. "Adding real-time capabilities to the Java programming language". *Communications of the ACM*, 1998, p. 44–57.
[NILSS02] A. Nilsson, T. Ekman, and K. Nilsson. "Real Java for real time - gain and pain". *Proceedings of the International conference on compilers, architecture and synthesis for embedded systems*, 2002, p. 304-311.
[OMG] Object Management Group. "CORBA Component Model", V3.0 formal specification, http://www.omg.org/technology/documents/formal/components.htm
[OMM00] R. Ommering, F. Linden, J. Kramer and J. Magee. "The Koala Component Model for Consumer Electronics Software". IEEE Computer, volume 33, issue 3, 2000, p. 78-85.
[ORE08] P. Oreizy, N. Medvidovic, and R. Taylor. "Runtime software adaptation: framework, approaches, and styles". *30th International Conference on Software Engineering*, 2008, p. 899–910.
[OSG05] OSGi Alliance. "OSGi Service Platform Core Specification Release 4". http://www.osgi.org, 2005.
[PAN09] M. Panahi, W. Nie, and K. Lin. "A Framework for Real-Time Service-Oriented Architecture". *2009 IEEE Conference on Commerce and Enterprise Computing*, 2009, p. 460-467.
[PAP03] M. Papazoglou and D. Georgakopoulos. "Service-oriented computing". *Communications of the ACM*, vol. 46, 2003, p. 25–28.
[PAR72] D. Parnas. "On the criteria for decomposing systems into modules". *CACM*, vol. 15, no. 12, 1972, p. 1053-1058.
[PEN89] D. Peng and K. Shin. "Static allocation of periodic tasks with precedence constraints in distributed real-time systems". *Proc 9th Int. Conf. on Distributed Computer Systems*, 1989, p. 190-198.
[PFE04] M. Pfeffer and T. Ungerer. "Dynamic real-time reconfiguration on a multithreaded Java-microcontroller". *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2004, p. 86-92.
[PIZ08] F. Pizlo and J. Vitek. "Memory management for real-time Java: State of the art". *Proceedings of the 11th IEEE International Symposium on Object-oriented Real-Time Distributed Computing* (ISORC'08), 2008, p. 248-254.
[PIZ10] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. "High-Level Programming of Embedded Hard Real-Time Devices". *EuroSys Conference*, 2010, p. 69-82.
[PLS08] A. Plšek, F. Loiret, P. Merle, and L. Seinturier. "A component framework for java-based real-time embedded systems". *Middleware*, 2008, p. 124–143.
[PRO08] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, and J. Adamek. "A Component-Oriented Framework for Spacecraft On-Board Software". *Proceedings of DAta Systems In Aerospace (DASIA 2008)*, European Space Agency Report Nr. SP-665, 2008.
[RASC05] A. Rasche and A. Polze. "Dynamic Reconfiguration of Component-based Real-time Software". *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2005, p. 347-354.
[RASM08] R. Rasmussen and M. Trick. "Round robin scheduling – a survey". *European Journal of Operational Research*, vol. 188, 2008, p. 617–636.
[RED02] B. Redmond and V. Cahill. "Supporting Unanticipated Dynamic Adaptation of Application Behaviour". *16th European Conference on Object-Oriented Programming*, 2002, p. 205-230.
[RICHA09] T. Richardson, A.J. Wellings, J.A. Dianes, and M. Díaz. "Providing temporal isolation in the OSGi framework". *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'09)*, 2009, pp. 1-10.
[RICHT03] K. Richter, M. Jersak, and R. Ernst. "A formal approach to MpSoC performance verification". *IEEE Computer*, vol. 36, 2003, p. 60–67.
[ROU08] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen and E. Stav. "Composing components and services using a planning-based adaptation middleware". *Software Composition*, 2008, p. 52–67.
[SAN02] C. Santoro. "An Operating System in a Nutshell". Internal Report, Dept. of Computer Engineering and Telecommunication, UniCT, Italy, 2002.
[SCHM02] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. "Tao: A pattern-oriented object request broker for distributed real-time and embedded systems". *IEEE Distributed Systems Online*, vol. 3, issue 2, 2002.

[SCHO07] M. Schoeberl, H. Sondergaard, B. Thomsen and A. P. Ravn. “A profile for safety critical Java”. *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’07)*, 2007, p. 94–101.

[SCHW95] E. Schweitz and N. Raleigh. “Real-Time Languages”. Term. Paper, 1995, p. 1-7.

[SHA04] P.K. Sharma, J.P. Loyall, G.T. Heineman, R.E. Schantz, R. Shapiro, and G. Duzan. “Component-based dynamic QoS adaptations in distributed real-time and embedded systems”. *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA’04)*, 2004, p. 1208-1224.

[SHI94] K. Shin and P. Ramanathan. “Real-time computing: A new discipline of computer science and engineering”. *Proceedings of the IEEE*, vol. 82, 1994, p. 6–24.

[SRI02] S. Sriram, S. and S. S. Bhattacharyya. “Embedded Multiprocessors: Scheduling and Synchronization”. Marcel Dekker Inc., 2002.

[STA04] J. Stankovic and R. Rajkumar. “Real-Time Operating Systems”. *Real-Time Systems*, vol. 28, 2004, p. 237-253.

[STA90] J. Stankovic and K. Ramamritham. “What is predictability for real-time systems?”. *Real-Time Systems*, vol. 2, 1990, p. 247–254.

[STA92] J. Stankovic. “Real-time computing”. *BYTE* (Invited paper), 1992, p. 1-19.

[STO92] A. D. Stoyenko. “The evolution and state-of-the-art of real-time languages”. *Journal of Systems and Software*, vol. 18, Issue 1, April 1992, p. 61-83.

[TAY09] R. Taylor, N. Medvidovic, and P. Oreizy. “Architectural Styles for Runtime Software Adaptation”. *WICSA/ECSA*, 2009, p. 171-180.

[TOU08] L. Touseau, D. Donsez, and W. Rudametkin. “Towards a SLA-based Approach to Handle Service Disruptions”. *IEEE International Conference on Services Computing,* 2008, p. 415-422.

[TOU10] L. Touseau. “Politique de liaison aux services intermittents dirigée par les accords de niveau de service”. PhD thesis, Université Joseph Fourier - Grenoble I, France, June 2010.

[TSA06] W. Tsai, Y. Lee, Z. Cao, Y. Chen and B. Xiao. “RTSOA: Real-time service-oriented architecture”. *2nd IEEE International Workshop Service-Oriented System Engineering (SOSE'06)*, 2006, p. 49–56.

[TYM98] P. Tyma. “Why are we using Java again”. *Communications of the ACM*, vol. 41, n. 6, 1998, p. 38-42.

[VER99] D. Verma. “Supporting Service Level Agreements on IP Networks”. Macmillan Technical Publishing, USA, 1999.

[WAN03] N. Wang and C. Gill. “Improving real-time system configuration via a QOS-aware CORBA component model”. *International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack*, 2003, p. 273-282.

[WEIC04] B. Weichel, M. Herrmann. “A Backbone in Automotive Software Development Based on XML and ASAM/MSR”. *SAE World Congress 2004*, Nr. 2004-01-295.

[WEIZ93] M. Weiser. “Hot Topics: Ubiquitous Computing”. *Computer*, 1993, p. 71-72.

[WER97] M. Wermelinger. “A hierarchic architecture model for dynamic reconfiguration”. *Proceedings of 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*, 1997, p. 243-254.

[ZHA00] X. Zhang, S. Balasubramanian, R. W. Brennan and D. H. Norrie. “Design and implementation of a real-time holonic control system”. *Information Science Special Issue on Computational Intelligence for Manufacturing*, vol 27, issues 1–2, 2000, p. 23–44.

ANEXOS

Um Estudo do Impacto de Restrições de Tempo-Real em Aplicações Java/OSGi

João Claudio Rodrigues Américo

Universidade Federal do Rio Grande do Sul, Instituto de Informática - Porto Alegre, Brasil

RESUMO

No contexto das aplicações Java, a popularização da abordagem orientada a serviços e da engenharia de software baseada em componentes fez com que muitas aplicações migrassem para plataformas dinâmicas, como OSGi [OSG05]. Entretanto, com a popularização de soluções como a RTSJ [BOL00] para o desenvolvimento de aplicações de tempo-real em Java, vemo-nos em uma situação em que essas aplicações podem possuir restrições temporais que não serão respeitadas devido ao dinamismo fornecido por estas plataformas. Neste trabalho, propomos uma estratégia de congelamento da arquitetura das aplicações abrigadas na plataforma durante tratamentos de tempo-real. As reconfigurações são executadas depois, contanto que as mesmas não desrespeitem acordos de nível de serviço entre o usuário e o prestador do serviço. Nossa abordagem foi implementada sob a forma de uma extensão do modelo de componentes iPOJO [ESC07]. Esta estratégia, apesar de suas limitações, impede que a plataforma introduza impreditibilidade na execução de aplicações de tempo-real.

PALAVRAS-CHAVE: Tempo-real, arquiteturas orientadas a serviços, modelos de componentes orientados a serviços, Java, RTSJ, OSGi, evolução dinâmica de software

1 INTRODUÇÃO

Adaptação dinâmica e restrições de tempo-real são duas características e necessidades comuns em software atualmente. Enquanto o primeiro prioriza flexibilidade e o tratamento em tempo de execução de mudanças imprevistas no meio, o segundo prima por preditibilidade e determinismo no tempos de resposta de uma aplicação. Muitas soluções para lidar separadamente estes aspectos foram desenvolvidas para a plataforma Java. Uma das soluções de tempo real mais adotada para Java é a Especificação de Tempo-Real para Java (em inglês "Real-Time Specification for Java" - RTSJ) [BOL00] e suas implementações que oferecem uma interface de programação completa para o desenvolvimento de aplicações de tempo-real em Java. Da mesma forma, a popularização dos princípios da Engenharia de Software Baseada em Componentes (en inglês "Component-Based Software Engineering" - CBSE) [CLE95] e das Arquiteturas Orientadas a Serviços (em inglês "Service-Oriented Architectures" - SOA) [PAP03] para o desenvolvimento de aplicações flexíveis e modulares em Java são responsáveis pela criação de modelos de components orientados a serviços e de plataformas de serviço.

Uma das plataformas de serviço mais populares é a plataforma de serviços OSGi. Ela foi projetada para ser uma especificação aberta de plataformas para o desenvolvimento e a implantação de serviços em passarelas residenciais, mas acabou tornando-se um padrão de facto para o desenvolvimento de aplicações modulares e flexíveis em Java. Sua popularização foi possível graças à sua adoção pela Fundação Eclipse para desenvolver plugins para seu ambiente de desenvolvimento integrado (em inglês "Integrated Development Environment" - IDE). A plataforma OSGi pode ser utilizada para criar aplicações orientadas a serviços e baseadas em componentes. Atualmente a plataforma OSGi encontra-se na sua quarta versão, passando a cobrir diversos domínios de aplicação, como telefonia móvel, supervisão industrial, automóveis e a nova geração de servidores de aplicações Java Enterprise Edition.

É inquestionável o fato de que geralmente esses dois domínios de aplicação (tempo-real e dinâmicamente adaptável) possuem conjuntos disjuntos de aplicações, dado o conflito entre preditibilidade e flexibilidade. Entretanto, dois fatores nos motivam a procurar soluções para o potencial número crescente de aplicações que se encontram na interseção desses dois domínios:

1) O fato de que mesmo aplicações críticas, que não podem ter sua execução interrompida, precisam ser atualizadas e passar por manutenção;
2) E o fato de que o sucesso do SOA e do CBSE estão levando inúmeras empresas a migrarem suas aplicações para plataformas de serviços e de componentes (como os servidor de aplicação JOnAS e BEA WebLogic Real-Time).

Este estudo concentra-se na identificação dos problemas gerados pelas modificações dinâmicas na arquitetura de aplicações de tempo-real em Java e na sugestão de estratégias para evitar a introdução de impreditibilidade da parte da plataforma OSGi nas aplicações abrigadas pela mesma. Duas abordagens complementares são propostas: Uma política de congelamento da arquitetura das aplicações executadas pela plataforma durante tratamentos de tempo-real; e o monitoramento baseado em acordos de nível de serviço das reconfigurações estruturais efetuadas em fases não-criticas. Nossa política de congelamento de arquitetura foi implementada através de uma extensão ao modelo de componentes iPOJO, demonstrando a viabilidade da mesma.

O resto deste artigo é organizado da seguinte forma. A seção dois apresenta os trabalhos relacionados nos domínios relevantes ao nosso estudo. A contribuição deste trabalho é apresentada na seção três. A seção quatro apresenta a implementação realizada para dar uma forma tangível à nossa proposição e como ela foi validada. Finalmente, a seção cinco apresenta as nossas conclusões e as perspectivas de trabalhos futuros relacionados a este.

2 ESTADO DA ARTE E TRABALHOS RELACIONADOS

Este trabalho situa-se na interseção entre três domínios: Interessamo-nos em um primeiro momento às aplicações de tempo-real e às extensões sugeridas para a plataforma Java para o suporte de aplicações de tempo-real. Após, discutimos brevemente sobre evolução dinâmica de software, as motivações para fazê-lo, as principais abordagens utilizadas e como estas tecnicas vem sido utilizada no contexto de aplicações de tempo-real. Em um terceiro momento introduzimos a plataforma de serviços OSGi e apresentamos os principais trabalhos descritos na literatura sobre OSGi e tempo-real.

2.1 Java para Tempo-real

Sistemas de tempo-real diferem de outros sistemas informáticos devido ao fato de que sua corretude depende não somente de aspectos funcionais, mas também de aspectos temporais [STA92]. Esses aspectos temporais são normalmente expressos sob a forma de prazos determinados para a execução de tarefas. Estas tarefas podem ser periódicas, aperiódicas ou esporádicas, dependendo de seu padrão de chegada [ISO00]. De acordo com a criticidade dessas tarefas e da importância de respeito dos prazos, as aplicações de tempo-real podem classificadas em brando, firme e duro [BRUNO00]. Duas propriedades são fundamentais em sistemas de tempo-real:

1) Eles devem ser **preditíveis**, ou seja, as restrições temporais e prazos de uma aplicação devem ser respeitados. Técnicas de *análise de escalonamento* e *verificação formal* são frequentemente utilizadas para verificar a preditibilidade de um sistema;
2) E eles devem ser **determinísticos**, ou seja, a execução da aplicação deve ser assegurada, mesmo com a presença de fatores externos que podem perturbá-la (e neste caso, alterar sua funcionalidade, performance e tempo de resposta). O determinismo de uma aplicação pode ser medido através de sua *latência* (tempo entre a geração de um evento e a resposta do sistema para o mesmo) e de seu *jitter* (variação estatística da latência).

Para desenvolver aplicações de tempo-real, toda a infrastrura sobre a qual a aplicação é executada deve ser tempo-real. Isto gerou a criação de algoritmos para o escalonamento de tarefas de tempo-real, de sistemas operacionais de tempo-real e de linguagens de programação para o desenvolvimento de sistemas de tempo-real. Em se tratando das linguagens de programação de tempo-real, estas podem apresentar o suporte para a expressão de restrições de tempo real através de três formas diferentes:

1) Eliminando comandos cujo tempo de execução é indeterminado;
2) Baseando-se em chamadas de sistema específicas a um sistema operacional de tempo-real; ou
3) Extendendo linguagens existentes.

No caso da plataforma Java, a terceira opção foi utilizada como estratégia para permitir a criação de sistemas de tempo-real com sua popular linguagem de programação.

Java [ARN00] é uma tecnologia desenvolvida pela Sun Microsystems em 1995, consistindo na definição da linguagem de programação Java, na definição de uma biblioteca padrão e na definição de um conjunto de instruções intermediário chamados byte-codes, juntamente com um ambiente de execução. Java tornou-se uma linguagem extremamente popular entre desenvolvedores de aplicação, devido a fatores como a sua portabilidade, flexibilidade, robustez e facilidade de aprendizado [TYM98]. Entretanto, apesar das várias vantagens introduzidas por esta tecnologia, a mesma é inapropriada para o desenvolvimento de sistemas de tempo-real [NILSS02], especialmente pelos seguintes fatores:

- **Gerenciamento automático de memória:** A plataforma Java define um coletor de lixo (em inglês, "Garbage Collector" - GC) como processo de gerenciamento automático de memória. Coletores de lixo facilitam a programação e servem como meio de evitar vazamentos de memória. Entretanto, para tal eles devem varrer a memória e encontrar os objetos de um programação que não são mais referenciados, um processo cuja duração não pode ser determinada.
- **Carregamento dinâmico de classes:** O carregamento dinâmico de classes em Java permite que apenas o carregamento de uma classe seja feito apenas quando ela é utilizada por uma aplicação. Isto faz com que a todo instante apenas as classes necessárias estejam presentes na memória. Porém, o processo de carregamento de classes inclui a resolução dessas classes (carregamento das classes que ela necessita), a verificação da corretude de seu byte-code e o carregamento do arquivo diretamente do disco, um processo que pode depender do tamanho da classe e da velocidade do meio.
- **Política de escalonamento de threads:** O comportamento de threads e de escalonadores em Java é definido de uma forma muito relaxada, onde threads com prioridades mais baixas podem preemptar threads com prioridades mais altas. Embora este comportamento impeça que threads de entrarem em estado de inanição (em inglês, "starvation"), ele viola os princípios de precedência de tarefas em sistemas de tempo-real, podendo introduzir impredibilidade no sistema.

Como podemos ver, os problemas que impedem Java em sua forma convencional de ser utilizado na concepção de aplicações de tempo-real não vem da linguagem, e sim do ambiente de execução. Por isso, durante os anos 90, inúmeras tentativas de extender a linguagem foram feitas, culminando em 2000 na RTSJ, desenvolvida por equipes como Sun e IBM. A RTSJ define novas regras para o ambiente de execução Java, assim como fornece uma interface de programação completa para a criação de aplicações de tempo-real em Java. Duas modificações merecem especial atenção:

- **Escalonamento e despacho de threads:** O escalonador da RTSJ é baseado em prioridades, possuindo ao menos 28 prioridades de tempo-real (mais elevadas que as prioridades anteriormente definidas por Java), sendo que as threads de tempo-real só podem ser preemptadas por outras threads de prioridade superior (política "run-to-block"). Além disso, o escalonador pode fazer análises de viabilidade para um escalonamento. Threads de tempo-real e outros objetos escalonáveis, como tratadores de eventos assíncronos, são ligados a objetos que representam a sua demanda de recursos.
- **Gerenciamento de memória:** A RTSJ extende o modelo de gerenciamento de memória da plataforma Java, adicionando zonas de memória onde o coletor de lixo não percorre.

Além dessas modificações, a RTSJ também propõe alterações nos mecanismos de sincronização e partilha, tratamento assíncrono de eventos, transferência assíncrona do fluxo de execução, terminação assíncrona de threads e acesso à memória física. Atualmente encontramos diversas implementações da RTSJ, sendo as implementações comerciais as mais populares.

Como dito anteriormente, aplicações de tempo-real devem ser preditiveis e confiáveis. Entretanto, muitas das aplicações de tempo-real que encontramos atualmente são dinâmicas, ou seja, fatores externos podem exigir modificações e adaptações em tempo de execução. Discutiremos na próxima sessão sobre adaptação dinâmica de software e como isto é aplicado no contexto de sistemas de tempo-real.

2.2 Adaptação dinâmica de software tempo-real

Os primeiros trabalhos sobre a importância da estruturação de uma aplicação datam do fim dos anos 60 [DIJ68, PAR72]. Estes trabalhos foram a base de uma disciplina da Engenharia de Software chamada Arquitetura de Software. Arquitetura de Software estuda formas de estruturar sistemas, representando os seus componentes, suas interconexões e as regras que regem sua

evolução ao longo do tempo [GAR93]. Arquiteturas de software são soluções importantes para lidar com a escalabilidade, evolução e complexidade de sistemas de software. Arquiteturas dinâmicas de software são arquiteturas na qual a composição dos componentes muda durante a execução do sistemas. Avanços nesse domínio foram motivados pelo surgimento d a computação obíqua [WEIZ93] e do crescimento da computação autônoma [KEP03]. As principais motivações para a adaptação dinâmica de software são o custo, risco e inconveniência de ter de parar um sistema por causa de modificações no meio. Um exemplo é a atualização dinâmica de software, na qual uma aplicação é capaz de se atualizar para reparar bugs e adicionar novas funcionalidades sem ter de interromper sua execução, como no caso de sistemas críticos e nonstop [MAG96]. O custo de toda essa flexibilidade reflete-se na segurança do sistema, uma vez que os efeitos de uma modificação feita em tempo de execução não podem ser previstos.

Diversas técnicas foram desenvolvidas para prover um comportamento dinâmicamente adaptável a aplicações, entre elas:

- **CBSE:** Componentes são unidades de software que são compostas para construir um sistema. Cada componente possui interfaces especificadas contratualmente e dependencias explícitas. Estes componentes podem ser compostos dinamicamente através de ligações retardadas (em inglês, "late binding"). CBSE fornece um mecanismo de base para evoluir a arquitetura de um sistema dinâmicamente;
- **Linguagens de Descrição de Arquitetura:** As linguagens de descrição de arquitetura (em inglês, "Architecture Description Language" - ADL) são linguages utilizadas para descrever arquiteturas de software. Elementos comuns de ADLs são componentes, conexões e configurações. ADLs podem ser utilizadas para especificar pontos de variabilidade nas arquiteturas de software.
- **SOA Dinâmico:** Arquiteturas orientadas a serviços são um estilo arquitetural e modelo de programação baseado no conceito de "serviços". Serviços são unidades de software cujas funcionalidades e propriedades são declaradas em um descritor de serviços. Além disso, serviços podem ser orquestrados e compostos para formar serviços mais complexos. Prestadores de serviços registram seus serviços em um Registro de Serviços. Usuários de serviços buscam serviços junto ao mesmo Registro de Serviços. Após negociação e acordo dos termos de uso de serviço, o usuário de serviços é conectado ao prestador de serviços. SOA Dinâmico [CER04] é uma extensão ao modelo SOA que considera que serviços podem aparecer, desaparecer ou modificar suas propriedades em tempo de execução.

Sistemas dinamicamente adaptáveis de tempo-real podem ser utilizados para implementar sistemas que necessitam de flexibilidade, sistemas adaptáveis cujas interações necessitam respeitar restrições de tempo-real ou ambos. Vários trabalhos na literatura visam a reconfiguração dinâmica de aplicações de tempo-real. Entre as principais estratégias para a criação de frameworks para a adaptação dinâmica de aplicações de tempo-real estão o uso de "modos" (arquiteturas pré-estabelecidas em tempo de compilação que podem ser trocadas em determinados momentos da execução) [BOR09, PRO08, WEIC04], de extensões de ORBs (em inglês, "Object Request Broker", um módulo que intermedia as requisições de clientes em uma rede e as envia aos objetos correspondentes) para tempo-real [WAN03, CARD] e a componentização de objetos representando atributos de qualidade de serviço [QUO, SHA04].

2.3 OSGi para Tempo-Real

A plataforma de serviços OSGi é uma especificação que adiciona mecanismos de modularização à tecnologia Java [HAL10]. As unidades de software são chamadas "bundles". Um bundle é constituído de um arquivo .jar e de metadados especificando seu nome simbólico, sua versão e suas dependências. Cada bundle em OSGi possui seu próprio carregador de classe e as interações entre eles são possíveis graças a um sistema explicito de importação e exportação de packages. Bundles podem instalados, desinstalados e atualizados dinamicamente, sem necessidade de parar e reiniciar a plataforma. Além disso, eles também publicar e consumir serviços dinamicamente. Embora grande parte do dinamismo da plataforma seja gerado automaticamente, desenvolvedores devem levar em conta o fato de que bundles e serviços podem aparecer e desaparecer inesperadamente.

A plataforma OSGi tem sido uma tecnologia adotada principalmente nos contextos de automação residencial e ambientes pervasivos, devido ao seu modelo de componentes orientado a serviços dinamico e aos mecanismos de gerenciamento remoto e de implantação contínua. Entretanto, a plataforma de serviços OSGi não possui suporte para aplicações de tempo-real, e a instalação e desinstalação dinâmica de bundles pode introduzir impreditibilidade em aplicações deste tipo. Poucos trabalhos dedicam-se ao suporte de aplicações de tempo-real na plataforma OSGi e os que o fazem dedicam-se principalmente a questão da isolação de componentes na plataforma, não aos problemas gerados pelo dinamismo da mesma [GUI08, RICHA09, COA07].

3 PROBLEMÁTICA

Quando pensamos na execução de aplicações de tempo-real na plataforma OSGi, diversos potenciais problemas devem ser considerados. O mais evidente deles é o fato de que a plataforma OSGi não foi concebida como uma plataforma para aplicações de tempo-real. Ela é baseada em classes da API padrão de Java e a interação das classes da mesma com classes de APIs de tempo-real pode introduzir impreditibilidade na execução de componentes de tempo-real. No caso específico da RTSJ, ainda podemos nos deparar a problemas como:

- **Vazamento de memória:** Na RTSJ, definições de classe e objetos estáticos são mantidos em uma zona de memória chamada memória imortal. Essa zona de memória é única à cada máquina virtual e não possui interferência do coletor de lixo, sendo alocada no momento em que está é executada e desalocada no momento em que a mesma é terminada. Consequentemente, mesmo que desinstalemos componentes na plataforma, suas correspondentes definições de classe e objetos estáticos permanecerão alocados na memória imortal enquanto a máquina virtual permanecer executando.
- **Inanição:** As threads do framework OSGi, responsáveis pela gerência do sistema, são threads convencionais Java. Uma vez que utilizamos RTSJ e componentes com threads de tempo-real, temos que considerar que a política de escalonamento é run-to-block, ou seja, threads com prioridades maiores não são preemptadas por threads com prioridades menores, e assim, as threads de gerência da plataforma serão bloqueadas por threads das aplicações que nela são executadas. Além disso, o fato de não existir um contexto global (bundles não sabem da existência de outros bundles) torna difícil a atribuição de prioridades entre threads.

Entretanto, neste trabalho consideraremos as questões relativas ao dinamismo e não à especificação OSGi ou à plataforma Java. Para tal, usaremos um cenário de aplicação de monitoramento por detecção de movimento, ilustrado na figura 1. Um componente responsável pela detecção de movimento conecta-se à uma ou diversas câmeras, recuperando periodicamente imagens e analisando-as. Estas câmeras podem ser conectadas, desconectadas e reconfiguradas em tempo de execução.

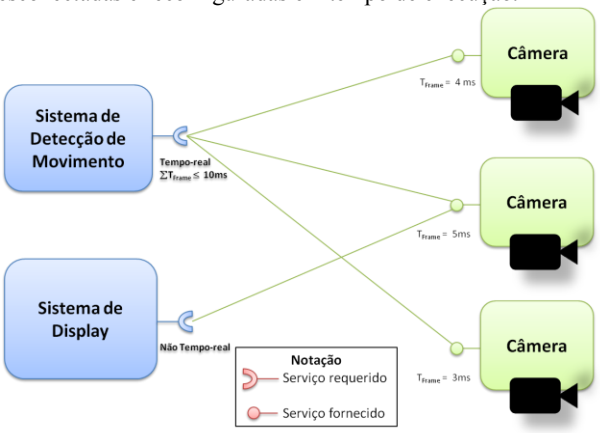

Figura 1. Cenário de Aplicação de Detecção de Movimento

Suponhamos nesse exemplo que o módulo de detecção de movimento possui restrições temporais e que o processo de busca dos frames nas diferentes câmeras não deve passar 10 ms. Neste cenário, podemos observer quatro aspectos diferentes do impacto do dinamismo de OSGi sobre aplicações de tempo-real:

- A plataforma OSGi verifica que o módulo de detecção tem dependências do tipo Camera e vai conectá-lo automaticamente às câmeras que entrarem no sistema. Suponhamos que a operação de captura e envio de imagem de uma câmera do sistema leva 4 ms e que em outra a mesma operação leva 5 ms. No momento em que uma terceira câmera aparece ela também é conectada ao módulo detector, mas se o tempo de captura e envio de imagens desta câmera for maior que 1ms, estaremos violando as restrições impostas pelo detector. Além disso, não sabemos quanto tempo vai levar a adição do módulo ao sistema e se a plataforma executar essa operação durante um tratamento de tempo-real, mais atraso pode ser introduzido.
- As câmeras também podem se desconectar durante o tempo de execução. O tempo que a plataforma leva para tirá-la do sistema e limpar todas as suas referências pode implicar em atrasos de tarefas de tempo-real. Além disso, se a câmera for retirada enquanto ela estiver sendo usada, a aplicação pode acabar chegando a um estado de erro ou inconsistência.
- Na plataforma OSGi, os componentes podem também ser atualizados em tempo de execução. O tempo dessa reconfiguração pode influir na preditibilidade de aplicações de tempo-real. Além disso, a versão atualizada do componente pode ter propriedades diferentes do antigo componente (tempo de captura e envio de imagens maior, por exemplo), violando as restrições temporais de módulos de tempo-real.
- Além dos componentes de tempo-real, outros módulos podem utilizar o serviço fornecido pelas câmeras: o fato que a câmera deve fornecer serviço para mais de um consumidor pode fazer com que um usuário não tempo-real bloqueie um usuário de serviço de tempo-real.

4 CONTRIBUIÇÃO

Propomos neste trabalho a distinção dos estados com processamentos de tempo-real, aplicando nestes uma política de congelamento da arquitetura das aplicações e impedindo que componentes sejam adicionados, removidos e atualizados. Reconfigurações podem ser feitas em fases de não-tempo-real, contanto que tais modificações não violem contratos de nivel de serviço estabelecidos entre usuários e prestadores de serviços.

4.1 Política de Congelamento de Arquiteturas de Software

Podemos ver as aplicações executadas por uma plataforma de serviços OSGi como um conjunto de estados e transições, onde cada estado representa uma possível arquitetura do sistema e as transições representam reconfigurações que levam de uma arquitetura do sistema à outra (ou seja, remoção e adição de componentes, que levam o sistema a trocar de estado, ou atualização, onde o sistema fica no mesmo estado, mas com diferentes propriedades). Tal representação é exemplificada na parte azul da máquina de estados da figura 2.

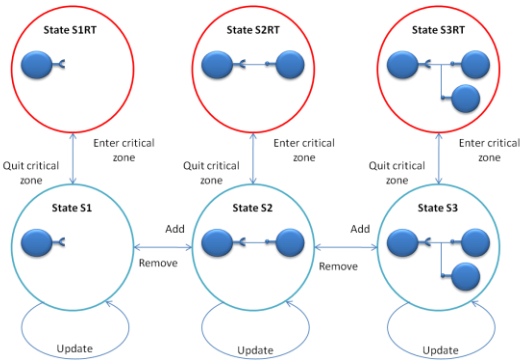

Figura 2. Diagrama de estados da arquitetura

Incluímos então estados onde a arquitetura não pode ser modificada (em vermelho, na figura 2). Estes estados correspondem à execução de seções críticas de código, onde a interferência da plataforma poderia fazer com que restrições temporais não fossem respeitadas. As reconfigurações são deixadas para estados não-críticos, monitoradas para que respeitem acordos de nível de serviço estabelecidos entre usuário e prestador de serviço.

4.2 Extensão de Acordos de Nível de Serviço para Aplicações Dinâmicas de Tempo-Real

Acordos de nivel de serviço (em inglês "Service Level Agreement" - SLA) são partes negociadas de um contrato estabelecido entre o prestador e o usuário de serviço, definindo formalmente o nível de serviço e as penalidades a aplicar quando as cláusulas não são respeitadas [VER99]. SLAs servem para atingir uma determinada qualidade de serviço. Para isso, eles contém informações como identificação das partes que assinam o contrato, o serviço prestado, o tempo de utilização do serviço, sua disponbilidade, custo e datas para renegociação do acordo. SLAs são controlados por um monitor responsável pela Gerência de Niveis de Serviço (em inglês "Service Level Management" - SLM). Este monitor é também responsável por aplicar as penalidades em caso de violação das cláusulas. A figura 3 ilustra o uso de SLAs e de SLMs na abordagem orientada a serviços.

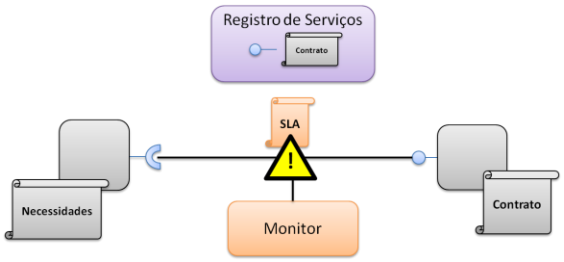

Figura 3. Acordos de nível de serviço

Neste trabalho, estendemos o modelo de SLAs dinâmicos para serviços intermitentes descrito em [TOU08], adicionando metadados que serão utilizados pelo monitor para admissão de novos componentes no sistema e gerenciamento das restrições de tempo real. No nosso acordo levamos em conta:

- Tipo de tarefa: Periódica, aperiódica ou esporádica;
- Tempo de período: No caso de tarefas periódicas;
- Tempo máximo de execução (em inglês "Worst case execution time" - WCET): Tempo máximo que a execução da tarefa pode levar
- Utilização de Recursos: Quantidade de recursos (memória RAM, CPU) necessária para a execução da tarefa.
- Prioridade: Prioridade de execução da tarefa, para fins de precedência e admissão (em casos onde apenas uma tarefa será admitida no sistema, a tarefa de maior prioridade será escolhida).
- Política: Ação a tomar no caso de desrespeito das cláusulas. Estas ações devem possuir um tempo máximo de execução conhecido.

O monitor intercepta toda reconfiguração do sistema, verificando a cada vez se o sistema está em um estado de tempo-real (e neste caso, guardando a reconfiguração para ser feita quando sair deste estado) e se as modificações respeitam as cláusulas dos SLAs.

5 Implementação e Validação

Para implementar a estratégia proposta, um protótipo foi criado baseado na extensão do modelo de componentes iPOJO [ESC07] sobre OSGi. Usar um modelo de componentes permite que possamos interferir no comportamento dos componentes sobre a plataforma OSGi sem ter que modificar o núcleo e o código da mesma. A arquitetura da solução é mostrada na figura 3.

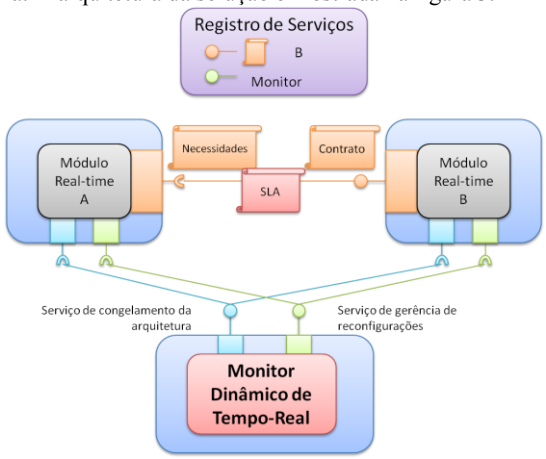

Figura 4. Extensão da Plataforma iPOJO para tempo-real

Os componentes são conectados a um monitor que verifica as necessidades do usuário e o contrato dos prestadores antes de estabelecer um SLA entre ambas as partes. Quando um componente vai executar uma reconfiguração, ele contata o serviço de gerencia de reconfigurações do monitor que vai verificar se a plataforma encontra-se em uma estado de processamento de tempo-real. Caso a plataforma não encontre-se neste estado, o monitor verifica se o nível de serviço negociado entre o prestador e o usuário não vai afetar nas restrições de tempo-real de outros componentes. Se a plataforma está em um estado de processamento de tempo-real, a reconfiguração e sua verificação são mantidas em uma lista de espera. A plataforma entra em estados de processamento de tempo-real explicitamente através do uso do serviço de congelamento da arquitetura. Antes de executar um código que necessite de preditibilidade, o componente utiliza este serviço para congelar toda a plataforma, reutilizando o serviço após o processamento para tirar a plataforma do estado de tempo-real. Devido a questões de tempo, a parte do monitor responsável pela gerência dos SLAs não foi implementada.

Nosso protótipo foi testado em uma aplicação de detecção de movimento, como descrito na seção 3. A arquitetura do sistema é mostrada na figura 4.

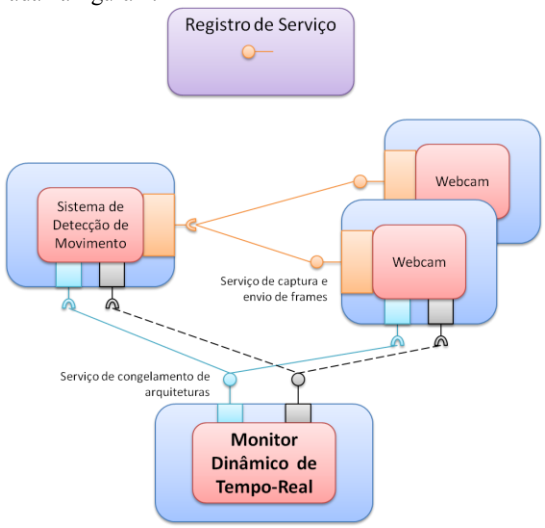

Figura 5. Arquitetura da aplicação de detecção de movimento

Como esperado, ele congelou a arquitetura da aplicação durante os períodos de tratamento de tempo-real, impedindo que componentes fossem atualizados, adicionados ou removidos da plataforma OSGi.

6 Conclusão

Este trabalho focou-se no conflito entre a preditibilidade requerida pelas aplicações de tempo-real e o dinamismo provido por plataformas de adaptação dinâmica de software, como o plataforma de serviços OSGi. Nossa motivação para interessar-se nesse conflito vem do uso crescente da plataforma OSGi para desenvolvimento de aplicações e da popularização do tempo-real para Java. Para lidar com as questões de dinamismo em aplicações de tempo-real abrigadas na plataforma OSGi, sugerimos a distinção entre a fase de processamentos de tempo-real, onde nenhuma modificação na arquitetura é permitida; e a fase de outros processamentos, onde reconfigurações podem ser feitas, sob condição de respeitar os acordos de nivel de serviço estabelecidos pelos componentes da plataforma. Nossa abordagem foi validada através de uma implementação baseada no modelo de

componentes iPOJO. Muito trabalho ainda deve ser feito, como a integração do gerenciamento de SLAs. Como todo trabalho em ciência, este trabalho também mostra perspectivas para futuros trabalhos na área, como a concepção de um nucleo de tempo-real para OSGi e esforços para a caracterização dos componentes e da mensuração de recursos utilizados. De toda forma, vemos nossa abordagem e protótipo como um primeiro passo na direção de uma extensão de tempo-real para modelos de componentes para a plataforma OSGi.

7 REFERÊNCIAS

[ARN00] K. Arnold, J. Gosling and D. Holmes. "The Java Programming Language". Addison-Wesley, 2000.

[BOL00] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Fun; M. Turnbull. "The Real-Time Specification for Java", Addison-Wesley, 2000.

[BOR09] E. Borde, F. Gilliers, G. Haïk, J. Hugue, and L. Pautet. "MyCCM-HI: un framework à composants mettant en œuvre une approche d'ingénierie dirigée par les modèles". Génie logiciel, 2009, p. 6-12.

[BRUNO09] E. Bruno and G. Bollella. "Real-Time Java Programming with Java RTS". Addison-Wesley, 2009.

[CARD] OW2 Consortium. "CARDAMOM Project". http://cardamom.ow2.org/, 2006.

[CER04] H. Cervantes and R. Hall. "Autonomous adaptation to dynamic availability using a service-oriented component model". *Proceedings of the 26th International Conference on Software Engineering*, 2004, p. 623-632.

[CLE95] P. Clements. "From subroutines to subsystems: Component-based software development". *American Programmer*, vol. 8, 1995, p. 31–31.

[COA07] G. Coates, "Real-Time OSGi", http://www.osgi.org/wiki/uploads/VEG/Aonix-RT-OSGi.ppt, 2007.

[DIJ68] E. W. Dijkstra. "The structure of the 'T.H.E.' multiprogramming system". *CACM*, vol. 11, no. 5, 1968, p. 453-457.

[ESC07] C. Escoffier, R. Hall, and P. Lalanda. "iPOJO: An extensible service-oriented component framework". *IEEE International Conference on Services Computing*, 2007, p. 474–481.

[GAR93] D. Garlan, and M. Shaw. "An Introduction to Software Architecture". *Advances in Software Engineering and Knowledge Engineering*, Volume I, World Scientific, 1993.

[GUI08] N. Gui, V. de Flori, H. Sun, and C. Blondia. "A framework for adaptive real-time applications: the declarative real-time OSGi component model". *Proceedings of the 7th workshop on Reflective and adaptive middleware*, 2008, p. 35–40.

[ISO00] D. Isovic and G. Fohler. "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints". Proceedings 21st IEEE Real-Time Systems Symposium, 2000, p. 207-216.

[KEP03] J. O. Kephart and D.M. Chess. "The Vision of Autonomic Computing". *Computer*, 2003, p. 41-50.

[MAG96] J. Magee, and J. Kramer. "Dynamic Structure in Software Architectures". *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1996, p. 3-14.

[NILSS02] A. Nilsson, T. Ekman, and K. Nilsson. "Real Java for real time - gain and pain". *Proceedings of the International conference on compilers, architecture and synthesis for embedded systems*, 2002, p. 304-311.

[OSG05] OSGi Alliance. "OSGi Service Platform Core Specification Release 4". http://www.osgi.org, 2005.

[PAP03] M. Papazoglou and D. Georgakopoulos. "Service-oriented computing". *Communications of the ACM*, vol. 46, 2003, p. 25–28.

[PAR72] D. Parnas. "On the criteria for decomposing systems into modules". *CACM*, vol. 15, no. 12, 1972, p. 1053-1058.

[PRO08] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, and J. Adamek. "A Component-Oriented Framework for Spacecraft On-Board Software". Proceedings of DAta Systems In Aerospace (DASIA 2008), European Space Agency Report Nr. SP-665, 2008.

[QUO] BBN Technologies. "Quality Objects Project". http://quo.bbn.com, 2006.

[RICHA09] T. Richardson, A.J. Wellings, J.A. Dianes, and M. Díaz. "Providing temporal isolation in the OSGi framework". *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'09)*, 2009, pp. 1-10.

[SHA04] P.K. Sharma, J.P. Loyall, G.T. Heineman, R.E. Schantz, R. Shapiro, and G. Duzan. "Component-based dynamic QoS adaptations in distributed real-time and embedded systems". *Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04)*, 2004, p. 1208-1224.

[STA92] J. Stankovic. "Real-time computing". *BYTE* (Invited paper), 1992, p. 1-19.

[TOU08] L. Touseau, D. Donsez, and W. Rudametkin. "Towards a SLA-based Approach to Handle Service Disruptions". *IEEE International Conference on Services Computing,* 2008, p. 415-422.

[TYM98] P. Tyma. "Why are we using Java again". *Communications of the ACM*, vol. 41, n. 6, 1998, p. 38-42.

[VER99] D. Verma. "Supporting Service Level Agreements on IP Networks". Macmillan Technical Publishing, USA, 1999.

[WAN03] N. Wang and C. Gill. "Improving real-time system configuration via a QOS-aware CORBA component model". International Conference on System Sciences, Software Technology Track, Distributed Object and Component-based Software Systems Minitrack, 2003, p. 273-282.

[WEIC04] B. Weichel, M. Herrmann. "A Backbone in Automotive Software Development Based on XML and ASAM/MSR". SAE World Congress 2004, Nr. 2004-01-295.

[WEIZ93] M. Weiser. "Hot Topics: Ubiquitous Computing". *Computer*, 1993, p. 71-72.

RealTimeizeMe: A tool for automatic transformation from Java legacy to Java Realtime code

João Claudio AMÉRICO[1, 2], Walter RUDAMETKIN[1, 3] and Didier DONSEZ[1]
[1] *Grenoble University 1, LIG/Adele, 38041 Grenoble, Cedex 9, France*
[2] *Informatics Institute, Federal University of Rio Grande do Sul, Caixa Postal 15064, 91501-970 Porto Alegre, Brazil*
[3] *Bull S.A.S., 1 Rue de Provence, BP208, 38432 Echirolles Cedex, France*

{Joao.Americo, Walter.Rudametkin, Didier.Donsez}@imag.fr

Abstract

The Real-time specification for Java (RTSJ) extends the standard Java API to provide mechanisms to increase predictability in applications' response times. Determinism is crucial for Real-Time applications, and desirable for most Java applications. For new projects, the inclusion of real-time features can be done at the conception or design phases, requiring few modifications; however, when considering legacy applications, it is impractical to manually modify large software to be RTSJ compliant. In this paper, we propose a means of automatically transforming java legacy code to java real-time code, thus benefiting from advances in the JVM and third party libraries regarding execution determinism. Our proposed tool, called Realtimeizeme, dynamically instruments Java classes at load-time using bytecode manipulation for adapting legacy applications to real-time applications. The tool runs as a JVM agent and has been tested instrumenting various java legacy applications. Benchmarks were performed on two different application servers and a database engine. The tool introduces a small overhead once per instrumented class. Default configuration provides conflicting results, but, in general, proper configuration improves speed and most importantly determinism.

Keywords: Real-time Java, Java agent, Dynamic bytecode manipulation, Legacy software

I. INTRODUCTION

Real-time systems differ from other information systems in the fact that their correctness depends on both functional and temporal aspects [25]. In order to satisfy these timing constraints, services and algorithms used by real-time systems must be executed in bounded time. Real-time applications do not necessarily have to be fast, but they must be predictable. Depending on the enforcement of deadlines, real-time systems may be divided into hard and soft real-time systems [25]. In hard real-time systems deadlines must be strictly enforced to avoid safety issues (e.g., weapon systems, nuclear power plants, automated transport systems). In soft real-time systems, the need for strict deadlines is more or less replaced by the need for homogeneous response times in order to ensure acceptable levels of service, thus the goal is to minimize response-time deviations. Missed deadlines are interpreted as degraded service quality, and should be avoided, but nevertheless the system continues to operate.

Java [13] has become one of the most popular general purpose languages. This popularity is in part due to its portability, reusability, security features, ease of use, robustness, rich API set and automatic memory management. Java has many advantages over traditional languages for programming, such as C and C++ [21]. In addition, nowadays it is arguably easier to find programmers with Java skills than those experienced with Ada or C. However, the same Garbage Collector that eases development is one of the main reasons why Java was not used to design critical, embedded and real-time applications. Indeed, garbage collection introduces unpredictable execution times [27]. As a result, the Real-Time Specification for Java (RTSJ) [1] was introduced, adding new features to improve the determinism of conventional Java. RTS, when compared to other solutions, alleviates developer effort in designing real-time applications by providing high-level abstractions for real-time mechanisms. Real-time Java is already being used in numerous defense and commercial applications [28].

The development of real-time systems is hindered by a fundamental issue; their design is much more complicated than that of conventional systems [4]. To truly benefit from the RTSJ, many choices must be made at design time. To start, tasks must be ordered by their importance and their priorities should be well understood and set accordingly. Thus, important tasks are handled first and without interruption, ensuring their timely completion. Communication and variable sharing must also be well designed to avoid unpleasant surprises at run-time. Specific design patterns for RTSJ applications have already been created to provide solutions for common real-time design problems [4, 20]. Even so, these design constraints limit current usage of RTSJ. Migrating legacy Java software to Real-time Java is far from being an easy task and is progressively worse when considering large and complex software [4].

We consider this an important issue to solve for two reasons:

1) Legacy applications, being the deterministic *weak-link*, neither benefit from real-time code, nor are usable by real-time applications. The majority of existing libraries are legacy java, imposing their re-implementation for RTSJ.
2) Current legacy applications would benefit from RTSJ features to increase determinism, but migration costs are generally prohibitive.

In this paper, we provide a tool that automatically transforms legacy Java applications into RTSJ compliant applications using configurable pre-defined transformations. In order to be applicable to most software, transformations need to be applied at the bytecode level and as late as possible. This goes against some of the goals in hard real-time systems, being able to statically guarantee timely execution, because late bytecode modification may introduce indeterminism, but it does make our approach a broader one and applicable to a larger array of software. Our tool, called *Realtimeizeme*, is a Java agent that instruments classes at load-time. We have tested it on popular open-source software that uses the Java technology and could benefit from increased determinism in execution times. We give an analysis of different aspects of our tool, including if an application successfully runs after our transformations, if the transformations provide increased determinism in execution times, and the overhead induced by our tool. Our goal is to provide a general solution to easily migrating existing code bases to RTSJ and deterministic libraries and to evaluate the resulting software by analyzing execution times. Our results conclude that RTSJ provides benefits in regards to real-time behavior, but automatic transformations must be performed carefully and are not always beneficial. Also, certain features of RTSJ virtual machine implementations limit taking these solutions further.

This rest of this paper is organized as follows. Section 2 presents technologies introduced in Real-time Java. In section 3 we present different approaches to code transformation. In section 4, we present *Realtimeizeme*, an implementation of our approach. Section 5 evaluates and validates the proposed tool. Section 6 discusses related works. Finally, Section 7 concludes the paper and presents our perspectives.

II. REAL-TIME JAVA

Real-time Java is a combination of different techniques allowing developers to create applications with real-time characteristics using the Java Platform. Despite the advantages provided by Java, some requirements needed for real-time systems are not met using conventional Java technology. In order to overcome these limitations, a group was created in 1998 to define real-time extensions for Java. This group included technical people from Sun, IBM and from all across the real-time industry. Their work culminated with the **Real-Time Specification for Java** (RTSJ), which defines real-time behavior in the Java Platform, through a collection of classes, constraints to the behavior of the virtual machine, an API and additional semantics. Among the concepts introduced by RTSJ, two additional programming constructs are fundamental: **Real-time Threads** and special types of **Memory Areas**.

Figure 1 shows how the RTSJ altered the thread hierarchy in Java with the inclusion of Real-Time Threads. Real-time Threads are threads whose priorities are higher than normal Java threads (java.lang.Thread instances). These priorities are set at the moment a thread is created. The virtual machine scheduler must always schedule the thread that has the highest priority to be executed. This thread will be executed in a **run-to-block** scheduling policy, which means that the thread will run until the end of its execution, only being interrupted in the case of a blocking operation or to allow the execution of a higher-priority thread. Two different types of real-time threads are provided: **Real-Time Threads** (RTT) and **No Heap Real-Time Threads** (NHRTT). NHRTT are addressed to hard-real time applications, are not allowed to access the heap memory and have a priority higher than the Garbage Collector (thus they cannot be preempted by it).

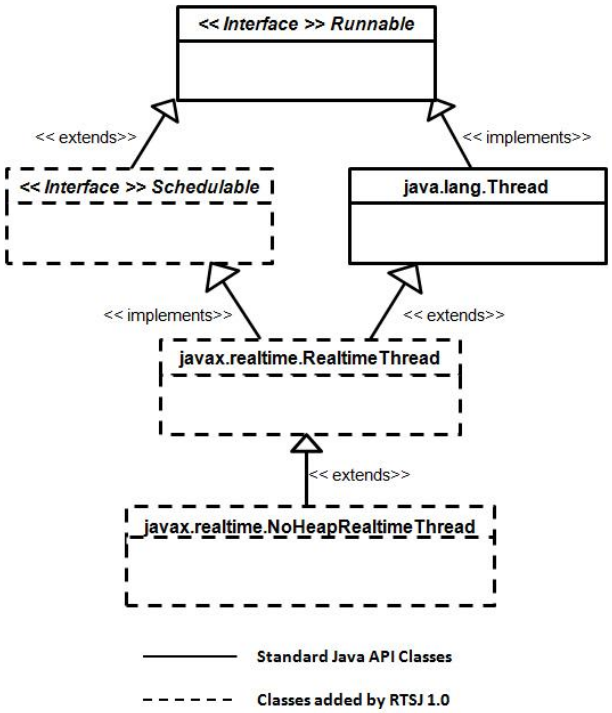

Figure 1. RTSJ thread hierarchy

Besides the traditional heap memory, two new memory areas were included. In these special memory regions, memory is not garbage-collected. Therefore, garbage collection operations do not introduce unpredictability into execution times. These areas are called **scoped** and **immortal memory**. As the name suggests, scoped memory is a memory region that is active inside a scope. When there are no more threads executing inside this memory area, it is automatically erased. In contrast, immortal memory is a region allocated during the virtual machine startup. Objects allocated in this area live throughout the application's lifetime.

However, the Garbage Collector and thread scheduling are not the only sources of indeterminism in java applications: most of the standard java libraries themselves are not real-time aware nor deterministic. Other libraries have been developed to substitute standard Java classes without introducing indeterminism. The first and most known fully RTSJ-compliant library is **Javolution** [16], an open source library that provides alternate implementations of standard library classes. Javolution claims to provide classes that are deterministic and sometimes faster than Java standard library classes. When possible, Javolution

classes implement the same interfaces as their counterparts in the Java standard library, easing substitution and retaining semantic compatibility.

III. TRANSFORMATION TECHNIQUES

Manually modifying applications' source code to benefit from RTSJ features is practical with small applications (up to the tens of thousands of lines of code). However, many existing applications that have real-time requirements are large, containing hundreds of thousands of lines of code, or even millions of lines of code. It would require a lot of time to locate all the classes and files to be changed and all the references to these classes in other files. In addition, in applications of this magnitude, manual modification can become an error-prone task. Creating real-time extensions for each application would increase its time-to-market, implying delays for each release. Moreover, large projects normally use third-party libraries, whose source code might not be available to be modified. In general, the longer projects go on, the more complex they become making the transformations required even more difficult to be implemented. Application servers, database engines and IDEs are examples of application types in which manual code manipulation, to insert real-time structures, is impractical. In addition, some application servers and IDE's dynamically generate even more code, so those generation processes would also need to be adapted. In table 1, we give statistics on popular open-source legacy software. As can be seen, their size and complexity make it time consuming to re-code these projects in order to benefit from the Real Time Specification for Java.

Name	Java LoC *#	% of Java*	Age+	Application Type
Apache Derby	574,441	87%	6 years	DBMS
Apache Felix	311,442	81%	5 years	OSGi platform
Apache Geronimo	242,385	68%	6 years	Application Server
Apache JMeter	2,897,114	52%	12 years	Load injector
Apache Sling	119,401	85%	3 years	Web Framework
Eclipse	5,292,154	68%	9 years	IDE
Glassfish	2,190,019	36%	5 years	Application Server
HSQL Database	336,620	55%	12 years	DBMS
JonAS	3,082,718	44%	12 years	Application Server
Netbeans	38,938,621	70%	14 years	IDE

*** According to Ohloh.net, metrics taken on February 22nd**
 # Blank and commented lines are not included
+ Generally, these projects already existed before their open source release. For example, Apache Felix is the evolution of Oscar project, started in 2001.

Table 1. Java legacy software

An approach to overcome the expense incurred in manually modifying large software is to use automatic manipulation and transformation techniques. Tools like Spoon [23] provide source code level manipulations, while ASM [19] and BCEL [30] can directly manipulate bytecode. Working at the source code level is generally more intuitive for the majority of developers. Source code level tools are based on the applications abstract semantic tree, and give higher level abstractions. However source code is not always available, as in the case of large projects with long lists of dependencies. Bytecode manipulation provides advantages such as not requiring source code and not requiring recompilation (i.e., the classes are already compiled) making it very fast. A bytecode level approach also resolves the problem of transforming third-party libraries, since the dependencies are always available in a compiled form at some point. Despite these advantages, this can be complex to implement.

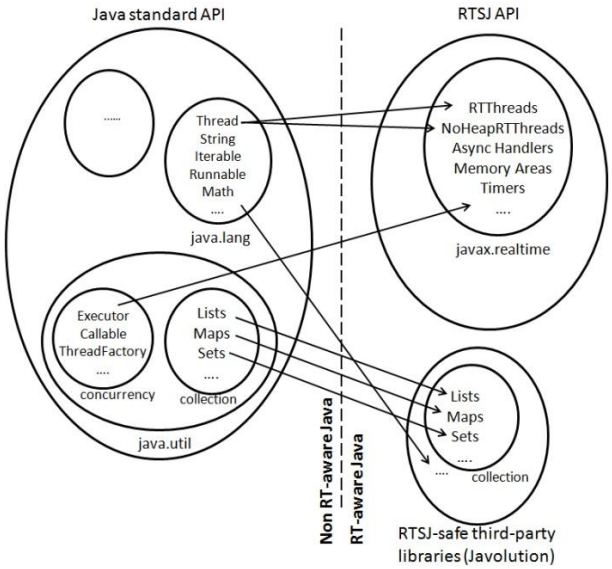

Figure 2. Mapping between standard and RTSJ-compliant classes.

We have chosen a bytecode manipulation approach, given the fact that it is a broader solution. Once we had selected the how, a second problem came to mind, the when. We have identified two different periods of interest for bytecode manipulation that give different outcomes: *build-time*, a static manipulation approach, and *load-time*, a dynamic approach. In the static approach, the tool browses through the compiled class-files, performing pertinent substitutions. This approach is useful when we have previous access to all the modules that will be deployed in the application and all their dependencies, thus we can ensure that no external code is left unanalyzed, which could potentially introduce greater unpredictability in the application. However, application servers load web applications at runtime and other applications may have mechanisms to generate new classes and new modules on-the-fly. In such an approach, these classes would not be transformed. Load-time class instrumentation can deal with these kinds of applications, since all classes can be intercepted and instrumented the first time they are loaded by the Java virtual machine. Intercepting class loading is an advantage and a disadvantage, as it works for the whole Java classloader hierarchy (i.e., no classes are missed), but it can generate significant overhead

Original program

```
void doSomething(Runnable r) {
   Thread t1 = new Thread();
   Thread t2 = new Thread(r);
   t2.start();
}
```

Realtimeized program (RTT)

```
//Thread.Type = RTT, Priority = p

void doSomething(Runnable r) {
   Thread t1 = new RealtimeThread(
      new PriorityParameters(p));
   Thread t2 = new RealtimeThread(
     new PriorityParameters(p),
     null,null,null,null,r));
   t2.start();
}
```

Realtimeized program (NHRTT)

```
//Thread.Type = NHRTT, Priority = p

void doSomething(Runnable r) {
   Thread t1 = new
NoHeapRealtimeThread(
      new PriorityParameters(p),
      ImmortalMemory.getInstance()
   );
   Thread t2 = new
NoHeapRealtimeThread(
     New
PriorityParameters(p),null,null,
  ImmortalMemory.getInstance(),null,r)
   );

   t2.start();
}
```

Figure 3. Transformation example: Thread to Real-Time Threads

depending on the number of classes loaded and their frequency. In addition, this approach would also avoid bootstrap-level modifications.

IV. IMPLEMENTATION: REALTIMEIZEME

In this paper we present our proof-of-concept implementation for a dynamic bytecode manipulation tool for introducing real-time code into Java classes: the *Realtimeizeme* tool. Our tool is implemented as a Java Instrumentation Agent [22], a program that runs embedded in a Java virtual machine. Realtimeizeme has the benefit of intercepting all classes loaded, transparently from the application, at the cost of run-time overhead introduced once for every loaded class. A configuration file is passed as an argument to the agent, to specify class substitutions and to configure the agent.

Independently of when and how the manipulation is performed, the tool finds and replaces selected non-deterministic Java classes with deterministic RTSJ-safe classes for each Java class that might introduce unpredictable response times and where a substitution is possible.

In figure 2, we present a schema representing our approach, a mapping between classes from the standard Java API towards RT-aware classes from the RTSJ API and from the third-party library, Javolution. In certain cases we can map a standard Java class to several RT-aware classes, such as standard Java threads, which may be substituted by RTT or NHRTT threads. These different mappings must be taken into account by the tool, offering different instrumentation possibilities to the developer.

RTSJ provides classes for real-time threads, their scheduling and dispatching, memory management, synchronization, resource sharing, asynchronous events, high resolution time, clocks, timers, real-time exceptions, POSIX signal handling, security policies and options for tuning the behavior of the implementation. However, RTSJ does not describe classes that cover other aspects that can also introduce unpredictability (like lazy initialization or array resizing in Java standard collections API). Thereby, RTSJ-safe libraries should be used to replace the unsafe classes.

Three sections may be distinguished in the configuration file passed to our Java agent, realtimeizeme.

Thread Replacement section: In this section developers specify which classes should have their threads substituted by real-time threads and the respective priorities to use. Normal Java threads (java.lang.Thread) can be replaced by either RTT or NHRTT threads. The tool is responsible for injecting the corresponding RTSJ code. Figure 3 shows a thread transformation example.

General Replacement section: This section is used for replacing classes with common interfaces or methods. Developers specify the class to replace (*replacee*), the class that will replace it (*replacer*), and the target. There are two sub-types of replacement:

- "*ReplaceByInterface*" is used for replacing classes that use common interfaces. It can be used when all method calls to the replacee object are method calls on the common interface, implemented also by the replacer. See figure 4.
- "*ReplaceAllReferences*" will have all references changed regardless of interfaces. A mapping from methods from the replacee class to the replacer class can be provided if required. See figure 5.

String Replacement section: Special attention has been paid to the String, StringBuffer and StringBuilder classes in the java.lang package given their popularity and the lack of implemented interfaces. These classes

Original program

```
void doSomething(Object o, int i) {
   List l = new ArrayList();
   l.add(o);
   l.get(i);
   Iterator it = l.iterator();
}
```

Realtimeized program

```
void doSomething(Object o, int i) {
   List l = new FastTable();
   l.add(o);
   l.get(i);
   Iterator it = l.iterator();
}
```

Figure 4. Transformation example: General transformations (ReplaceByInterface)

Original program

```
void doSomething(Object o, int i) {
   ArrayList a;
   {...}
   a = new ArrayList();
   a.add(obj);
   a.get(i);

  // X's method signature has an ArrayList object
   x.method(a);
   a = method_that_returns_ArrayList();
}
```

Realtimeized program

```
void doSomething(Object o, int i) {
   FastTable a;
   {...}
   a = new FastTable();
   a.add(obj);
   a.get(i);

  // X's method signature now requires a FastTable object
   x.method(a);

  // return type of the method modified
   a = method_that_returns_FastTable();
}
```

Figure 5: Transformation example: General transformations (ReplaceAllReferences)

Original program

```
void doSomething(String str, char[] ch,
                 char c, int i, int j) {
   StringBuilder sb = new StringBuilder(str);
   {…}
   sb.append(str);
   sb.insert(i, c);
   sb.append(c, i, j);
   str.toLowerCase();
   str.substring(i);
}
```

Realtimeized program

```
void doSomething(String str, char[] ch,
                 char c, int i, int j) {
   StringBuilder sb = new StringBuilder(str);
   Text t = new Text(sb.toString());
   {…}
   t.plus(string)
   t.insert(offset, Text.valueOf(c));
   t.concat(Text.valueOf(c,i,j));
   t.toLowerCase();
   t.subtext(i, t.length());
   sb = new StringBuilder(t);
}
```

Figure 6: Transformation example: String/StringBuffer/StringBuilder transformations

have methods with linear complexity (i.e., O(n)) while Javolution provides classes that perform inclusion, deletion and concatenation in O(log n). Although Javolution classes should be preferred when possible, automatic substitutions are complicated because method signatures differ, so each method must be individually considered. Wrapper objects may also be used in order to limit the effect of these replacements to a method, at the cost of additional overhead introduced by the conversions. A wrapper is used to convert the replacee into the replacer class, and at the end of the method, reconvert the replacer into a replacee class. See figure 6 for an example.

For all replacements, each replacement type provides a substitution target. Developers may specify class names, packages and methods in order to select classes that will undergo substitution. The '!' character before the class name excludes the package/class/method from substitution. The '*' is the *globber* character, which returns true when compared. The declaration order is important, early declarations take precedence (i.e., if we have *"*,!a.b.**"* as the target, a class *a.b.c* will be instrumented since *'*'* will be parsed before the exclusion *"!a.b.**"*). The same classes can be specified in different sections of the configuration file, providing different types of substitution in the same class.

Realtimeizeme is composed of two modules: *RTAgent* and *RTTransformer*. The *RTAgent* module is responsible for parsing the configuration file and defining the *ClassTransformer* object that performs the class instrumentation at load time. Its method *"premain"* is called before executing the target application [22]. The RTTransformer class extends ClassTransformer and is defined by RTAgent as being the object used for instrumentation. Its method *transform* receives a class definition and returns the instrumented bytecode which is loaded by the classloader. Inside that method, the loaded class name is compared to the class names specified in the configuration file; bytecode manipulation is performed by specified modules (thread, strings and general transformation). Those modules perform calls to classes that extend ASM classes [19], which use the Visitor pattern design to visit classes without representing their bytecode, and the Adapter pattern design to chain visits and compose transformations.

V. VALIDATION AND RESULTS

We have performed two types of tests to validate our tool: a verification to see if the application still works after using our tool and after changing to a real-time RTSJ compatible virtual machine, which we have tested on several large software; and a benchmark test to see if real-time requirements are met after the bytecode substitutions, and to see the overhead introduced by our tool at execution time. The first series of tests were performed using a Sun Java Real-Time System v2.2 virtual machine, under a Solaris 10 Update 7 environment with an AMD Turion 64 X2 TL-62 2.1 GHz processor. It is important to note that some applications, using a default configuration to replace everything, did not continue to operate correctly after using our tool, usually because of class cast exceptions. When such occurrences happened we took a trial-and-error approach, viewing the classes that were problematic and excluding them from the automatic transformation that appeared to be at cause. This could be minimized if we use a static analysis tool to ensure that substitutions are coherent and that they do not break functionality. Of course, this would increase overhead of the application and some complicated cases, like those of sophisticated type casts or reflection, might not be easily detected. In Table 2 we can see the results.

Name	Java RTS	Realtimeizeme	Executed test
Apache Derby 10.5.3.0[+]	Passed	Passed	Execution of SQL commands by the standalone application
Apache Felix 2.0.4	Passed	Passed	Execution of a sample iPOJO application
Apache Geronimo 2.2	Failed	Not tested	-
Apache JMeter 2.3.4	Passed	Passed	HTTP Charge injection in servlets
Apache Sling 5[#]	Passed	Passed	Deployment of an example web application
Eclipse 3.5.0	Passed	Passed	Creation and execution of a Java project
Glassfish v2.1.1[*]	Passed	Passed	Deployment of a sample web application
Glassfish v3[*]	Failed[1]	Not tested	-
HSQL Database 1.8.1.2	Passed	Passed	Execution of SQL commands by the standalone application
JOnAS 4.10.7	Passed	Passed	Deployment of a sample web application
JOnAS 5.1.1	Failed[2]	Not tested	-
Netbeans 6.8	Passed	Passed	Creation and execution of a Java project

*** Open Source Edition**
[#] Standalone Application
[+] Embedded driver
[1] This application server requires a JVM 6.0-compatible and Java RTS is based on JDK 5.
[2] Classes may be loaded out-of-order in real-time virtual machines

Table 2. Initial test results

All tested applications that correctly executed using the real-time virtual machine, continued to work after binding the java agent for class instrumentation with certain configuration corrections. This test demonstrates the feasibility of the transformations on large code bases.

Regarding our benchmarking tests, we performed execution tests on the JOnAS 4.10.7 and Glassfish 2.1.1 application servers, and the HSQLDB 1.8.1.2 database. These applications were selected for the benchmark because they would clearly benefit from increased determinism in response times. The tests were executed on a Sun Real-Time System 2.2 virtual machine in Solaris 10 Update 7, with a Pentium 4 HT processor running at 3GHz. For the application server benchmark, a non-instrumented J-Meter client was run on a separate machine, an AMD Turion 64 X2 TL-62 2.1 GHz processor, running Ubuntu 9.10. J-meter was used to perform HTTP accesses on servlets installed on the application servers. Client and server were connected by a local network running at 100mbits. For HSQLDB, we evaluated a sequence of executions of the test script provided with the application. For both benchmarks, three different cases were considered:

- Execution of the application without our tool;
- Execution of the application with our tool bound to the virtual-machine, intercepting every class load, but not performing any modifications (empty configuration file); such a test provides data on the overhead regarding the JVM calls to the agent at every class load
- Executing the application with our tool bound to the virtual-machine, intercepting every class load and performing modifications specified in the configuration file (as many substitutions as possible were performed).

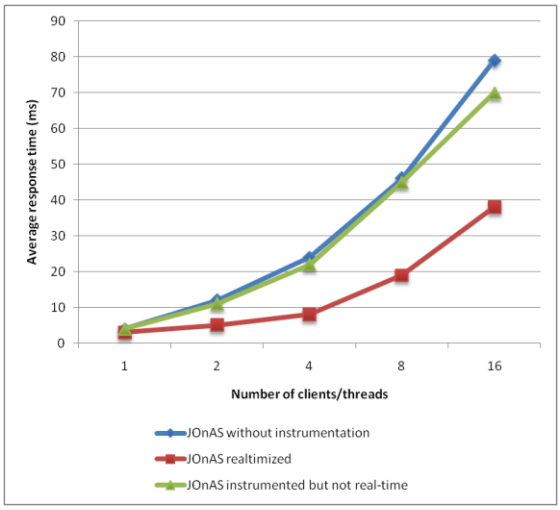

Figure 7. JOnAS - Average response time chart

Figure 7 shows the average response times obtained in our application server tests. JOnAS 4.10.7 average execution times were significantly higher without the Realtimeizeme instrumentation (from 79ms to 38ms with 16 clients connected, configured in J-Meter), specially when high concurrency takes place.

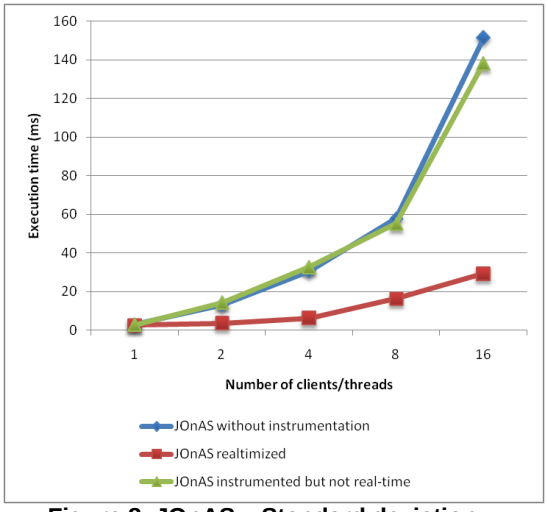

Figure 8. JOnAS – Standard deviation

Figure 8 shows the standard deviation values of the same test results shown previously. The instrumented application server presented a lower standard deviation than its non-instrumented counterpart, and shows a large improvement when many clients are accesing the servlet concurrently. This means that response times in application servers became more homogeneous after

using our tool, thus increasing the determinism and predictability in the test applications.

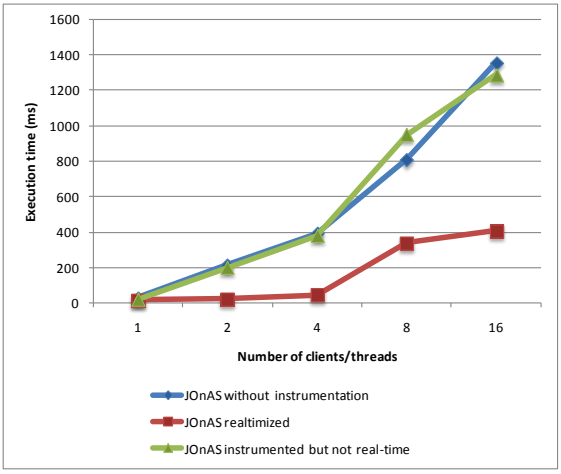

Figure 8. JOnAS – Jitter

Regarding Jitter, which is arguably one of the more important measures in real-time computing and is stated as the difference in time between the longest execution and the shortest, JOnAS has benefited from instrumentation at almost all levels.

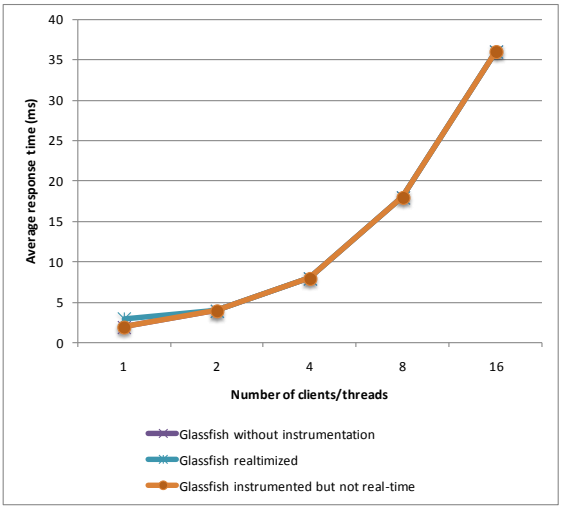

Figure 9. Glassfish - Average response time chart

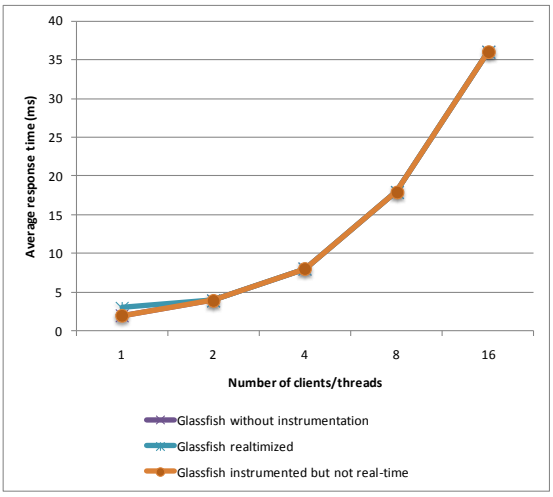

Figure 10. Glassfish – Standard deviation

As seen in both Figure 9 and Figure 10, Glassfish kept the same average execution times with or without instrumentation, independently of the amount of clients connected, and the standard deviation was the same. Specifically, in average execution times, the Glassfish tests show that our tool did not introduce significant overhead or any visible benefits.

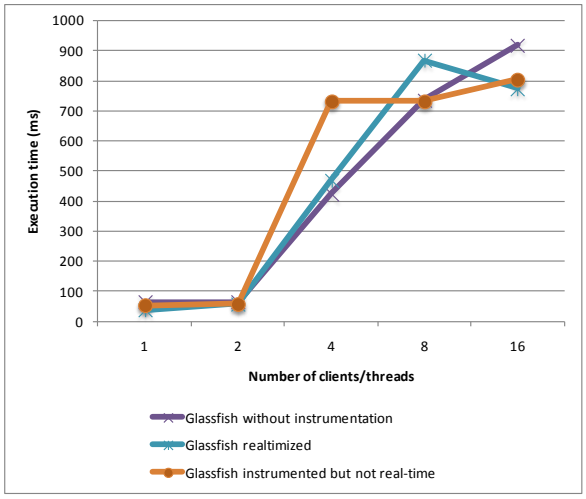

Figure 11. Glassfish – Jitter

Jitter on Glassfish, as seen in Figure 11, was less symmetric compared to average execution times and standard deviation. When 4 and 8 concurrent clients were executing, jitter was actually increased, but recovered at 16 clients.

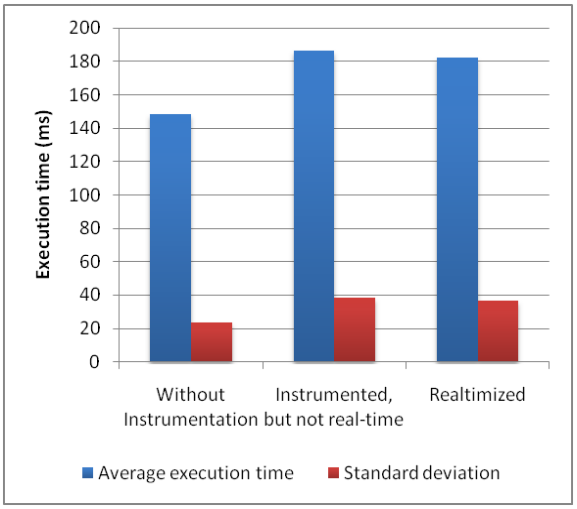

Figure 12. HSQL Database Engine In-Memory - Test Script

Figure 12 presents the average execution times and the standard deviation of our tests on the HSQL database, using the integrated in memory execution test. Running our agent, with or without instrumentation being performed, caused the application to run slower. This most likely indicates that there is overhead when activating the Java agent (even when not instrumenting classes), and that in this application the benefits of the substitutions performed were not significant enough to overcome the overhead (as in the case for Glassfish), or even worse, they were detrimental. It is also well known

that database systems use specific concurrency control algorithms to deal with real-time transactions [31]. Tested automatic transformations probably interfere with these fine tuned non-real-time concurrency control algorithms, especially considering that the thread execution policy changes from a un-specified policy (commonly implemented as round robin) to a run-to-block policy, as specified in RTSJ.

VI. Related Work

Interest in Real-time Java systems has been increasing for both, classical uses of the technology and new applications classes that would benefit, such as, real-time database systems [31], IT telecommunications frameworks, and even general-purpose web-containers. Generative programming techniques [32] have already been used to ease the development of real-time software. Some component-based frameworks that focus specifically on automatic RTSJ-compliant code generation at build time. Soleil [7,8] is a framework for creating real-time systems in Java. It provides an abstraction for RTSJ concepts during the design of the application, generating the correspondent code which allows the developer to focus on the functional parts of the system. Similarly, the framework proposed by Etienne et al. [6] provides a programming model to facilitate the design of real-time systems by means of the Dependency Injection pattern [24]. Compadres [9] is a project proposing a framework for distributed real-time embedded systems, focusing on memory management aspects. The memory management issue is also the focus of the Golden Gate project [10]. However, all of these projects propose solutions for the design layer, not applicable to legacy software, in which such modifications to structure can become costly.

Extensions for memory area management are also an intensive research domain. The difficulty of dealing with memory areas and their restrictions have motivated researchers to develop new abstractions for memory area manipulation. Mechanisms for memory regions inference [5, 2, 11] and programming models [4, 14] have already been proposed to address this issue. Deters et al. [12] proposed an aspect-oriented approach for automatically transforming Java programs into Memory Area-aware RTSJ code. However, scope computation and join points discovery are run offline and the modifications are done at source code level. A compile-time analysis strategy is used by Cherem and Rugina [2] and Garbervetsky et al. [11] for translating Java applications into programs with region-based memory management. Thus, these approaches present static solutions, not considering applications with dynamic class generation or late deployment.

A dynamic code instrumentation approach for automatic scoped memory management is also presented by Deters et al. [3]. According to this paper, static analyses need restricted assumptions about the behavior of the application. But as most of previous works, they do not deal directly with threads, their priorities nor the usage of RTSJ-unsafe libraries.

As a note, the syntax used in the configuration file to specify classes and packages to be replaced was inspired by the syntax used in BND [29] to export and import packages.

VII. Conclusion And Perspectives

In this paper we have presented an approach to automatically transform Java legacy applications to RTSJ compatible applications. Our approach uses a Java agent to instrument all classes loaded into the virtual machine. We have defined class mappings, from non-deterministic classes, to deterministic classes that exist in the RTSJ API and in RTSJ safe third party libraries, such as Javolution. Our tool, *Realtimeizeme*, is configurable, providing a means of specifying at the class level, which substitutions should be performed.

Our results have been encouraging but also are problematic. For one, in order to use our tool on large software, we must provide custom configuration files, because performing full substitutions can break the application and at this moment we do not do further analysis in order to determine the safeness of non semantically equivalent substitutions, such as those where no common interface exists. At the moment, we go by creating configurations using a *trial and error* approach. If a class causes a break (e.g., Class Cast Exception), we remove it from the substitutions. Depending on the application, this can take a little time, but in our test applications, it was less than a couple of hours, as in the case of the Eclipse framework. In general, any application that runs on the real-time virtual machine will run with our tool activated.

Regarding our benchmarks, the overhead produced by our tool is only visible when a class is loaded. After warming up (i.e., loading all classes), this overhead disappears. However, in some cases, there may be many classes created dynamically (e.g., serialization) which causes this overhead to persist. Nevertheless, during execution, we have seen that some applications improve in both performance and determinism, while others may become slower and less deterministic. For example, replacing all threads in an application, and then giving them the same real-time priority, may cause terrible performance due to the *run-to-block* scheduling policy for real-time threads. The run-to-block policy implies that when many threads of the same priority are scheduled, one thread can be executed for long periods of time causing other threads to starve, increasing jitter dramatically. The RTSJ specifies an optional functionality, dynamic changing of thread priority, which would make it possible to change a thread's priority during execution. This would be an interesting way of providing a custom real-time scheduler that could adapt to system load, but in practice, no RTSJ certified real-time virtual machine implements dynamic priorities at this time. In general, a certain level of knowledge of the application and of RTSJ is required to correctly configure *Realtimeizeme*, and the same con be said regarding the configuration of real-time JVMs.

In general, it is better to give high priorities to threads that run for short periods and perform important tasks. General worker threads do not require real-time priorities, so one could avoid substituting them. A

proper mix of thread types and priorities can make an application more responsive and faster.

Future work plans include the implementation of more complex code transformations, such as new mappings between standard Java classes and RTSJ-safe classes. We also plan to focus more on memory and context management mechanisms instead of only using real-time thread priorities. Also necessary, static analysis to determine the implications of a substitution and to find conflict points is also of interest and exhaustive execution tests to ensure that reachable code is correct after substitutions. By adding these features to our tool, and with more extensive knowledge on the application to be transformed, our tool could be used as a first attempt to migrate legacy applications to the RTSJ framework and evaluate performance, or taken further, could be used as part of a build system to produce a separate real-time compatible solution using a single centralized code base.

VIII. REFERENCES

[1] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] S. Cherem and R. Rugina. Region Analysis and transformation for Java Programs. ISMM'04, 2004.

[3] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM 02*, pages 25-35, 2002.

[4] F. Pizlo, J. Fox, D. Holmes and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 101-110, 2004.

[5] A. Ferrari, D. Garbervestky, V. Braberman, P. Listingart and S. Yovine. JScoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *eTx'05*, pages 50-54.

[6] J. Etienne, J. Cordry and S. Bouzefrane. Applying the CBSE paradigm in the real time specification for Java. In *Proceedings of the 4th international Workshop on Java Technologies For Real-Time and Embedded Systems (JTRES '06)*, pages 218-226, 2006.

[7] A. Plsek, F. Loiret, P. Merle, L. Seinturier: A Component Framework for Java-Based Real-Time Embedded Systems. In *Proc. ACM/IFIP/USENIX 9th Int'l Middleware Conference* (*Middleware 2008)*, pages 124-143, 2008.

[8] A. Plsek, P. Merle, L. Seinturier: A Real-Time Java Component Model. In *ISORC'2008*, pages 281-288, 2008.

[9] J. Hu, S. Gorappa, J. A. Colmenares and R. Klefstad. Compadres: A Lightweight Component Middleware Framework for Composing Distributed, Real-Time, Embedded Systems with Real-Time Java. In *Proc. ACM/IFIP/USENIX 8th Int'l Middleware Conference (Middleware 2007)*, Vol. 4834, pages 41-59, 2007.

[10] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray and K. Reinholtz. Project Golden Gate: Towards Real-Time Java in Space Missions. In *ISORC'04*, pages 15-22, 2004.

[11] D. Garbervestsky, S. Yovine, V. Braberman, M. Rouaux, A. Taboada. On transforming Java-like programs into memory-predictable code. In *Proceedings of the 7th international Workshop on Java Technologies For Real-Time and Embedded Systems (JTRES '09)*, pages 140-149, 2009.

[12] Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron. Translation of Java to Real-Time Java using aspects. In *Proceedings of the International Workshop on Aspect-Oriented Programming and Separation of Concerns*, pages 25-30, 2001.

[13] K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[14] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, T. Zhao. Scoped types and aspects for real-time Java memory management. In *Real-Time Syst.*, Vol. 34, pages 1-44, 2007.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, 1997.

[16] J.-M. Dautelle. Fully time deterministic Java. In *AIAA Space 2007*, 2007.

[17] A. Rudys and D. S. Wallach. Enforcing Java Run-Time Properties Using Bytecode Rewriting. In *Proceedings of the International Symposium on Software Security*, 2002.

[18] U. W. Eisenecker and K. Czarnecki. *Generative Programming: Methods, Tools and Applications.* Addison-Wesley, 2000.

[19] E. Bruneton, R. Lenglet and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Proceedings of adaptable and extensible component systems*, 2002.

[20] E. G. Benowitz and A. F. Niessner. A Patterns Catalog for RTSJ Software Designs. *Lecture Notes in Computer Science*, 2003.

[21] P. Tyma. Why are we using Java again?, *Communications of the ACM*, vol. 41, n. 6, pages 38-42, 1998.

[22] Sun Microsystems, Inc. Java 2 Platform Se 5.0 - Package java.lang.instrument. Web pages at http://java.sun.com/j2se/1.5.0/docs/api/java/lang/instrument/package-summary.html

[23] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. In *IEEE Distributed Systems Online*, vol. 7, n. 11, 2006.

[24] M. Fowler. Module Assembly. In *IEEE Software*, vol. 21, n. 2, pages 65-67, 2004.

[25] G. K. Manacher. Production and stabihzation of real-time task schedules. *J. ACM 14'*3, pages 439-465, 1967.

[26] J. Stankovic and K. Ramamritham. *Tutorial on Hard Real-time Systems*. IEEE Computer Society Press, 1988.

[27] D. F. Bacon, P. Cheng and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the Thirtieth Annual ACM Symposium on the Principles of Programming Languages*, pages 285–294, 2003.

[28] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, 2009.

[29] bnd - Bundle Tool. http://www.aqute.biz/Code/Bnd

[30] M. Dahm, "Byte Code Engineering", Proceedings JIT'99, Springer, 1999.

[31] John A. Stankovic, Sang Hyuk Son, Jörgen Hansson: Misconceptions About Real-Time Databases. IEEE Computer (COMPUTER) 32(6):29-36 (1999)

[32] Krysztof Czarnecki, Ulrich Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley Professional, 2000, ISBN 0201309777