



Universidad  
Carlos III de Madrid

## **TESIS DOCTORAL**

# **CONTRIBUTIONS TO THE SAFE EXECUTION OF DYNAMIC COMPONENT-BASED REAL- TIME SYSTEMS**

**Autor:**

**Julio Ángel Cano Romero**

**Directora:**

**María de la Soledad García Valls**

**DEPARTAMENTO DE INGENIERÍA TELEMÁTICA**

**Leganés, Diciembre, 2012**



Autor: Julio Ángel Cano Romero

Directora: María de la Soledad García Valls

Firma del Tribunal Calificador:

Firma

Presidente:

Vocal:

Secretario:

Calificación:

Leganés, de de



# *Resumen*

Los sistemas de tiempo real han basado tradicionalmente su desarrollo en modelos altamente predecibles ya que estos requieren garantías temporales en su ejecución.

A lo largo de los años, la tecnología de tiempo real ha ido penetrando en diferentes campos de aplicación y ajustándose a paradigmas de desarrollo software más novedosos. Esto ha presentado y presenta en la actualidad un tremendo reto ya que estas aplicaciones suelen tener un alto grado de dinamismo, lo que entra en conflicto con la predictibilidad temporal y, en general la ejecución segura de los mismos.

Hoy en día se está realizando un gran esfuerzo en el desarrollo de sistemas cada vez más dinámicos que permitan adaptar su estructura en tiempo de ejecución para adaptarse a entornos que presentan condiciones cambiantes. La capacidad de soportar este tipo de dinamismo presenta ventajas destacables como permitir corregir fallos y añadir funcionalidad mediante actualizaciones en caliente, es decir, poder actualizarse sin necesidad de realizar paradas en su servicio, lo que podría implicar costes monetarios en muchos casos o pérdidas temporales de servicio.

Por otro lado, las técnicas de diseño y desarrollo basadas en componentes se han hecho muy populares y su aplicación a los sistemas de tiempo real gana terreno día a día. Uno de los principales motivos de ellos es que el uso de componentes permite simplificar el diseño del sistema, la reutilización de código e incluso la actualización del mismo mediante la sustitución de componentes.

En esta tesis se aborda el objetivo de proveer a los sistemas de tiempo real de cierto grado de dinamismo para poder reemplazar componentes de forma segura, que permita incorporar nuevas funcionalidades o corregir errores existentes. Para ello, en esta tesis se ha elaborado un marco de trabajo para dar soporte a reemplazos de componentes de forma segura, entendiendo como tal que el hecho de que no se produzcan ejecuciones incorrectas debido a la ejecución concurrente de múltiples tareas, así como el garantizar los tiempos de ejecución de cada tarea y acotar la duración temporal de los reemplazos. El marco de trabajo propuesto está basado, pues, en componentes de tiempo real, que tiene en cuenta los requisitos temporales en la ejecución de los componentes del sistema y de las tareas propias del marco que dan soporte a estos mecanismos de reemplazo. Este marco de trabajo incorpora un modelo genérico de componente con tareas de tiempo real, un modelo de reemplazo de componentes cuyos tiempos de ejecución son conocidos y limitados en tiempo y diferentes estrategias de aplicación de dicho modelo de reemplazo de componente. Las contribuciones propuestas integran el análisis de la planificabilidad de los componentes del sistema y de las tareas del marco de componentes para permitir establecer los parámetros de reserva de los recursos necesarios para las tareas del marco. Por último, se realiza una validación empírica en la que se comprueba experimentalmente la validez del modelo tanto de forma genérica como en un escenario específico y determinando también los recursos necesarios para su implementación.



# *Abstract*

Traditionally, real-time systems have based their design and execution on barely dynamic models to ensure, since design time, the temporal guarantees in the execution of their functionality.

Great effort is being applied nowadays to progressively develop more dynamic systems, with the target of changing during their execution and to adapt themselves to their environment. The capability to change and to reconfigure themselves represents remarkable advantages as the capability to fix errors and to add new functionality with on-line updates. This means to be able to be updated without needing to stop the service, that may imply monetary losses in many cases.

Design and development techniques based on components have become popular due to the use of components, which allows simplifying the system design, code reusability and updates through the substitution of components.

The target of this thesis work is to provide certain degree of dynamism to real-time systems allowing them to replace components, incorporating new functionality of fixing existing bugs. On that purpose, a component-based framework is proposed, as well as the corresponding task in charge of providing dynamism to the system. The main contribution is to provide a framework to allow safe component replacements. Safe meaning that incorrect executions of tasks are avoided even y multiple tasks are executing concurrently and making use of the same data. Also that temporal guarantees are provided for every task. This framework incorporates a generic component model with real-time threads, a components replacement model with execution times that are known and bounded, and different strategies to apply such component replacement model.

Some mechanisms to maintain a seamless and safe execution, regarding concurrency, before, during, and after applying the processes in charge of replacing running components are also described. Seamless execution means that components themselves do not perform the replacements, and safe means that temporal guarantees are provided and components are not affected in their execution. Part of these mechanisms are the system schedulability analysis and the framework tasks as well as reserving the needed resources for such scheduling to be correct.



# Table of Contents

---

<b>Chapter 1 Introduction and objectives.....</b>	<b>1</b>
1.1 Introduction.....	1
1.2 Motivation.....	2
1.3 Objectives.....	3
1.4 Overview of the solution contribution.....	4
1.5 Document outline.....	4
<b>Chapter 2 State of the art.....</b>	<b>7</b>
2.1 Real-time systems.....	7
2.1.1 Real-time tasks.....	8
2.1.2 Real-time systems scheduling.....	8
2.1.3 Real-time Programming Languages and Operating Systems.....	13
2.2 Resource management and QoS techniques.....	14
2.2.1 QoS-based resource management scheduling and architectures.....	15
2.3 Component and service-based frameworks for real-time systems.....	17
2.4 Dynamic systems and Components.....	23
2.4.1 Definitions and Concepts.....	23
2.4.2 Other Approaches for Dynamic Software Adaptation.....	25
2.5 OSGi Service Platform and Real-Time Specification for Java.....	27
2.5.1 Real-time Java and OSGi.....	31
2.6 Real-Time and Reconfigurable Frameworks and Middleware.....	38
2.7 Discussion.....	45
<b>Chapter 3 System model and framework overview.....</b>	<b>49</b>
3.1 Component-based system model.....	50
3.2 Replacement problem description and alternatives.....	59
<b>Chapter 4 Strategies and algorithms for dynamic real-time replacement of components.....</b>	<b>65</b>
4.1 Component model.....	67
4.2 Framework tasks and their operations.....	71
4.3 Strategies for component replacement.....	77
4.4 WCET calculation.....	82
4.5 Replacement acceptance test.....	83
4.6 Discussion.....	84
<b>Chapter 5 Framework specification.....</b>	<b>87</b>
5.1 Component implementation.....	88
5.2 Framework.....	91
5.2.1 Component description model.....	91
5.2.2 Replacement task.....	94
5.2.3 Component registry.....	96
5.2.4 Framework tasks.....	97
5.2.5 Calculation of WCET of component operations.....	103

---

5.3 OSGi integration.....	104
5.4 Predictability consideration on Java language.....	111
5.5 Summary.....	112
<b>Chapter 6 Validation.....</b>	<b>115</b>
6.1 Schedulability tests.....	116
6.2 Multimedia scenario implementation.....	122
6.3 Summary.....	125
<b>Chapter 7 Conclusions.....</b>	<b>127</b>
7.1 General conclusions.....	127
7.2 Future works.....	130
<b>Chapter 8 Conclusiones.....</b>	<b>131</b>
8.1 Conclusiones generales.....	131
8.2 Trabajos futuros.....	134
<b>Chapter 9 Bibliography.....</b>	<b>137</b>

---

## Index of figures

---

Figure 1: MARTE clock restrictions example.....	22
Figure 2: OSGi Bundle Life Cycle.....	29
Figure 3: Provided and required interfaces.....	51
Figure 4: Component types in the system.....	51
Figure 5: Synchronous invocation of operations.....	52
Figure 6: Shared use of passive component.....	53
Figure 7: Basic life cycle of a component.....	57
Figure 8: Component replacement overview.....	57
Figure 9: Load and replacement requests acceptance.....	58
Figure 10: Example of load and replacement of component.....	59
Figure 11: Erroneous replacement execution. Replacement task interrupted by component execution.....	60
Figure 12: Component Co2 replacement.....	61
Figure 13: Replacement task model.....	63
Figure 14: Example of active active component making use of passive components operations.....	69
Figure 15: Component life cycle.....	70
Figure 16: Framework tasks overview.....	71
Figure 17: Component Replacement Process.....	74
Figure 18: Component replacement.....	75
Figure 19: Independent time reservation for component replacement.....	78
Figure 20: Shared time reservation for component replacement.....	79
Figure 21: Acceptance test for replacement.....	84
Figure 22: Basic component configuration interface.....	89
Figure 23: Example of configuration and composition interfaces.....	90
Figure 24: Example of usage of the configuration interface by the replacement task.....	91
Figure 25: Component Description.....	92
Figure 26: RTThread class implementation.....	94
Figure 27: Method to enqueue replacement requests.....	94
Figure 28: Replacement thread execution.....	95
Figure 29: Framework architecture for the replacement task.....	96
Figure 30: Component Registry Model.....	96
Figure 31: Dependencies in reverse order.....	97
Figure 32: Component life cycle in the Java specification.....	98
Figure 33: Component instantiation process.....	99
Figure 34: Sequence diagram of component registering and instantiating.....	100
Figure 35: Sequence diagram of component unregistering.....	101
Figure 36: Replacement task collaborative diagram.....	102
Figure 37: Replacement process.....	102
Figure 38: Calculation of WCET of component operations.....	103
Figure 39: System architecture after the framework and OSGi integration.....	105
Figure 40: Component life cycle within OSGi platform.....	106
Figure 41: Imported and exported packages in bundle MANIFEST file.....	107
Figure 42: Service registration notification with Service Tracker.....	107
Figure 43: Service Tracker interface.....	108
Figure 44: Real-time component registration in framework.....	109

---

Figure 45: Code to intercept a new component addition.....	109
Figure 46: Code to intercept a new component removal.....	110
Figure 47: Code to replace a component.....	110
Figure 48: Schedulability tests.....	117
Figure 49: Example of schedulability tests application.....	117
Figure 50: Pessimistic replacement model. Percentage of failures regarding processor load and number of components.....	118
Figure 51: Selective replacement model. Percentage of failures regarding processor load and number of components.....	119
Figure 52: Acceptance percentage of tasks with and without CPU time reservation for component replacement .....	120
Figure 53: Rejection percentage in execution with & without processor reservation for component replacement task.....	121
Figure 54: Domestic scenario (a) and Video Streaming Server components (b).....	123
Figure 55: Multimedia scenario testing.....	124

## Index of tables

---

Table 1: OSGi Implementations.....	30
Table 2: Implementations of the RTSJ.....	34
Table 3: Component reconfiguration frameworks.....	47
Table 4: Temporal information of operations of passive component j.....	54
Table 5: Active component tasks operation invocations description.....	55
Table 6: Example of computation time of passive component operations.....	69
Table 7: Example of computation information of active component.....	69
Table 8: Example of component timing information in framework classes.....	93
Table 9: Generic tests replacement times.....	121
Table 10: Component update Acceptance Test Times.....	122
Table 11: Scheduling parameters of tasks (ms).....	123
Table 12: Multimedia scenario replacement times.....	125



# Chapter 1

## Introduction and objectives

---

### ***1.1 Introduction***

Methods and techniques for the development of traditional real-time systems have had the goal of identifying and avoiding the sources of uncertainty with the aim of constructing systems with time-predictable execution targeted at the safety critical real-time domain. Over the years, systems have evolved with the market demands and with the increased computational power of the new hardware bringing in new applications with softer temporal requirements. Techniques have been adapted to fit to the general purpose computing environments where timely operation is a clear added value. As a consequence, real-time techniques have been progressively merged with other paradigms (e.g., distribution and middleware) that introduce a number of challenges as, for instance, uncertainty and dynamic behavior.

Providing real-time execution guarantees in the presence of dynamic behavior requires that some bounds be imposed in order to achieve feasible solutions. Most modern distributed systems benefit from a certain degree of dynamism. Currently, they are mostly built with service oriented paradigms and/or component technology providing loosely coupled software architectures where the entities (components or services) encapsulate the functionality pieces that can be, if needed, replaced by other entities. The goal is that the replacement of an entity can be done seamlessly to the rest of the system. Whereas this is a conceptual point in favor, in practice, it becomes difficult to implement especially if the replacement must also be seamless in the temporal domain, e.g., time-bounded in order not to cause any deadline miss. In the last decades, solutions have been provided for the development of real-time systems based on component as a baseline to increase productivity and liability in development.

Recently, contributions are appearing to support some degree of controllable flexibility in

these systems. Component models are being adapted, trying to minimize conflicts that appear mixing real-time support and dynamism. Timely run-time software replacement techniques are, then, a corner stone for reconciling real-time systems development and dynamic behavior.

Typically, real-time systems do not present dynamic behavior since it deeply challenges predictability and timeliness. In this doctoral work, the main challenges for this problem are identified and a framework to enable components to be replaced at run-time without temporal interference is provided. This way failures in replacements are avoided. We consider that the dynamic behavior of a system refers to the capability to install/load new components in the platform, replace components, uninstall/unload components, and modify their connections at run-time. These capabilities allow the system to adapt to run-time changes and to prevent or fix execution failures.

There are systems with real-time requirements that can be benefited with this dynamism. Enabling the capability to load, unload or replace components on-line can save the costs associated to stopping the platform. Multimedia applications can make use of these techniques to implement functional replacements and reconfigurations on-line without interrupting the provided service.

A component model and the corresponding algorithms to enable that components can be loaded and replaced at run-time, without interfering in the execution deadlines of running components are described. Simulations of our replacement model are provided that validate the presented ideas.

The framework provided in this work supports run-time replacement of components with real-time requirements. We also include a timely replacement scheme that avoids blocking the execution of components. This framework relies on the existence of *management tasks* for *installing/loading*, *uninstalling/unloading*, and *replacing* components.

## 1.2 Motivation

Traditionally, real-time systems have been designed to be unmodified at run-time in order to maintain the system predictability. Over the years, the new software paradigms and the presence of dynamism in the emerging applications, has proved that some dynamism can be provided to real-time systems (mainly soft real-time systems as multimedia applications [1]) and that costs can be reduced by the use of some techniques as component-based systems (see [2]).

Software architectures based in components allow dealing with the complexity, scalability and the evolutions of designed applications. The design of systems is simplified. The component based systems may also evolve during their execution. The interactions between components may also change on-line, to adapt their execution to the environment or to correct execution bugs.

The on-line replacement of components represents a high degree of dynamism in a component-based framework as it implies including the load, unload and reconfiguration of components or their connections.

Providing safe on-line replacement of components in a real-time system represents a very

high level of dynamism for a real-time environment, where deadlines must not be missed and uncertainties may generate scheduling problems.

### **1.3 Objectives**

This thesis work provides contributions to enable predictable execution in component-based real-time systems that include some level of controlled dynamism. A framework is proposed that incorporates concepts from dynamic systems. Dynamic systems allow to modify their architecture at run-time by loading, unloading, reconfiguring or replacing components. Real-time applications that should not be paused or stopped may increase their reliability incorporating this dynamism.

This work shows that time-bounded dynamism can be achievable in real-time systems. The system is still predictable and safe while, at the same time, it can be updated or reconfigured at run-time obtaining flexibility and other benefits of component-based systems.

The main target of this thesis is, then, to provide a framework that contains a component replacement model for a real-time component-based system that is time-bounded and guarantees safe execution. Specific targets to accomplish this are:

- Study and analysis of current solutions to support dynamism in real-time systems, and their capability to replace components at run-time. Also, their capability to maintain the temporal guarantees of the system during the reconfiguration or replacement of components at run-time.
- Proposal of a component replacement model. The characteristics of the replacement model are:
  - A generic component model is used with real-time characterization.
  - Enough resources are reserved for replacements to do not affect the correct scheduling of the real-time components in the system. Running components are not stopped to be replaced.
  - The running components meet their deadlines during their normal operation and also in the event of a replacement.
- Specification of a framework able to perform the correct scheduling of the component and the needed tasks, as the loading, unloading and replacing tasks. The framework also specifies the method to calculate the WCET of components on-line and the acceptance test to load or replace components.
- Integration of the implemented framework in a dynamic platform to prove that a dynamic platform can incorporate real-time characteristics. At the same time it proves that dynamism can be provided to a real-time framework, with a variable number of components and run-time replacements of such components.
- Validation of the proposed framework. Set of different scenarios for execution of simulations to prove the capabilities that are proposed in this work. These tests take into

account the execution time of components, the time required to perform replacements, and the interference generated by such replacements.

### **1.4 Overview of the solution contribution**

Temporal information of the execution of component threads is provided to the system when the component is loaded. Besides, information about the number of connections of a component and the time required to copy its internal state are used.

The number of components running on the platform and their characteristics is previously unknown. Run-time execution in a dynamic component-based system requires to integrate safety mechanisms as, for example, avoiding state corruption and interruption of the component's execution. Beside reserving resources for the main mission tasks, i.e., those executing the components code, additional resources are reserved for the framework tasks. Needed resources are calculated based on the information about the temporal information of components and the required time to execute them.

Framework tasks enable the replacement of components in the system. This allows the system execution to be reliable and to avoid component threads to miss their deadlines all through its operation lifetime, i.e., through their normal operation and reconfiguration stages.

Restrictions and specifications are provided for the safe execution of the framework tasks. Different strategies are provided for the schedulability of component replacements. The first one enables replacing a component every time the component is executed. The other one is adjustable according to the application needs.

### **1.5 Document outline**

This document is structured as follows:

- Chapter 2 : Contains basic terminology and concepts that are the baseline for this work. Also, related work on real-time systems, component technology, replacement and reconfiguration strategies, and service oriented reconfiguration middleware are presented. An analysis of their missing points and gaps is also given.
- Chapter 3 : General aspects of the framework and its basic elements are described here. The real-time task model, the component model and expected replacement task characteristics are also described. The specific problems regarding the component replacement in real-time environments and difficulties to perform such component replacements are described. This chapter described the environment on which the solution proposed in this work is applied.
- Chapter 4 : A detailed description of the framework is provided. This framework includes all the required elements to provide a safe component replacement with real-time characteristics. The component model used is also detailed. Additional operations included in the framework are detailed, as the calculation of the WCET (Worst Case Execution Time) of the components at run-time, and the acceptance test for the load and

replacement of new components.

- Chapter 5 : A specification of the described framework is provided in this chapter. This specification is implemented in Java. The component model, WCET calculation, acceptance test, and the rest of processes are implemented following the description proposed in the previous chapter.
- The implemented framework is integrated in a well-known dynamic platform named OSGi. This platform is unaware of real-time components and their characteristics. With the inclusion of the framework in OSGi both platforms obtain get benefited. The real-time framework is provided with new capabilities as the load and unload of code. The OSGi platform is provided with support for real-time components. This support is provided as transparently as possible, using the interfaces already provided by the platform.
- Chapter 6 :Ideas proposed in this work are validated in this chapter. The effects on the schedulability of any generic system are tested after including the capability to replace real-time components at run-time. The usability and overhead generated by the use of the proposed infrastructure are tested. A specific scenario for multimedia applications, representing soft real-time systems is also created and tested. The validity of the proposal is shown.
- Chapter 7 : General conclusions of the presented work are provided here. The main contributions are remarked. Future work are also suggested.



# Chapter 2

## State of the art

---

### 2.1 Real-time systems

Real-time systems are those where time constraints in the execution are essential for them to function correctly. From [5]: “A system is called a **real-time system**, when we need quantitative expression of time (i.e., real time) to describe the behaviour of the system”. And also from [6]: real-time systems are subject to “operational deadlines from event to system response”. There are different kinds of real-time systems depending on the strictness of the time constraints.

One of the main characteristics of real-time systems is that their correctness depends on both functional and temporal aspects [7]. Timing correctness requirements comes from the impact of a real-time system upon the real world. These 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, tasks can be periodic, sporadic or aperiodic. Periodic tasks are invoked or activated 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 [8], [9]. 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.

Deadlines can be relative to an event or absolute, indicating an exact 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 [10]. In hard real-time systems, all deadlines must be strictly enforced to avoid safety problems (e.g., weapon systems, nuclear power plants, automated transport systems) and to ensure system correctness [8] and predictability [11]. Firm

real-time systems are those in which results produced as soon as the deadline expires become useless for the application, but consequences may not be severe and system may still function correctly. 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.

### 2.1.1 Real-time tasks

A real-time application is formed by real-time tasks. These tasks usually execute periodically, analyzing sensors and applying actions according to the values of these sensors. Beside *periodic* tasks, *aperiodic* tasks are also considered. While periodic tasks are executed using a fixed period, *aperiodic* tasks are executed depending on events occurred in the system. Finally, *sporadic* tasks execute periodically but with variable execution periods previously not known. Then, a minimum period is considered in the system for schedulability reasons. This time is named Minimum Interarrival Time (MIT).

Usually a real-time system is described as a set of tasks. Real-time tasks are characterized as follows:

$$\tau_i = (C_i, T_i, D_i, P_i) \quad (1)$$

Where:

- $C_i$  is the Worst Case Execution Time (WCET), the maximum length of time the task could take to execute.
- $T_i$  is the execution period of the task. A new job (task instance) is launched every fixed time infinitely.
- $D_i$  is the relative deadline, the time given to the task to finish its execution after every invocation. This time should not be greater than the period  $D_i \leq T_i$ .
- $P_i$  is the task priority. Every task can have a different priority or different tasks can have the same priority. Priorities can also change along the time.

### 2.1.2 Real-time systems scheduling

Scheduling is the process of determining where and when each task is executed [10]. This scheduling determines the correct or incorrect execution of the real-time tasks. One of the main characteristics of real-time scheduling is that the main target is not to minimize the execution time or maximize the CPU usage to satisfy the tasks constraints.

A simple execution test is not enough as it does not assure the absence of fails. A real-time scheduling theory [12] arises to solve the problem of determining a correct scheduling for a real-time task set.

A scheduling algorithm is defined as a set of rules that determines the execution of tasks along the life of the system so that task restrictions are satisfied. This algorithm determines the moment at which a task is selected for execution and how long it executes.

To assure that the scheduling algorithm can satisfy task restrictions some tests have to be passed previously. These tests tend to be complex so some simplifications are proposed. These simplifications reduce the complexity and execution time in exchange of precision. Schedulability tests are classified then as *exact*, *sufficient* and *necessary* [13]. Exact tests always give a positive answer if the task set is schedulable and negative on the other case. These tests usually take so long that can not be applied on-line. Tests proposed as sufficient are more computationally inexpensive. A positive answer indicates that the task set is schedulable, but a negative answer does not necessarily imply that the task set is not schedulable. Some task sets that are actually schedulable will be rejected. On the other hand necessary tests indicate when a task set is not schedulable, but some positive answers might not be actually schedulable.

Schedulability tests can also be classified depending on the way to determine the schedulability of the task set. Some of them are based on the CPU utilization factor ( $U_i = \frac{C_i}{T_i}$ ). This represents the CPU load of every task. The other possibility is to use the response time. This is the time that takes a task to finish its execution since it is scheduled to execute. The response time of any task should be less than or equal to its deadline.

Other schedulability tests are based on 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 [14], [15].

Scheduling algorithms can also be classified as static or dynamic. Static algorithms determine all the characteristics of the tasks off-line and no changes are made during the execution of the system. Dynamic scheduling algorithms allow to change task properties, like priorities, while the system is running. Dynamic algorithms can also provide preemption, where running tasks can be deallocated of the CPU to allow other tasks to start their execution, depending on their priorities.

The ultimate goal of a real-time system is to achieve time predictability. Determinism and predictability are closely related, because one depends on 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) [16]. 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. One of them is *latency*: it is the time between an event and a system response to that event. The other one is *jitter*: in the context of real-time systems, detects unsteadiness in system latency. Depending on the application, having a normalized latency and a small jitter may be more important than deadline enforcement.

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 [17].

Beside hard and soft real-time tasks non real-time tasks can be considered in the system. These tasks do not have a deadline and their execution is considered as best-effort. The scheduler will try to execute them using free CPU time and finish them as soon as possible. Their execution is considered to not interfere in real-time tasks execution given that any real-time task has a higher priority.

### ***Static and dynamic scheduling***

Static scheduling means that tasks execution sequence is created off-line. Then the scheduler follows this sequence to continuously launch executing tasks without making changes in the sequences. This scheduler is simple and has no computational overload in the system [18]. Although this method is simple and predictable it is also static and hard to maintain because it cannot be changed [19]. This method also has the inconvenience of the difficulty to add aperiodic tasks to the scheduling plan [20].

Dynamic scheduling solves some of the problems of the static scheduling planning. A priority is assigned to every task. The scheduler chooses the task with a higher priority at every moment to be scheduled for execution. Priorities can be fixed or change along the application time. The main dynamic scheduling algorithms are described by Liu and Layland in [21]. One of these algorithms is Rate Monotonic Scheduling (RMS) based on the assignment of fixed priorities depending on the period of every task. The other one is the Earliest Deadline First (EDF) where the higher priority is dynamically assigned to the task with the closest deadline at the moment. These algorithms are considered to be optimum in their kind, so if any other algorithm is able to find a feasible scheduling plan for a task set then these algorithms will also find it [22].

### ***Rate Monotonic Scheduling (RMS)***

To be able to apply RMS to a task set, some suppositions are made a priori. According to [21], these suppositions are the following:

- Only critical real-time tasks are considered.
- Deadline of every task is the same as the period ( $D_i = T_i$ ).
- There are no dependencies between tasks, nor shared resources.
- Execution time of every task is constant during the execution of the application.
- Task's execution is not halted.
- Aperiodic tasks are not critical, so no deadline is considered for them.
- The scheduler is based on the assigned priorities and with preemption.

Once these restrictions are accomplished then priorities are monotonically assigned to tasks according to their period. Then:

$$\forall \tau_i, \tau_j: T_i > T_j, P_i < P_j \quad (2)$$

A larger period means a lower priority assigned to the task. The schedulability test proposed by Liu & Layland is based on the CPU utilization factor of the task set. This factor is defined as:

$$U = \sum_{i=1}^N \frac{C_i}{T_i} \quad (3)$$

$N$  indicates the total number of tasks in the tasks set. Liu & Layland show a theorem demonstrating that if given any task set with assigned priorities then the set is schedulable, with any possible phase shifts, if:

$$U = \sum_{i=1}^N \frac{C_i}{T_i} \leq N(2^{\frac{1}{N}} - 1) \quad (4)$$

According to this theorem, if utilization factor is less than a given threshold then the task set is schedulable. Increasing the number of tasks, it can be found that under a given utilization factor any task set is schedulable.

$$\lim_{N \rightarrow \infty} N(2^{\frac{1}{N}} - 1) = \ln 2 \approx 0.69 \quad (5)$$

Equation (5) shows that any task set with a CPU utilization factor lesser than approximately 0.69 is schedulable. The condition imposed by this test is sufficient but not necessary. Many task sets with a greater utilization factor could also be schedulable under the RMS algorithm.

Lehoczky [23] proposes a technique to apply an exact schedulability analysis to the task set. This technique is based on the calculation of the required workload that has to be completed between the activation time of the tasks and their respective deadline. If this workload can be completed in time then the task set is schedulable. The time to complete the workload of a task depends on the interference generated by the tasks with higher priority. Tasks with higher priority have a shorter time between activations, so they will be activated more times interfering with tasks of lower priority. So given a task, and those with a priority higher or equal to the first one, the workload required to be completed between time  $0$  and time  $t$  is given by:

$$W_i(t) = \sum_{j=1}^i C_j \lceil \frac{t}{T_j} \rceil \quad (6)$$

This workload has to be lower or equal to the time  $t$  to complete it. In that case, the task set is schedulable. For schedulability analysis, only times  $t$  equal to periods of tasks ( $T_i$ ) are used. Based on this work, the Worst Case Response Time (WCRT) of a task can also be calculated. The minimum time to accomplish the task execution is  $C_i$ , and the interference of higher priority tasks is added, giving Equation (7).

$$w = C_i + \sum_{j < i} C_j \lceil \frac{w}{T_j} \rceil \quad (7)$$

The addition of higher priority tasks increases the time to complete task  $i$ . The method to find the real WCRT is to apply the Equation (7) iteratively until the time considered is equal to the time required to accomplish such workload. If the iteration does not converge (when results are greater than the deadline) then the task set is not schedulable. The way to apply iteratively the equation is as follows:

$$w^0 = C_i$$

$$w^{n+1} = C_i + \sum_{j < i} C_j \lceil \frac{w^n}{T_j} \rceil \quad (8)$$

When  $w^{n+1} = w^n$  then the WCRT is found and the task set is schedulable.

### ***Earliest Deadline First (EDF)***

Making use of the same restrictions that in RMS, Liu & Layland propose a scheduling algorithm with dynamic priorities. In this case, priorities depend on how close are deadlines from actual time. This makes priorities change continuously according to:

$$\forall \tau_i, \tau_j : d_i > d_j, P_i < P_j \quad (9)$$

Where  $d_i$  and  $d_j$  are the absolute deadline of the tasks at the moment the test is applied. The task with the highest priority is selected for execution and the actually executing task is preempted. In the same work it is demonstrated that using EDF the full utilization of the CPU can reach 100%.

$$\sum_{i=1}^N \frac{C_i}{T_i} \leq 1 \quad (10)$$

To be able to reach higher CPU utilization factors than in RMS priorities have to be recalculated continuously introducing an overload in the system. The complexity of the priorities sorting algorithm is  $O(n \log n)$ . This overload is much higher than the needed by RMS but depending on the use it can compensate on the CPU utilization factor.

### ***Distributed real-time systems***

A distributed real-time system contains several interconnected nodes using communication networks. These nodes contain concurrent threads that exchange messages through the communication networks. Threads in the same node can also share data using traditional synchronization mechanisms.

In statically distributed systems a schedulability analysis that determines the correct assignation of resources (CPU and network) can be applied. Priorities can be assigned then to threads and network utilization.

A distributed system is modeled as a set of transactions activated by the arrival of one or more external events. Activities are carried out by the system when transactions are activated. These activities can generate new events that will activate new transactions.

Distributed systems can be developed using the communication services provided by the operating system. But the complexity of the implementation can increase exponentially if the number of nodes is increased. The use of different operating systems also increases this complexity.

This complexity can be simplified through the use of middleware [24]. This is an intermediate software layer that simplifies the management and programming of distributed systems. Middleware can actuate at different levels in the system.

The first middleware level is the *communication level*. This level provides abstraction of low level communication and distribution. This usually represents the *addressing* of the software elements to be invoked, *marshaling* of the data or invocations to be transferred through the network, *dispatching* this information to the corresponding resource and *transporting* it through the communication links.

The second middleware level is the *component level*. This is usually a formal model where functionality is encapsulated in software elements. These elements can be reusable and transparently distributed over the network. The middleware provides transparent communications to the distributed components.

According to the mechanism used for the distribution in the distributed service middleware can also be classified as based on RPC (Remote Procedure Call), Distributed Object Model (DOM) and Message Oriented Model (MOM). The alternative is the direct usage of the communications services by the programmer. The direct usage of communication services provided by the operating systems may result complex and error prone.

### 2.1.3 Real-time Programming Languages and Operating Systems

Using traditional technologies and methodologies in real-time development is costly and difficult [25]. 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 unpredictable execution times.
- Extending existing languages.
- Being constructed jointly with an operating system, offering a run-time execution environment.

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 expressed 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 [26].

Assembly, procedural and object-oriented languages are the most common general-purposed languages used for developing real-time systems. Despite the lack of most of 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 [27]. 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 ESTEREL, among others [27], [28].

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 [29].

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 [30]. Some basic paradigms found in traditional operating systems cannot be applied to RTOS's. 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 with some characteristics like cost enforcement.

The IEEE Portable Operating System Interface for Computer Environments (POSIX 1003.1b) [31] 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 memory, execution scheduling, timers, interprocess communication (IPC), real-time files and real-time threads. Other basic requirements [32] 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 Resource management and QoS techniques***

For the platform to be able to provide a Quality of Service (QoS), even if it is done for hard or soft real-time systems, a Resource Manager is necessary to manage available resources and assign them to the running processes in the platform. There are many QoS and Resource Management (RM) architectures designed to manage resources in component based platforms. Here are described some of the most important architectures that influence this work.

Quality of Service-based resource management is the trade-off of resources assigned to tasks for the quality of the delivered output by such tasks. The requirements of an application are not explicitly expressed as part of the functional requirements of the application. Some of the QoS

requirements are security, performance, availability, etc. Specifically QoS is also used to represent the capability of the application to satisfy some performance related constraints. This is the case of multimedia and real-time applications.

QoS is also applied to the non-functional requirements of services provided by the application. These contractual requirements impose restrictions on the service and the resources required by the implementation. The contractual relationship between the service provider and the client is given by the Service Level Agreement (SLA). In this SLA, objectives are described as a set of Service Level Objective.

To achieve its objectives the application or the platform has to manage the available resources in the platform. A contractual relationship is established between the platform and the application or service provider. The platform has to assure that the resources required by the application are available when needed.

Resource accounting is the mechanism used by the platform to manage the resource usage by the applications. Due to the limited availability of resources an acceptance test has to be passed by services or applications installed in the platform to assure that enough resources are available for them to provided the quality levels in their SLA.

### 2.2.1 QoS-based resource management scheduling and architectures

There are several techniques available to provide QoS through the resource management and assignation. In soft real-time systems like multimedia applications two of the most used are Constant Bandwidth Server and Budget Management Scheduling. In this section these scheduling models are described here and architectures based on them.

#### *CBS (Constant Bandwidth Server)*

CBS is a scheduling algorithm based on EDF to provide temporal isolation between hard and soft real-time tasks [33], [34]. It mainly manages aperiodic tasks without overloading the use of the CPU and preserving the CPU bandwidth for real-time periodic tasks.

A time budget is assigned to every task ( $Q_s$ ) for every time slot  $T_s$ . A server deadline is generated to a time  $T_s$ . The task budget is recharged after every time slot. The bandwidth server is denoted as  $U_s = Q_s/T_s$ . The use of EDF also enables the capability to reach a full processor utilization:

$$\sum_i \frac{Q_{(i)}}{T_{(i)}} = U \leq 1 \quad (11)$$

This resource reservation model is widely used in many soft real-time multimedia systems like [33] and [35].

Enhancements are proposed by García-Valls et al. in [36]. A dual band priority assignment algorithm is applied to a constant bandwidth server to enable safe overruns of greedy tasks when it does not interfere in the normal execution of non-greedy tasks.

### ***Budget management scheduling***

There are soft real-time applications where execution times of tasks are variable. Multimedia applications require different processing times for every frame or picture. A method is applied where budgets of time are assigned to optimize the processor time usage instead of strict deadlines. Strict deadlines halt the execution of tasks and this can provoke that work applied in a video frame be lost. Assigning flexible budgets of time where tasks executions are allowed to overrun such budget or to start its execution if free processor time is available [37].

Budget management scheduling is also useful when tasks' execution times vary along the time due to configuration needs. Budget time can be reserved for reconfiguration purposes [1].

### ***HOLA-QoS architecture for QoS resource managers***

García-Valls et al. proposes a mode based QoS resource managed architecture named HOLA-QoS aimed at consumer electronic embedded systems[38]. This architecture contains several layers to describe and manage the QoS of the applications in the platform. The layers in HOLA-QoS are the following:

- QoS Management layer (QSM): handles QoS at application level.
- Quality Level Control layer (QLC): handles QoS configurations for every application.
- Budget Control layer (BDC): manages budgets at task level.
- Run Time Control layer (RTC): accounts and enforces systems resources at task level, directly dealing with the Real-Time OS or the hardware platform.

Every layer is designed homogeneously so that all of them have the same components:

- Admission: Performing admission control to determine if a request can be satisfied by the system.
- Settings: To perform the required operations to change the current system configuration. The system configuration is modeled here as a tuple of  $(A_x, Q_y)$  where  $A_x$  is one of the applications of the system and  $Q_y$  is the  $y^{\text{th}}$  quality level for such application.
- Monitoring: Monitors the resources usage required by the application or its tasks to provide it for upper layers.
- Alarm Handler: To detect abnormal tasks or applications behaviors as execution overruns.
- Interface with External Actor: Enables other actors to interact with any of the layers in this architecture, like the user or different layers.

The RTC layer also contains a component that allows to encapsulate the RTOS functionality simplifying its use and adjusting it to the architecture needs.

QoS is characterized at every level of the architecture. Every application can have many QoS levels, with several configurations each one. QoS levels are composed of cluster configurations which describe different sets of budgets and task configurations. Every element of this QoS

characterization will be used by its corresponding level of the architecture.

### ***AQuoSA***

The AQuoSA framework implements an architecture for quality of service (QoS) control of time-sensitive applications in multi-programmed embedded systems [39]. This architecture is based in the use of CBS and EDF.

AQuoSA provides dynamic resource reservation for tasks based on the prediction of needed resources of executing tasks. The basic elements of this architecture are a general supervisor and task controller containing a predictor and a feedback controller:

- *Feedback controller*: Calculates the error in previous predictions on the resource reservation for an specific task. It evaluates the worst-case effects caused by the uncertainty of  $C_i$ .
- *Predictor*: It is tightly associated to the task or application as it is based on the stochastic properties of the process. For example, in case of a MPEG4 video parsing task it can be based on the use of pixel count, byte count, macroblock count, etc. It tries to maintain a prediction error within previously fixed limits.
- *Supervisor*: It is in charge of ensuring that each task receives its guaranteed amount of bandwidth, even when the cumulative requests of bandwidth from different controllers exceed the limit of CPU utilization.

### **2.3 Component and service-based frameworks for real-time systems**

Component-based frameworks have always been used to reduce the complexity in the development of large systems. Functionalities are decoupled, easier to implement and the reusability increased. In real-time systems, complex programming interfaces are used. The application of component-based frameworks in real-time systems again simplifies their implementation increasing also the productivity in the development of large real-time systems.

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.

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) [40][41], Web services and JavaBeans [42].

Even though SOA (Service Oriented Architecture) and component-based are mainly 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 [43].

Here are described the main characteristics of a software component:

- It is a composition unit. Applications are built by composing components in a predicted and known way.
- The way components interact with other components is specified by their interface. Service components described here make use of interfaces that represent a contract between the client and server components. The server provides an interface that can be used and the client requires that interface.
- Beside the component interfaces the component can define more requirements to operate. These requirements can be related to the platform and resources needed to operate correctly or according to a Quality of Service Level.
- Once the component requirements are checked then the component can be deployed independently to other components. Beside that, a component replacement should be feasible without problems while the new instance provides the same functionality.
- Users have no need to know the internals of the component to manage it. The component is considered to be an opaque element.

Component systems can also be generated by composition. Composition of components 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 [44].

For a component to be used in the platform and by a user without requiring knowledge about its internal implementation details, it has to provide information to the platform and the user. This information is named *metadata*, and allows the platform to handle the component. Metadata is used to install the component, set it up, connect it to other components, etc.

Metadata can be split in two kinds:

- Functional metadata: Describes information about the functionality of the component. This is mainly represented by the provided and required interfaces of the component. Configuration information can also be included here as it belongs to the functional capabilities of the component.
- Implementation specific metadata: This information is specific of the implementation of the component. It can be used by the platform to choose the specific component implementation to be deployed at every moment. Component instantiation information or required platform resources are examples of implementation metadata.

### ***Component Models***

One of the most representative component models is CORBA Component model (CCM) [45]. Component interfaces are described using CORBA's Interface Definition Language [46]. From the interface specification two elements are generated in the target programming language: a stub and a skeleton. The skeleton is the structure code to be filled with the code implementing the component. The stub is to be used by the client to access transparently the remote implementation. Components are instantiated and managed by component containers. Object Request Brokers (ORB) are in charge of making these instances remotely available. Communication through events is also available between components.

CORBA has evolved to support many different programming languages, operating systems and communication networks. Supported languages are C, C++, Java, Ada, Python among others. The abstraction of the operating system and network is done through the use of the IIOP protocol (Internet Inter-ORB Protocol). Every invocation is encapsulated using this standardized protocol.

Other component models commonly referenced are Fractal [47], a hierarchically-structured component model, and Koala [48], 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 [49], Koala is restricted to switching between statically-defined components.

Recent component models introduce new features in order to provide more flexibility. For instance, iPOJO [50] is a runtime service-oriented component model which can be used to develop applications over the OSGi service platform. iPOJO injects POJOs (Plain Old Java Object) at runtime, through the management of service providing and dependencies. iPOJO provides component containers which manage all service interaction and allows adding non-functional properties, such as persistence, security and autonomic management. Component dependencies and non-functional properties are managed 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) [51] 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. MADAM provides an adaptation manager and a middleware framework for runtime adaptation.

### ***Component model for real-time applications***

The main target in a real-time application is its predictability. The construction of a real-time application through the use of components has to ensure its predictability. Traditionally all the timing requirements specified for the application activities have to be certified at design time.

To be able to certify that these timing requirements are met the components that compose the application have to include enough information for the timing analysis to be applied. Component metadata for a real-time application must include, then, enough information for timing analysis.

This information includes time characteristics, concurrency and synchronization needs of component activities. Although this information is available the internal behavior of the component is still opaque and there is no need to access or modify it.

Another requirement for a real-time component is that all the services defined in every component, as well as services provided by the platform, are bounded in time. This means that execution times are known and predictable.

This determines that a real-time component must provide real-time metadata beside the default metadata provided by any other component. This metadata is used by the application designer and by the platform to analyze the timing requirements and to determine if they are met. The complete real-time model of the application should be acquired based on the real-time metadata of the integrating components.

There are some characteristics that have to be taken into account then obtaining the real-time model of an application based on real-time components:

- The temporal behavior of a component not only depends on its own code. It also depends on the temporal behavior of the services used by the component to implement its functionality. This means that beside the provided metadata to describe the services provided, and required, by the component more information has to be provided about the collaboration and synchronization mechanisms that have to be applied between components.
- Platform functionality timing also affects the timing behavior of the component. Real-time component model may also have to make reference to services and real-time characteristics of the supporting platform.
- Available resources in the system also determine the temporal behavior of a component (processor, memory, network bandwidth, etc.).

This determines that the temporal behavior of a component cannot be determined on its own, but in a context given by the rest of components of the application and the platform characteristics at run-time.

### ***MARTE***

MARTE (Modeling and Analysis of Real-Time and Embedded systems) is a UML profile created by OMG. It is designed to provide capabilities to UML for model-driven development of Real Time and Embedded systems.

The MARTE profile [52] covers several steps in the model-driven development like specification, design and verification stages. It provides a foundation for the description of real-time and embedded systems. These basic concepts are refined to support modeling and analyzing concerns. No development process is enforced by MARTE. Only support is provided by the annotation of model elements with information required to perform analysis of the developed system.

Hardware and software can be modeled using the MARTE profile. It also enables

interoperability between different development tools specialized in different development stages. For the support of every different area and development stage the Extension Units are defined. These Extension Units describe different elements as well as applicable properties.

As an example of Extension Units are SRM and HRM for Software Resources Modeling and Hardware Resource Modeling respectively. Software synchronization elements are represented in the SRM, like pipes, *mutex* elements, etc. Different kinds of processors and memories are represented in the HRM. The Schedulability Analysis Modeling units allow to parametrize elements to be able to apply schedulability analysis. Elements used in this unit are resources, workloads, timing properties, etc.

There are works oriented to provide higher level component design capabilities based on MARTE like [53]. This work introduces a component model based on MARTE. It provides some dynamic capabilities to the final code with the use of mode changes. Due to the model-driven nature of MARTE model transformations and automatic code generation can be applied. Other works like [54] are also based on the MARTE profile and mode changes to provide adaptability for dynamic power management.

By now MARTE extensions are not designed to provide more dynamicity to the system, enabling the capability to load, unload and replace components at run-time and, at the same time, maintaining the quality of service of the system.

### **Real-time parameters for a component in MARTE**

MARTE provides an extension to the profile to be able to represent time in UML models [55] [52]. This extension is based in SPT [52], the previous UML extension to model time. Whereas SPT is only capable to model physical time the new MARTE extension provides the capability to model physical and also logical time.

MARTE profile creates the *instant* concept in a time base relation where these instants can be partially ordered. Time bases can also be related representing a complete multithreaded or distributed application.

Beside instants, the *Clock* element is created. It makes reference to a *TimeBase* and so to the instants of this timebase. Dense or discrete clocks can be described depending on the associated kind of unit to it. *Units* are described in the NFP (Non-functional property) domain view of the MARTE profile. Specializations are created in this domain view of the *TimeUnitKind* for different time resolutions (second, millisecond, etc.). Clocks themselves are specialized in Chronometric and logical clocks. Chronometric clocks make reference to physical time and includes some non functional properties like stability, skew, etc. A logical clock is based on events. A tick occurs for every clock event. So the time is counted in the number of ticks in a logical clock.

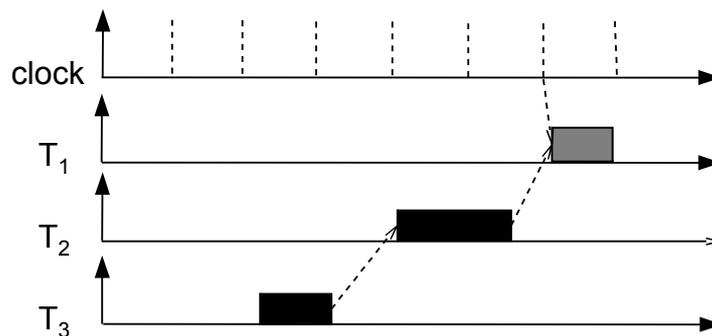
Based on clocks MARTE is able to specify not only instants but also duration times. A *TimeValue* is used to define a moment in time based on a clock, its units and an specific instant. A *DurationValue* allows defining a time interval making reference to two different instant values.

Once basic elements of time are defined they can be associated with other elements of the model. A *TimedElement* is a model element which is associated to one or more clocks in the

model. A *TimedEvent* is also included to represent events in the system with their respective associated time. This timed events also allow to provide semantic meaning to events and elements in the system.

### Example

A complete library is provided with elements to be used in UML models. Clocks and clock constraints are included. Clock constraints provide semantic relationships between different clocks in the system, like *isPeriodicOn*, *alternatesWith*, etc. This is the Clock Constraints Specification Language [56]. Clock constraints semantics provided are useful to specify timing properties and requirements of tasks in the system.



**Figure 1:** MARTE clock restrictions example

Figure 1 shows an example of graphical representation of different tasks and their timing relationship. The `clock` signal is described with the `discretizedBy` constraint as `clock = idealClock discretizedBy  $\omega$`  where  $\omega$  is a discrete period time. The *idealClock* is provided by the MARTE time library as the *ideal* physical time.  $P$  indicates the time between instants of occurrences of `clock`. Another dependency denoted in the figure is that  $T_3$  precedes  $T_2$ . This can be expressed with the *precedes* constraint as in  $T_3$  *precedes*  $T_2$  or  $T_3$  *strictly precedes*  $T_2$ . Being  $T_3$  and  $T_2$  cyclic tasks it could also be denoted that these tasks may alternate their execution with the *alternatesWith* constraint as  $T_3$  *alternatesWith*  $T_2$ . This means that  $T_3$  and  $T_2$  will alternate their executions.

Several alternatives could be applied for the  $T_1$  task. It could be considered that two constraints must be applied for  $T_1$  to execute only at determined instants given by `clock` and after the execution of  $T_2$ . It could be represented with *clock precedes*  $T_1$  and  $T_2$  *precedes*  $T_1$ . But if the intention is to represent that the execution of  $T_1$  is performed after  $T_2$  but a determined instant given by `clock` the closer restriction would be  $T_1 \equiv T_2$  *sampledOn clock*.

### Discussion

For MARTE profile only some intents are proposed as [57], but for System-on-Chip embedded systems only partial dynamism is provided. On the other hand other works (see [53]) deal with distributed reconfiguration of real-time systems while this work is centered in local timed reconfiguration.

One of the limitations of MARTE is that it is designed for real-time systems that lack of dynamism. It is not designed to represent changes in the system at run-time. Schedulability

analysis can be applied by tools at design time but they cannot be used to apply schedulability analysis at run-time because of components which timing execution parameters change.

### **2.4 Dynamic systems and Components**

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 [58]. 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 also known as run-time evolution or adaptation [59]. The main reason to introduce dynamism in real-time systems are the risks of having critical systems that cannot be easily modified to face possible system failures and environment changes [60].

In this section, we explore concepts and techniques used for dynamic software adaptation.

#### ***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 updates are a requirement for many software systems [61] where run-time updates are essential to be able to provide continuous service upon failure as well as to fix bugs or provide new features without halting the system.

This dynamism uses to be unacceptable in critical systems due to the unpredictability of the consequences of run-time modifications of the systems. System modifications change the global requirements of the system and can make it be unstable or unfeasible under the new conditions.

On the other hand this dynamism can simplify enormously the development of the system. Increases the reusability of components, which offer a common interface without the need to know the internals of the used implementations at development time and may allow updating or repairing some parts of a critical system without stopping its execution.

#### **2.4.1 Definitions and Concepts**

According to the standard glossary of software engineering systems [62], 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 [63].

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 adaptation permits an application to be recomposed dynamically during execution. This recomposition permits changing the application behavior, beside adding or eliminating parts of the application. This dynamic recomposition is called dynamic (or run-time) 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 [64].

Software capability to be adapted to environment changes is becoming more important every day. Resource management is one of the characteristics that make a system be able to adapt itself to the needs of the environment, the applications running at a given time or its usage.

Dynamism is also important when updates are required at run-time to be able to fix software failures or apply optimizations to a high availability system.

Wei Li introduces [65] a set of required characteristics for dynamic reconfiguration of component based systems. An update will not be correct if it can produce incorrect timing that breaks ongoing transactions or the state of the system is not correctly transferred. Here are described some of these characteristics. Almeida in [66] also remarks some of these requirements.

One the main requirements in a component reconfiguration or replacement is **consistency**, where structural integrity of the system is maintained after the update. Also the relative states of the components are consistent with the previous execution so that this execution can continue without fails. Some invariants may need to be maintained in the system after the update for consistency purposes.

Other relevant aspect of a component replacement is the **availability** representing the capability of the whole system or part of it to keep running while the replacement process is accomplished. Quiescence [67] is defined as a state where the whole system is stopped and prepared for a reconfiguration. In this state no transaction will be interrupted or lost during the reconfiguration. Tranquility [68] is proposed as an alternative. It represents blocking only the affected components by the reconfiguration process while maintaining in execution the rest of the system.

Provided that the framework makes use of a **timing control** over the reconfiguration or replacement then modifications can be made to the system with stopping any of its running components or in a limited amount of time. This generally means a total availability of the components of the system during the reconfiguration operation.

**Coexistence** makes reference to the capability of maintaining simultaneously different versions of running components. **Continuity** is when the replacement is made progressively and requests during the process are not interrupted.

Finally **QoS assurance** is considered as the capability to assurance that the QoS provided by the system will not be affected by any of these reconfiguration processes. An interesting element that makes the platform able to provide QoS Assurance is the use of **contracts**. These contracts represent the quality levels provided and required by installed components as well as the platform. These contracts also allow negotiation and management of quality levels during reconfigurations.

### 2.4.2 Other Approaches for Dynamic Software Adaptation

Dynamic compositional adaptive software is the software which is able to adapt itself and its components at run-time to handle resource availability at the moment and other environmental changes. Most approaches implementing dynamic compositional adaptation are based on dynamically linking and unlinking components or indirectly intercepting and redirecting interactions among software entities [69]. Various techniques may be used to achieve this, such as, manipulating function pointers, aspect weaving, proxies or middleware interception [70]. In this section are described some alternatives to the component-based systems:

- Separation of Concerns
- Computational Reflection
- Dynamic Service-Oriented Architectures
- Web Services

All of these alternatives provide dynamism mechanisms to design and construct dynamic adaptive software.

#### *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 [71].

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*. This weaver can insert aspect code in specific code locations, known as join points.. In order to select join points to insert aspects code, point cuts are created.

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

AspectJ [72] 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 run-time 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.

### ***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. 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 [73].

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

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

### ***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 [74].

Dynamic SOA (D-SOA) adds the dynamism to SOA. This dynamism can be depicted in two different concepts: dynamic availability [75], 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. In both cases, the service consumer must be notified of the context changes [76].

Other technologies used to adapt software architectures at run-time are P2P (Peer2Peer), software design patterns, agent-oriented programming and generative programming [70].

### ***Web Services***

Web Services [74] are, perhaps, the responsible of service-oriented computation of being so popular. They allow the interoperable machine-to-machine communication over a network. A web service is a service, identified by a URI whose service description (which is made using WSDL, a XML-based language) and transport (services interact by means of SOAP calls carrying XML data) is performed using open Internet standards. Service discovery uses a UDDI 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 [77], 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.

Similarly, Microsoft proposed new specifications based on web services, like UPnP [78] with the use of SSDP (Simple Service Discovery Protocol) for the automatic discovery of services in local networks with the use of SOAP for service invocation. DLNA<sup>1</sup> (Digital Live Network Alliance) is an example of implantation of this technology.

Another proposal originated by Microsoft is DPWS<sup>2</sup> (Device Profile for Web Services). This technology is closer to web services. It is similar to UPnP but oriented to devices with limited resources where UPnP could consume more resources.

### **2.5 OSGi Service Platform and Real-Time Specification for Java**

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.

Java is one of the most widely used languages. It offers many advantages over other languages like memory management (garbage collector), that at the same time represents an issue in real-time systems because of unpredictability. But many other characteristics represent a problem for modularity, that is not supported natively, making decoupling of concerns somewhat difficult sometimes. The lack of control of different versions of classes code is also related to this problem, while the low level usage of *classloaders* [79] do not solve the problem completely.

The OSGi platform comes to create a module system for Java which tries to solve these problems. OSGi uses some layers to manage the use of Jar files, classpath access and classloaders so that every module has only access to modules it is related to.

Next subsections describe the OSGi platform more in detail. First are included some definitions and concepts used in the platform, then the OSGi platform layers and a brief survey of existing implementations.

#### **Definitions and Concepts**

The OSGi specification defines a way to create true modules (bundles, in OSGi terminology)

---

1 <http://www.dlna.org>, September, 2011.

2 <http://www.ws4d.org/>, September, 2011.

and to make them interact at run-time. 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 classpath. 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, OSGi offers a local services registry to support the services publish, search and bind mechanisms.

The OSGi Service Platform specification is divided into two parts: OSGi framework core and OSGi Standard Services. While the first is the run-time which provides the functionality of the OSGi platform, the second one defines APIs for common tasks. In turn, the framework is divided into three layers: Module Layer, concerned about code sharing and packaging; Lifecycle Layer, that focuses on the runtime module management; and Service Layer, which deals with modules interaction and communication. Next subsections describe these layers.

### ***Module Layer***

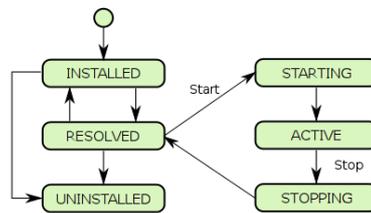
The module layer is the responsible for the bundle management. As described before, bundles are the unit of modularization of the OSGi platform, in the form of a JAR file with resources and additional metadata on its manifest file. This additional information includes human-readable information, bundle name and code (packages exported or imported by the bundle). This information is used to perform bundle dependency resolution. 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 in the OSGi framework, 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 provides logical boundary enforcement, version verification, reuse improvement, configuration flexibility and configuration verification.

### ***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 2

shows a state diagram containing all possible state during the lifetime of a bundle.



**Figure 2:** OSGi Bundle Life Cycle

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. A bundle in the Resolved state can be started when executing the start command, which leads it to the Starting state. The framework then looks for the `Activator` class of the bundle indicated on its metadata and executes its `start()` method. 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. If its dependencies were already resolved, the bundle returns to the Resolved state. 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 Uninstalled.

The OSGi framework makes no automatic management of services resources of bundles. Activator methods like `start()` and `stop()` must start services and register them to make them available to the rest of the platform. Any other resource must also be reserved and freed when these methods are invoked. A usual example of unmanaged resources by the OSGi platform are Java threads created in the start method but not stopped when the bundle is uninstalled. This leaves resources occupied (like CPU and memory) that cannot be used by other bundles and can generate errors in the platform.

The Lifecycle layer depends on the Module layer. The Module layer manages dependences between bundles that are installed. Resolved state of the bundles depends on these dependences.

### ***Service Layer***

The Service layer defines a model to provide and consume services as in SOA. Bundles can publish and discover services making use of the OSGi Service Registry. This registry is accessible through a `BundleContext` object, which is used by OSGi bundles to access the OSGi framework facilities.

In the OSGi specification, services are actually POJOs with associated Java interfaces (contracts) and meta-information which are published in the OSGi Service Registry. Whenever a

bundle needs a service, it makes use the BundleContext object to access the service registry 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 ServiceReferences, that is, the information and the indirect reference for the corresponding services providers. Through the ServiceRefence it can be retrieved the real service implementation (POJO) and make direct use of it (binding).

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 service is unregistered.

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.

### ***OSGi Implementations***

Since its first release, many implementations for the OSGi specification have been developed. One of the first open source OSGi implementations was Oscar. This implementation was later integrated as part of the Apache Software Foundation<sup>3</sup>. Apache Felix, Eclipse Equinox and Knopflerfish are examples of currently well known implementations. As of date the last OSGi specification version is 5.0 (Release 5)<sup>4</sup>. We list some OSGi implementations in the Table 1.

<b>Name</b>	<b>Developer</b>	<b>R4 Compliant</b>	<b>License</b>
Apache Felix 4.0.3	Apache	Yes	Apache License v2.0
Equinox 3.8.1	Eclipse	Yes	EPL v1.0
Knopflerfish 3.4	Knopflerfish	Yes	BSD

**Table 1:** OSGi Implementations

In addition, many enterprises have started adopting OSGi technology in their solutions, mainly for their rich client platform applications. One of the first applications of the OSGi technology in the industry was in the Eclipse IDE<sup>5</sup>, under the form of Eclipse plug-ins. Cisco is another giant company which included ProSyst's mBS as optional add-on on its Application Extension Platform (AXP)<sup>6</sup>. Cisco AXP allows the integration of applications with Cisco's Integrated Services Router (ISR).

<sup>3</sup> <http://www.apache.org/>, September, 2011

<sup>4</sup> <http://www.osgi.org/Download/Release5>, June, 2012

<sup>5</sup> <http://www.eclipse.org/osgi/>, September 2011

<sup>6</sup> [http://www.cisco.com/en/US/prod/collateral/routers/ps9701/data\\_sheet\\_c02\\_459075.html](http://www.cisco.com/en/US/prod/collateral/routers/ps9701/data_sheet_c02_459075.html), June, 2012

### 2.5.1 Real-time Java and OSGi

Java [80] 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++ [81]. 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 [82]. As a result, many different solutions were designed to improve the determinism of conventional Java.

In the next sections, Java issues with real-time and solutions are introduced. The principal shortcomings found using the Java to design real-time systems are explained. Different solutions developed to overcome the difficulties are discussed section as well as implementations of real-time systems in Java.

#### *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 [83]. 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<sup>7</sup>, JVM [84] and JVM hardware implementations [85], [86]. 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 [87]. 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.

---

<sup>7</sup> <http://download.oracle.com/javame/embedded.html>

- **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 (WCET):** 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.

### ***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 (beside the already existing not real-time Java priorities), 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. In addition, it 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 [88].

### ***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 2.

Implementation	Developer	Certification	Implementation	Platform compatibility	Java Compatibility
RTSJ-RI <sup>8</sup>	Timesys	Yes	Reference	RTLinux/x86	JRE 1.3
Java Real-Time System <sup>9</sup>	Sun Microsystems / Oracle	Yes	Commercial	Solaris, RTLinux/x86, Solaris/SPARC	JRE 5.0
IBM Websphere Real-Time <sup>10</sup>	IBM	Yes	Commercial	RTLinux/x86	JRE 6.0
Real-Time JRE <sup>11</sup>	Apogee	Yes	Commercial	RTLinux/x86, LynxOS	JRE 5.0/J2ME
Jamaica Virtual Machine <sup>12</sup>	Aicas	No	Free / Commercial	RTLinux, SunOS, Solaris/x86	JRE 5.0
J-Rate (Java Real-Time Extension) <sup>13</sup>	University of California	No	Open Source (GPL)	RTLinux/x86, RTLinux/PowerPC	JRE 1.4-5.0
OVM (Open Virtual Machine)	Purdue University	No	Open Source (BSD)	RTLinux/x86, PowerPC; OSX/PowerPC	JRE 1.4-5.0

**Table 2:** Implementations of the RTSJ

### *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 [89]. In addition, it is not well suitable for hard real-time applications due to performance issues [90]. In order to overcome those issues, many independent solutions were already proposed. For instance, Aonix PERC<sup>14</sup> is initially based on RTSJ, but it defines its own class hierarchy. PERC integrates a static analysis system to verify scope safety and resource requirements for hard real-time systems.

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

8 <http://www.timesys.com/java/>

9 <http://java.sun.com/javase/technologies/realtime/index.jsp>

10 <http://www-01.ibm.com/software/webservers/realtime/>

11 <http://www.apogee.com/products/rtjre>

12 <http://www.aicas.com/sites/jamaica.html>

13 <http://jrate.sourceforge.net/>

14 <http://aonix.com/perc.html>

adaptations at runtime. In the next section, it is introduced OSGi as a platform for real-time systems.

### ***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. 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. As it is still at an early stage. In [91], it is presented a descriptive approach for real-time support in the OSGi framework, where the real-time guarantee is implicitly provided by the container run-time 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 native code (not Java) runs directly in the real-time operating system layer.

Richardson et al. in [92] 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 [93]. 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 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<sup>15</sup>. 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 [94].

Basanta-Val et al. [95] propose an architecture with different levels of integration to provide real-time characteristics to the OSGi platform. Following are the proposed levels of integration:

- *Level 0 - Real-time Java available for OSGi bundles*: The `javax.realtime` API provided by RTSJ is made available for installed bundles to import it. It requires the use of a real-time Operating System and a Java virtual machine with RTSJ support.
- *Level 1 – Real-time bundle description*: The bundle is characterized with its corresponding real-time parameters, i.e., memory model, garbage collector requirements, scheduling parameters (cost, deadline, priority, ...). A bundle named TROSGI (`es.uc3m.it.trosgi`) is incorporated here in the system to handle the characterization of real-time services in bundles and applying them.
- *Level 2 – Enhanced OSGi for real-time performance*: This level requires changes in the implementation of OSGi to provide admission control, fault tolerance and multi-constrained and adaptive admission control.

Issues raised by the consideration of real-time requirements in the OSGi Service Platform are discussed in the following subsections.

### ***Resource Management in OSGi***

Java VM provides encapsulation of the physical resources, hiding some resources management to the programmer. Automatic memory allocation and deallocation with the use of the garbage collector [96] is an example. The virtual machine hides the memory allocation mechanism to the programmer. The execution of the garbage collector is unpredictable. It comes an issue when the Java Virtual Machine is to be used in a real time environment [97].

The other main resource managed by the Java VM is the CPU usage. Threads provided by Java have an indicative number for priority that will be mapped to the underlying Operating System. But control over these threads scheduling, cost enforcement and other capabilities is inexistent in standard Java. RTSJ comes to solve this problem. Although not all implementations support certain capabilities like cost enforcements in threads, or even cost monitoring.

To be able to manage resources the first step is to monitor them. The next section describes works oriented to resources monitoring in the Java VM or in the OSGi framework itself. Following sections describe the efforts to provide Java and OSGi mechanisms to be able to manage resources usage in real-time environments.

### ***Resource monitoring***

---

15 [http://download.oracle.com/docs/cd/E13221\\_01/wlrt/docs10/index.html](http://download.oracle.com/docs/cd/E13221_01/wlrt/docs10/index.html), June, 2012

T. Meittien et al [98] propose a method to monitor resource usage of services in the OSGi platform. With this method resource usage of services or components in the OSGi platform can be analyzed and previewed. RTSJ is not used here for resource monitoring. Several methods are evaluated and discarded, like the use of Jconsole [99], NetBeans Profiler [100], etc. The tools finally used are JVM Tool Interface [101] and BCI (ByteCode instrumentation) [102] to respond to events received from JVM TI and polling for CPU usage information.

To be able to isolate resource usage accounting to the corresponding OSGi bundle modifications are made in the platform. Even with these modifications resources accounted for the OSGi platform itself and JVM, like garbage collector execution are not taken into account.

OSGi assigns a classloader to every bundle in the platform, but created threads in a bundle still belong to the “main” thread group. To be able to account threads resource usage to the corresponding bundle a modification to the OSGi platform is made so that a new different thread group is created for every bundle. This way every new thread will be assigned to its corresponding bundle thread group. These thread groups are managed separately for resource usage accounting.

There is a problem still to be solved when a service thread invokes a service of a different bundle. Execution of this service is done in the thread execution of the calling service, not the invoked service. So resources used by the invoked service are accounted to the caller service. To solve this problem a proxy service is used. This proxy implements the same interface that the service it acts on behalf of. Upon the reception of a service request it creates a new thread belonging to the bundle of the service provider. This way resources used by the services provider are correctly accounted.

One of the limitations of this work is that resource monitoring is made at bundle level. It does not take into account that a bundle may provide several different services or components. So resources consumed by a single service or component in a bundle cannot be known. This approach, like many others described below, does not take into account this fact so they suppose that every bundle provides only one service or component at a time.

As previously commented, ProSyst offers an OSGi implementation<sup>16</sup> for embedded devices on top of in the J9 Java VM provided by IBM. This JVM provides some resource management<sup>17</sup> capabilities like memory spaces and open sockets. Memory used by the bundle, opened sockets, available memory for the bundle, limited number of threads and data storage space are resources this OSGi implementation is capable of managing.

To be able to correctly account resources used by every bundle it encourages the service implementations to notify to the OSGi framework that a context switch (related to service provider context) is being made so that it knows which code belongs to every bundle.

This resource management, as in the previous work, is done at bundle level. Context switches are not even handled transparently using a proxy. The context switch has to be done by the services themselves so that the resource manager is aware of resources used by every bundle.

---

16 <http://www.prosyst.com/index.php/de/html/content/97/Products-OSGi-Implementation/>

17 [http://dz.prosyst.com/pdoc/mBS\\_PE/um/framework/concepts/resource\\_mngt.html](http://dz.prosyst.com/pdoc/mBS_PE/um/framework/concepts/resource_mngt.html)

This resource monitoring seems to be aimed at resource limitation due to hardware platform limitations ensuring a QoS, but this solution does not provide explicit QoS adaptation mechanisms like changing implementation of services, etc. or being aware of real-time constraints.

CPU and memory are not the only resources in the platform required to be used to provide services, as shown by the ProSyst OSGi implementation based on the J9 VM. Access to devices can be a resource to be shared among service providers. OSGi is designed so that devices can offer their own service provider to give access to device capabilities [103].

## **2.6 Real-Time and Reconfigurable Frameworks and Middleware**

Whereas dynamism and reconfiguration are widely applied in non critical systems, applying dynamism in critical systems, even in soft real-time systems is a difficult to implement issue. In this section some critical systems that provide dynamism or reconfiguration capabilities are described. Some dynamic systems and their adaptation to critical systems are also included.

Beside the already described architectures to provide QoS in previous sections, many other approaches are also used to provide dynamic software adaptation in critical systems. 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.

Some techniques to provide dynamism at run-time include the use of concurrent classloaders [104], agents [105], [106] and function blocks [107]. But these techniques are too specific, they are not component-based, and do not provide real-time capabilities.

Only component-based and service oriented systems are considered here. 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 [108]. Many different approaches have been used in the component-based software engineering literature in order to introduce real-time requirements in component models.

The presented related work in this section is divided in 4 groups:

- *Component-based dynamic adaptive systems*: In this subsection, component-based frameworks with certain degree of adaptation are described. Most of them do not provide real-time support.
- *Real-time support component-based systems*: Component-based systems with real-time support are described here. Some of them are based on the Java language and RTSJ support. Most of them do not support reconfiguration or component replacements.
- *Run-time upgrade/replacement frameworks*: Finally, proposals to support run-time replacements are described here. These proposals are similar to the work presented in this thesis, but all of them lack some characteristics that are covered here.

- *RT-OSGi and reconfiguration support*: Here are described the reconfiguration capabilities in the OSGi platform and its support for real-time.

### ***Component-based Dynamic Adaptive Systems***

Fractal [109] is a component model with support for many programming languages. Beside service interfaces it also provides control interfaces encapsulating component in a membrane. With these control interfaces it can provide control over component attributes, component bindings, content of components, and the component life cycle. Fractal also provides support for run-time configuration of components, but no safe real-time component replacements is considered.

SOFA [110] is another component model that provides run-time reconfiguration characteristics. It also contains specific reconfiguration patterns that permit modifying the architecture of the system at run-time according to predefined characteristics. This component model also lacks of support for real-time components and their replacement at run-time.

OpenCOM [111] is a component model designed to be efficient and to provide reconfiguration capabilities. It waits for the component to be free of working transactions and then performs the corresponding reconfiguration or replacement. No real-time characteristics are taken into account for the components execution and replacement.

Sharma et al [112] proposed the use of components, called qoskets components, to encapsulate and re-use adaptive QoS systemic behaviors over the Quality Objects (QuO) framework<sup>18</sup>.

Qoskets are reusable components that contain a generic behaviors and a set of related QuO contracts representing different qualities, system condition objects, and related code. Qoskets instantiations which implement the Qoskets functionality. Qoskets offer the same interface as their instantiations, working as proxy. Functionality is delegated to the instantiation while the Qosket is able to reconfigure it depending on the system conditions.

Qoskets are able to reconfigure the behaviors of their instantiations, but they are not designed to provide component replacements at run-time. It mainly represents a mode change.

ASSIST is a programing environment focused in the design of distributed, high performance, and reconfigurable applications [113]. It is based on the use of ASSIST-CL (Coordination Language). This language allows declaring a hierarchical direct graph of modules, where each module can be a structured subgraph of modules.

The use of ASSIST-CL allows the compiler to identify well defined, clear reconfiguration safe points. In these points the internal state of modules is considered to be stable and consistent.

For the reconfiguration, ASSIST makes use of three elements:

- VPM: virtual processors on which modules are supposed to work. They will be physically assigned on reconfigurations.

---

<sup>18</sup> <http://quo.bbn.com>

- ISM: in charge of managing input interfaces of modules, and data distribution to virtual processors.
- OSM: handles output interfaces of modules, sending output results and communications to the external world.

To perform reconfigurations, ASSIST generates reconfiguration-aware templates in the target languages (i.e., C, C++, FORTRAN) to implement the corresponding code for reconfiguration in a safe manner. This way, changes are applied at safe points.

Almeida et al. [66] proposes a method to provide dynamic reconfiguration support for distributed systems based on CORBA. The method is designed to maintain the structure of relationships between entities of the system, keep the application state elements that must not change, and to maintain the consistency of internal states of different entities. This work provides no support for real-time characteristics in components or system reconfiguration.

### ***Real-time support component-based systems***

There are many component models that aim at real-time systems development. A representative one of these works is [114] where aspects are used to implement several characteristics of components and to *weave* them with functional code. This aspect weaving requires calculating the end to end WCET of every component service invocation for schedulability analysis.

Calculations of WCET are performed at compilation time where aspects are weaved. No model or information is provided for dynamism for the system to be able to load, unload or replace components at run-time.

WCET and end to end deadline calculation of these frameworks are mostly based on [115] where all the schedulability characteristics of components composition are analyzed. It describes the different natures of the components regarding their execution as:

- ***Cyclic***: it encapsulates a periodic activity.
- ***Sporadic***: encapsulates a bounded aperiodic activity.
- ***Active***: the component encapsulates a background process (generally a cyclic activity).
- ***Passive***: it encapsulates reentrant operations, but not an active process.

This work also defines the end-to-end deadline as a constraint placed on a set of components. The end-to-end sequence of control and data-flow across components is considered a transaction. All components must collaborate in the transaction for the deadline to be met.

It also makes a distinction between synchronous and asynchronous deadlines. A synchronous deadline implies a precedence-constrained sequence of execution. When an object invokes a successor in the sequence its execution is suspended until the successor finishes its execution. In an asynchronous deadline it is supposed that threaded objects communicate by reading and writing data using shared protected objects. The execution of reader and writer objects are only limited by the synchronization of the shared data, but their execution is not suspended until the

execution of the other finishes.

VEST [116] is another component framework based on aspect oriented composition of components. This component framework also incorporates end to end scheduling calculation for aspects composition. This work also lacks support for run-time dynamicity.

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 [117]. Many research works are dedicated to this subject [117–119]. Another useful application for RT-SOA is the support of remote critical care [120]. It is also worth to mention the IRMOS European Project<sup>19</sup>, which investigates on the use of real-time technologies and SOAs for networking, computing and storage levels.

It is important for RT SOA that services express their Quality of Service constraints making use of Service Level Agreements [121] for RT SOA systems to be able to determine at run-time if changes in services composition are feasible according to QoS constraints.

Many research works focus on component models for building RTSJ-compliant applications [122–125]. Most of them provide higher-level abstractions for creating real-time threads and/or real-time memory management, in order to alleviate the development process. However, dynamic adaptation issues are only treated by [123].

Plšek et al. [123] define a comprehensive component model to implement most of the characteristics provided by RTSJ. Components are encapsulated in *membranes* that include *interceptors* to manage RTSJ concerns. Two categories of interceptors are used:

- *Active interceptors*: manage threaded components and their configuration.
- *Memory interceptors*: manage the type of memory used by the component as well as communication between different memory areas.

Plšek et al. also remark in their work that the dynamic adaptability in their framework is still a problem to be tackled.

RTComposer, a framework described in [126], is also built using 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.

Components can provide several implementation of their operation. The scheduler is able to create *macro-schedules* alternating the execution of such implementations correctly to not generate deadline misses.

This framework allows dynamism adding and removing components at run-time but macro-schedules are pre-calculated before the component is added to the system, and no support is provided for their replacement.

Another programming environment based on Java for creating real-time components is the

---

<sup>19</sup> [www.irmosproject.eu](http://www.irmosproject.eu)

Exotasks project [127], which focuses mainly on memory isolation and the notion of logical execution time LET [128] to offer exact execution times of components independently of the execution platform. This solution is similar to RTComposer, although not based on the RTSJ specification but on the use of the J9 IBM virtual machine, modified to offer some memory isolation.

iLAND (mIddLewAre for deterministic dynamically reconfigurable NetworkED embedded systems)[129] is an ongoing research project whose objective is to develop a component-based middleware with deterministic dynamic functional composition and reconfiguration. Operating systems, service-oriented and real-time approaches are combined to achieve this objective.

Some of the specific objectives are the following:

- A component-based middleware architecture for embedded systems with abstraction of the specific platform and resources.
- Determinism for composition and reconfiguration algorithms applied to service-based networked applications.
- QoS-based resource management to support environmental or programmed changes according to needs. These changes will be based on deterministic platform enhancements.
- Application modeling support for deterministic dynamic reconfiguration and composition in development tools.
- Validation through prototypes.

As noted in the objectives real-time composition and reconfiguration is provided. But this reconfiguration is deterministically designed.

In [129] and [1] Marisol et al. introduce a service-based framework and a time models for real-time services reconfiguration. The framework allows the run-time reconfiguration of services based on all the previously available service implementations and established configuration modes. The time model makes use of budget scheduling to reserve enough resources to reconfigure the application services.

In [1] a reconfiguration budget time is modeled to provide to the applications in the system to be reconfigured. Reconfiguration process is split in a reconfiguration initialization, applications' reconfiguration times, intermediate activities between reconfigurations and reconfiguration wrap-up. These times are not fixed, so a budget is previously calculated and fine tuned at design time.

As stated before available application configurations are established at design time. Component replacement is not explicitly contemplated. The replacement of a service implementation would imply a change in reconfiguration times that has not been foreseen.

### ***Run-time upgrade/replacement frameworks***

In [130], it is noted the need to take into account the WCET verification at run-time when components are upgraded. But it only supposes that the WCET of the new component is less or

equal to the time-budget of the replaced component, with the same period and deadline. No schedulability test is contemplated in other case. Resource reservation for the component replacement itself is not contemplated either.

Regarding the capability to provide component replacements at run-time with safety, a model is proposed in [131]. Sha proposes to keep the old working component in the system when it is replaced. In case that the execution of the new component fails then the execution of the old component is restored for safety purposes. No schedulability analysis is provided to assure that component replacements are actually schedulable or the QoS is maintained.

Rasche and Polze in [132] and [133], presented a technique for dynamic reconfiguration of component-based software, in which the time where components are blocked is used for management tasks like reconfiguration of components. Actually the reconfiguration is managed and applied in a way that the execution of all the current transactions between components is correct. Although they claim that they apply the algorithm they still include a time where the application is interrupted, which definitely cannot assure meeting deadlines. The process includes the following steps:

1. The configuration is requested.
2. The application is interrupted, known as *black out* time. Transactions affecting the components to be replaced are blocked.
3. New components are *loaded and initialized*.
4. Old components that are not used in the new configuration are *deleted*.

The real-time deadlines of the transactions are not taken into account and no model is provided to apply reconfiguration at safe instants nor schedulability of the whole system.

As stated before, in [68], the concept of tranquility is proposed where only the affected components are blocked during the replacement process, although this tranquility is not proved to be reachable in bounded time.

Other works are actually real-time concerned, as the one proposed by Wahler in [134]. In this work a model is presented for component replacement. This model is focused in the copy of the state of the component, which can take several execution periods until it is completed. Although this method does not assure that the copy of the component state is completed in a fixed number of periods given that the component is still working and its state modified. A method based on the use of two elements is proposed. A *teach* and *learn* elements are added to the system. The *teach* element knows the internal organization of the data in the component to be replaced and transfers this information to the *learn* element. The *learn* element populates the internal data of the new element with the received information.

As much as possible data is copied from the old component to the new one in each iteration. If already copied information is modified in the running component then will have to be copied again until the copy is completed. This method also obviates the need to analyze the available slack time to make these component updates and the management code that realizes this updates.

Schneider et al. [135] introduce a proposal to make use of their real-time middleware OSA+

to be able to apply component replacements. Different levels of application blocking are proposed to perform replacements:

- *Full Blocking*: The old component receives a reconfiguration request. While the old component is blocked the state is transferred and the components are switched. Finally the new component is active. The old component is stopped since it receives the reconfiguration request.
- *Partial Blocking*: The new component starts the state transfer while the old component is still working. Finally a components switch is requested. If some data has been modified in the while it is updated. The old component is blocked during the switch process. The worst case is supposed to take as long as the full blocking case.
- *Non-Blocking Approach*: This approach is similar to the partial blocking. In this case the remaining information to be transferred is evaluated before the components switch is requested. If the time required to perform the information transfer is lower than the acceptable black out time then the switch is applied. In other case the switch is delayed while the information continues being transferred.

Partially blocking, and non-blocking modes are proposed depending on how many components are blocked during the replacement process. Timing control of the replacement is supposed, but no exact temporal model is described for the replacement. Neither CPU time reservation nor interference of a replacement in the schedulability of the system is considered in this work. The time required for a replacement is not bound, although the non-blocking approach should provide some kind of bounds.

To provide QoS assurance during the reconfiguration, previous works like [1] already propose methods to reserve resources to be able to reconfigure the system according to its varying needs. This method is oriented to multimedia systems and based on a budget scheduling model and dynamic priority assignation [36]. Resources are reserved for every component independently so that these components can perform a reconfiguration to be adapted to the QoS needs of the system. This method is centered in providing QoS assurance for a continuously reconfiguring system but for multimedia systems. Budget times are designed for reconfiguration of components.

Gorinsek et al. [136] proposes a model based on contracts to assure QoS of the system during the update. The system also proposes the use of monitoring and resource enforcement to correctly accomplish these updates. Updates are accepted or rejected according to contracts and resource monitoring. Three types of contracts are proposed:

- *Component contracts*: It specifies the minimum and maximum amount of resources the component requires.
- *Intercomponent contracts*: It details messages exchanged by the component.
- *Update contracts*: It details the maximum amount of resources an update may spend. This takes into consideration the updating method to be used, such as state transfer.

No actual solution is provided to implement such model and contracts in a time controlled

way or maintaining a minimum availability of the system.

### ***RT-OSGi and reconfiguration support***

To ease the modularization and reconfiguration of systems the use of middleware represents a good option. Middleware provides mechanisms to manage software components, their organization and the way to communicate them. Middleware is then, the way to provide reconfiguration of components and non-functional characteristics. The main middleware platform analyzed here is the OSGi platform.

The OSGi Service Platform is a service platform which addresses the lack of support for modularity in Java applications [79][103]. OSGi provides management of bundles (jar packages containing code as well as other kind of resources). These bundles can be started and stopped so that they can instantiate services and publish them through the subscription mechanism provided by OSGi. This subscription mechanism consists of a service registry. Instantiated services can be published in the registry by service providers and searched and bound by service clients. Services can be added or eliminated from the platform at any time by active bundles, other services or the fact of installing, uninstalling or updating bundles.

The platform dynamism and the use of a garbage collector by the Java platform makes the system unpredictable. This unpredictability represents a real concern in the use of the OSGi platform for real-time applications and the use of implementations of the Real-Time Specification for Java [137].

Unlike typical real-time platforms OSGi introduces a grade of dynamism that makes the response times and resource availability to change at run-time, affecting running services. So it has to be taken into account in order to predict feasibility of the whole system.

The component model provided by OSGi is named Declarative Services [103]. It is based on service components which declare the provided service and required services to work. Services (and components based on them) can be loaded, unloaded and replaced at run-time without stopping or blocking the execution of related components. This is done making use of continuity where old components are maintained while transactions or references to them are kept by other components. No QoS assurance is provided at all by the platform and neither a timed execution of these maintenance tasks.

One of the last proposals to provide real-time characteristics to OSGi [95] through the integration with RTSJ also does not provide direct support for run-time replacement of components. Although the proposed architecture may provide support to implement the work described in this thesis.

This dynamic platform represents a good target as a dynamic system to which provide a method to perform safe component replacements at run-time with real-time characteristics.

## ***2.7 Discussion***

Nowadays hardware is not a real issue in real-time systems where applications must be not only logically correct but also temporally correct. So most of the work has been focused on

creating deterministic and safe software in logical and temporal aspects.

The next step comes to provide these real-time systems with new capabilities like adaptability to the environment changes. There are many paradigms to achieve this adaptability already existing in the computational world but most of them are not real-time aware. Service Oriented Architectures is one of these approaches. One of the most well known service platforms available is the OSGi framework. This framework is characterized by its abilities for dynamic deployment of services (as well as updating and removal of these services).

### ***OSGi Constraints in Real-Time Environments***

One of the main characteristics of the dynamic platforms like OSGi is their flexibility and the capability to change the execution environment at run-time (whether adding or removing services). This changes in the platform uses to render this platform unavailable to support real-time systems given that these changes may take the system to unknown or unpredicted conditions. Some aspects of the relationship of OSGi with real-time systems and are studied in this section.

The dynamism of service-oriented component models [138] may compromise the determinism and reliability of real-time applications. Services changes do affect the WCET analysis, as well as resources reservation of already running services. This makes the application behavior unpredictable.

The capability to install new services may lead the framework to a state where resource availability can not be guaranteed for all of the installed services. This represents a security issue that can provoke a Denial-of-Service attack [139]. The most used technique to avoid these denial-of-service is the temporal isolation of services so that no service can interfere with other running services in the platform and reserved resources.

Another issue is the lack of global view of the system when developing a component. Priorities can be assigned to the component threads that may not guarantee the final application to work properly. A correct scheduling analysis and assignment of priorities across components is required to assure the avoid of starvation or deadlocks of services.

One of the OSGi problems is the lack of real-time capabilities at all. It is implemented using standard Java and not RTSJ. The use of different kinds of memory areas in RTSJ like immortal memory and scope memory may lead to unresolved references as well as impossibility to unload classes and hence uninstalling or updating a service.

The standard Java threads also do not provide the CPU time reservation and control like those of RTSJ that require real-time applications. This threads usage neither assures a secure threads termination when a service is uninstalled. Services can leave threads running consuming resources that should be freed or provoking exceptions in the system trying to use freed resources.

In general, OSGi provides no mechanism for resource management, so this concern is left to the components programmer.

A generic model is presented in this work that can be applied in any dynamic framework,

based on real-time or for QoS adaptation purposes. This work also proposes a method to assure that further updates can be applied according to resources needed by the component. This way enough resources are reserved to keep the QoS and feasibility of the system during the replacement process. This replacement is executed in a timely controlled way so that no component is blocked for the process.

Proposals	Consistency	Availability	QoS assurance	Timing Control	Contract
Almeida et al. [66]	Yes	Partial	No	No	No
Rasche el al. [132]	Yes	Partial	No	No	No
Qoskets [112]	Yes	Partial	Yes	No	Yes
Plšek el al. [123]	Yes	-	-	Yes	No
iLAND [129]	Yes	Total	Yes	Yes	No
Wahler el al. [134]	Yes	Total	Yes	Yes	No
Gorinsek el al. [136]	Yes	-	Yes	-	Yes
OSA+ [135]	Yes	Total	No	Yes	No
OSGi [103]	Yes	Total	No	No	No
This work	Yes	Total	Yes	Yes	No

**Table 3:** Component reconfiguration frameworks

Table 3 shows a summary with the closer related works and their characteristics. These works mostly provide component replacement or reconfiguration with some real-time support.

The table summarizes the characteristics required by components reconfiguration or replacement, and described in Section 2.4.1. Some of these characteristics are not described by some of the referred works, as the possible availability of the system during a reconfiguration process in Plšek el al. [123], so no information can be provided in the table.

All of the proposals described here are designed to maintain the **consistency** in the system in a reconfiguration process. Else, the reconfiguration process would be useless.

The first proposals provide only partial **availability** of the system. This means that, although parts of the system may still execute, the affected components are blocked until the reconfiguration finishes. These options are not considered useful in a real-time environment because deadlines would not be met.

Some of them are also designed to maintain the **QoS** provided by the system during the replacement or reconfiguration process. This does not mean that real-time requirements are met. This means that enough resources are reserved for the components or the reconfiguration process to maintain or to adjust the provided QoS according to the needs.

Proposals with **timing control** are those which provide real-time characteristics and are aware that a timing control has to be applied in order to maintain the schedulability of the system. Some of them do not provide such timing control, although they are QoS-aware. It is a required characteristic for a real-time system to execute properly.

Only two proposals take into account the explicit use of **contracts** for the components execution or their reconfiguration. It is considered here that the execution parameters of a

component represent a contract for the system when the component is accepted. The system is supposed to provide enough resources for the correct execution of the component according to its timing requirements.

Wahler et al. is one of the proposals that covers most of the characteristics. This proposal still lacks some requirements as an acceptance test to assure that those replacements are possible. The time required for the execution of the replacement is not bounded.

Gorinsek et al. only describes a generic system based on contracts, but no replacement model is described.

OSA+ describes different replacement models to be applied in the framework. A concrete specification of these replacements would be necessary to appreciate if they can be bounded in time. A description of the resource management performed in the system for the replacement process would also be useful. The overload in resources generated by the replacement process is not taken into account.

The solution provided takes into account all of these aspects. The replacement of the component is designed to maintain the consistency of the system. The availability of the system is total, applying the replacement without interfering in the execution of other components. The QoS is maintained because the resources needed by the replacement process to replace any component are taken into account in the acceptance test. The schedulability of the system is also maintained due to the schedulability analysis performed in the acceptance tests. Although not explicitly, this also represents a contract regarding the QoS aspects of the system.

## Chapter 3

### System model and framework overview

---

The system model, as well as the type of components managed in this work, are detailed in this chapter. This sets the basis for the type of schedulability analysis to be applied and its repercussion in the proposed replacement models. Here are described:

1. A *component-based system model*, with the characteristics and elements required for analysis of temporal behavior.
2. A *schedulability analysis* technique for the described system model taking into account the component characteristics.
3. A description of the *problem* to be solved and the possible alternatives to solve it. The following chapter describes the solution to the problem proposed in this work.

Following the definition of Szyperski [140]:

*“A software component is a binary unit of composition with a contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”*

The type of components contemplated in this work contain the following elements: an internal *state* and *interfaces*. The internal state represents the information that the component uses for its internal workings. Interfaces allow the component to communicate and interoperate with other components. Some components can have an internal task to implement the component functionality. The component state will only be taken into account when the component replacement model is described because it is assumed that it does not affect the components schedulability.

Component interfaces are the basic enablers for the interaction of components. They express a contractual functionality offered by the components. Given that interfaces represent a specified

contract, they can be published and found by other components in the system. Components may be deployed once, and they do not change nor are eliminated from the platform. Then, the use of specified interfaces has a dynamic nature that can be provided to a component [138]. The specified interfaces allow higher dynamicity in the system because components can be replaced while their interface complies with the interfaces of the components that it is connected to. So, this contractual nature also allows components to be added and replaced while the system is being executed if the new components adjusts to the same contracted functionality.

### ***Target applications***

Many real-time applications can be benefited of seamless component replacement. This component replacement can be used as a way to perform mode changes in the system or updates of components.

Component replacements require additional processor time to be performed. Hard real-time embedded systems may not be able perform replacements due to the use of very adjusted processor power and uncertainties. On the other hand not embedded or soft real-time systems may be benefited of the replacement models proposed in this work.

Two replacement models are proposed. Different kind of applications can make use of different replacement models. One of the replacement models reserves a high amount of processor time for the replacements, obtaining almost immediate replacements when they are requested. The other replacement model allows reducing the reserved processor time for replacement at the cost of a higher time since the replacement is requested until it is performed. In case that more than one replacement are requested at the same time, they should be prioritized and applied in order.

This proposal is centered in one replacement at a time. It does not solve the problem of replacing multiple component at a time. But it can be used as a base to perform such kind of replacements.

Application on which to apply the proposed solution must fulfill some requirements that are described here:

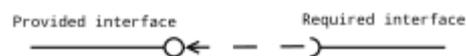
- It has *real-time* characteristics. Responses or computation have to be completed before a specified time deadline.
- It is expected that the application suffers functional updates with relative frequency but not constantly, i.e., a typical streaming video server.
- No distributed applications are considered here. Only the schedulability of centralized applications is considered.
- The application design is component-based to provide dynamism, among other component-based characteristics as composability and reutilization.

### ***3.1 Component-based system model***

The proposed component model contains *active* and *passive* components [141] (see Figure 5).

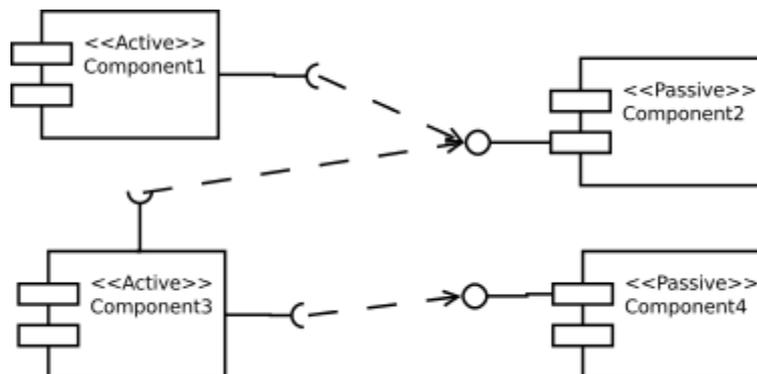
- *Passive components*: Interfaces are sets of operations that can be invoked by other components offering a specified functionality. Passive component operations are only activated when invoked or when an event request is received. Passive operations are code of the passive components but they are executed in the context of the calling thread.
- *Active components*: Contain threads of execution that follow different activation patterns. In this model, only periodic and single task components are considered. Active components are modeled with an execution period ( $T$ ), an execution cost ( $C$ ), a deadline ( $D$ ) and a priority.

Basically, passive components provide interfaces that are required by active components. Active components invoke operations of the provided interface of the passive components. Figure 3 shows the nomenclature used in this work, based on UML [142].

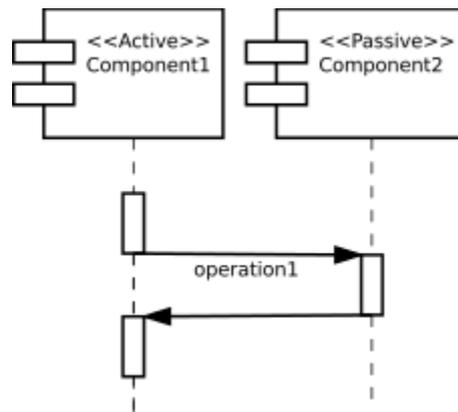


**Figure 3:** Provided and required interfaces

Passive components offer a service provided through operations that can be invoked (see Figure 4). Each operation requires some time to complete, therefore it has an associated computation time. Synchronous invocation of operations is considered here. This means that active components invoke operations of passive components as part of their functionality. The execution time of the active component is increased by the execution time of the passive component invoked operations (see Figure 5). No operation is invoked asynchronously, since passive components do not contain tasks. In that case, this component would be considered part of an active component operation that is also invoked periodically.



**Figure 4:** Component types in the system



a) Operation invocation

```

Active Component1 {
    Thread {
        ...
        Component2.operation1();
        ...
    }
}
  
```

b) Code of active component thread

```

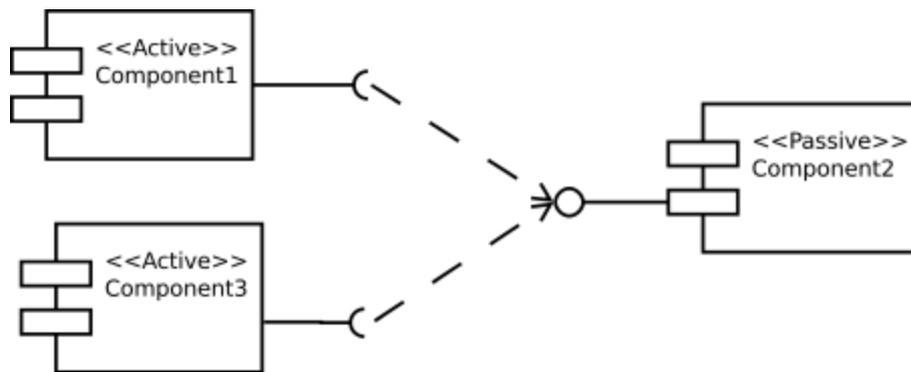
Passive Component2 {
    operation1() {
        ...
    }
}
  
```

c) Code of passive component operation

**Figure 5:** Synchronous invocation of operations

In this work, it is assumed that active components are periodic, i.e., they provide functionality that is realized by a periodic task that is activated at constant time instants ( $T$ ). It makes use of code provided by passive components in the system. The time to complete such functionality depends on the deployed passive components in the system and the time that they require to complete their functionality. For example, Figure 6 shows that `Component1` and `Component3` depend on execution times of operations in `Component2`. If `Component2` is replaced then the execution times of the threads of `Component1` and `Component3` will also change.

Delays of operation invocations have to be considered to support the shared use of passive component's functionality. If operations in the passive component are not reentrant, then invocations of operations of passive components will block and delay the execution of the task in active components. An operation is not reentrant if it cannot be invoked while it is still in execution; it would create inconsistencies in its executions, and produce delays in the execution of tasks due to such dependencies [143].



**Figure 6:** Shared use of passive component

### *Temporal properties*

Each active component has a periodic task  $\tau_i$  that executes its functionality based on the invocation of passive components operations (  $C_i^{pass}$  ).

The system functionality is described as a set of tasks (see Equation (12), where  $m$  is the number of active components), each one providing its functionality. Each task's functionality can be described based on the composition of functionality provided by the existing passive components at the execution time.

$$System = \{ \tau_i : 1 \leq i \leq m \} \quad (12)$$

To be able to analyze the schedulability of the system, the following data have to be specified with respect to the components (see Table 4):

- The *computation time of every active component's task* has to be calculated. That represents the list of mission tasks in the system, and all of them have to be schedulable.
- The *computation time of every operation in passive components* has to be considered. They represent part of the execution time of the active component tasks.
- Beside the provided operations by a passive component, information about their *Worst Case Execution Time* has to be provided to the system. The WCET of the operation is the maximum time that the operation could take to execute. Also, the information about the number of times that the active component invokes the passive component operation has to be provided.
- It is also necessary to indicate if the operation is reentrant or not, to be taken into account in the schedulability analysis of the system. As described before (see **Figure 6**), the invocation of an operation can be delayed if the passive component operations are not reentrant. These delays have to be taken into account in the computation time of active component tasks. An example of temporal information of passive component is provided in **Table 4**.

Operation	WCET	Reentrant
Op <sub>j,1</sub>	C <sup>pass</sup> <sub>j,1</sub>	Yes/No
Op <sub>j,2</sub>	C <sup>pass</sup> <sub>j,2</sub>	Yes/No
Op <sub>j,3</sub>	C <sup>pass</sup> <sub>j,3</sub>	Yes/No
...	...	...

**Table 4:** Temporal information of operations of passive component  $j$

In this work, it is assumed that the system scheduler implements a Priority Inheritance Protocol [143]. The concurrent execution of multiple active components and the shared use of passive components may provoke undesired delays in the execution of active components. The invocation of shared operations may delay the execution of an active component (e.g.,  $Co1$ ) by another active component with lower priority (e.g.,  $Co3$ ). The component with higher priority has to wait until the component with lower priority finished the shared operation invocation. During this delay, another component with lower priority than  $Co1$  may preempt the execution of  $Co3$ . This preemption may provoke unbounded delays in time.

The Priority Inheritance Protocol raises the priority of the lower priority component ( $Co3$ ) to be the same as the higher priority component that it is blocking. In this example it is the priority of  $Co1$ . In this case,  $Co2$  would not preempt  $Co3$  and it will not generate unexpected delays.

If a component invokes an operation of passive component  $j$  and the operations is not reentrant, i.e., the component represents a shared resource, then this invocation would be delayed until a previous ongoing operation invocation finishes. The use of a Priority Inheritance Protocol is assumed. This represents delaying the execution of the active component for a time that can be in the worst case the maximum of the computation time of the passive component operations. The maximum time of delay in invoking an operation of passive component  $j$  is represented as  $B_j$  in Equation (13), where  $k$  is the number of operations in the passive component:

$$B_j = \max\{C_{j,i}^{pass} : 1 \leq i \leq k\} \quad (13)$$

To calculate the computation time of the active component task it is needed the description of the behavior of the task. This description has to include the execution time of the task code ( $C_i^{act}$ ), and the information about the invoked operations ( $C_{j,l}^{pass}$ ) as well as the number of times that those operations are invoked ( $n_{j,l}$ ) by the active component task. This information is provided by the active component. In Table 5 it is shown an example of detailed information about the execution of an active component task's execution.

Active Comp.	Min. Cost	Operation Invocation	Invocation Times
$Co_1$	$C^{act}_1$	$C^{pass}_{1,1}$	$n_{1,1}$
		$C^{pass}_{1,2}$	$n_{1,2}$
$Co_2$	$C^{act}_2$	$C^{pass}_{2,1}$	$n_{2,1}$
		$C^{pass}_{2,2}$	$n_{2,2}$

Table 5: Active component tasks operation invocations description

The WCET of a task  $i$  is, then, given by the addition of all its operation invocations of an active component has to be calculated because it depends on the operations that it invokes. Equation (14) shows the calculation of the WCET of task  $\tau_i$  based on its behavior and the characteristics of the passive components.

$$C_i = C_i^{act} + \sum_1^k (C_{j,l}^{pass} * n_{j,l}) \quad (14)$$

For schedulability analysis it is still needed to know the timing characteristics of the active component task  $\tau_i$ . The description of the active components is done through the characterization of the periodic execution of the component's task. Basic parameters of a task in a RMS scheduling pattern are:

$T_i$ : the periodicity of the task  $i$

$D_i$ : the deadline of task  $i$

$C_i$ : the Worst Case Execution Time (WCET) of task  $i$  (given by Equation (14))

$P_i$  the priority of the task  $i$  given by

### **Schedulability of the system**

Schedulability analysis can, then, be applied following RTA with the available information. The system is composed of tasks, every task with a WCET, calculated with the Equation (14), a given periodicity ( $T$ ) and a deadline ( $D$ ). The possible delays provoked by the invocation of shared passive components ( $B_i$ ) have to be included in the response time calculation. Lehoczky et al. work [23] based schedulability can be applied:

$$R_i = C_i + B_i + \sum_{j=hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (15)$$

Where

$R_i$  is the response time of task  $i$

$C_i$  is the WCET of task  $i$ , from Equation (14)

$B_i$  is the time the task  $i$  can be delayed by the invocation of a shared passive component operation, from Equation (13)

$T_j$  is the period of task  $j$

$C_j$  is the WCET of task  $j$ , that is of higher priority than task  $i$

$hp(i)$  are the tasks with higher priority than task  $i$

The task is schedulable if  $R_i \leq D_i$ . Equation (15) is applied recursively until  $R_i$  converges. If it does not converge then it is not schedulable.

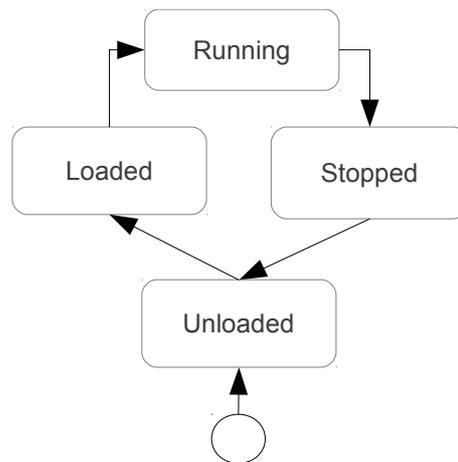
As a summary, the system consists of components of two types:

- *Passive components*: Offer operations that can be invoked by other components. From this point components are supposed to be reentrant by active components. This means the delays in invocations ( $B_i$ ) are not contemplated in Equation (15). This is done for simplification as it does not affect the rest of the calculations.
- *Active components*: Offer a functionality that executes periodically with real-time characteristics. Active components may invoke operations of passive components. The number of invocations of passive operations is not limited.

### **Component Life cycle**

The basic life cycle of a component in the system is as shown in Figure 7, with the following states:

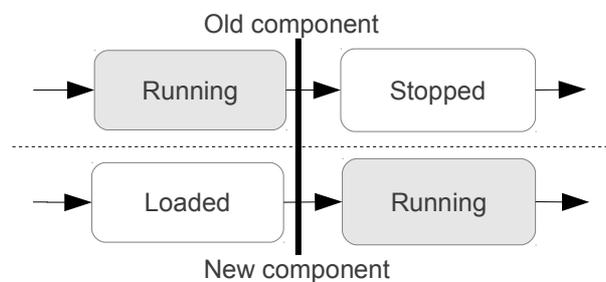
1. *Unloaded*: The component is still out of the main memory and the information about this component is not known by the system. The component will return to this state after completing its life cycle.
2. *Loaded*: The component has been transferred to main memory and is prepared to be executed. This state will be split in different sub-states in the next chapters according to the needs of the framework.
3. *Running*: The component has been accepted, prepared to be executed and is currently in execution in the system. An active component has its thread executed according to its timing characteristics. The operations of a passive component are being invoked.
4. *Stopped*: The component has been stopped, and it will not execute any more in the system. Its code is still loaded in the system. From this state, the component will only return to the initial state of *Unloaded* after unloading its code and its information.



**Figure 7:** Basic life cycle of a component

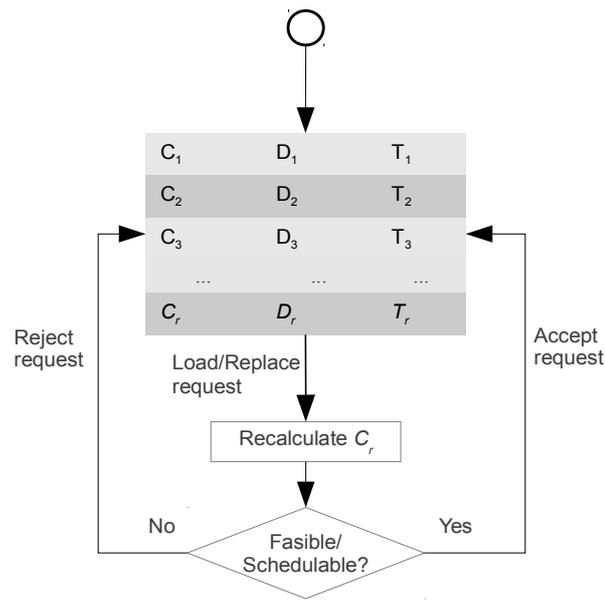
As stated above, these states will be refined in next chapters according to the framework needs. Different sub-states will be performed by different actors. For instance, the code of the component will be loaded by the OSGi platform, while the other actions will be performed by the implemented framework.

For the replacement of components, the state changes in their life cycles will be synchronized. This means that while the old component to be replaced is being stopped, the new component will be started in a synchronized manner, as shown in Figure 8. The timing in the reconfiguration process is critical, so this is the main target of this work.



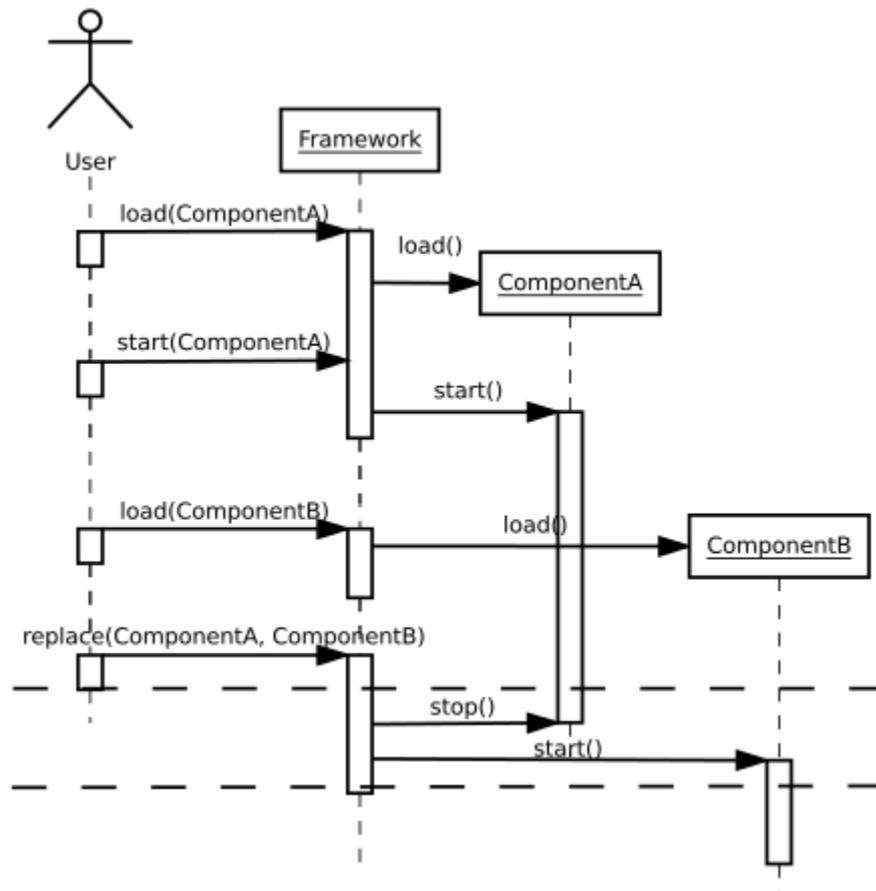
**Figure 8:** Component replacement overview

Loading and replacing components requires an acceptance test. Beside the processor time for the new component, time has to be reserved for the replacement task to maintain the QoS during the replacements. Loads and replacements will be requested during the execution of the system. If the new component to be loaded or replaced is schedulable in the system and the modifications in the schedulability of the replacement task are also feasible then the load or replacement is accepted (see Figure 9).



**Figure 9:** Load and replacement requests acceptance

A sequence representing an example of component loading and replacement is show in Figure 10. The system receives requests from the user. The user requests loading and starting ComponentA. The system performs the operations on behalf of the user. The user requests to load a new component (ComponentB) and that the first component be replaced by the second. The system performs the required operations. In the case of the replacement, the required operations are enclosed in broken lines. This represent a critical operation where the execution of the application must not be interrupted. The replacement of the component must not be noticed.



**Figure 10:** Example of load and replacement of component

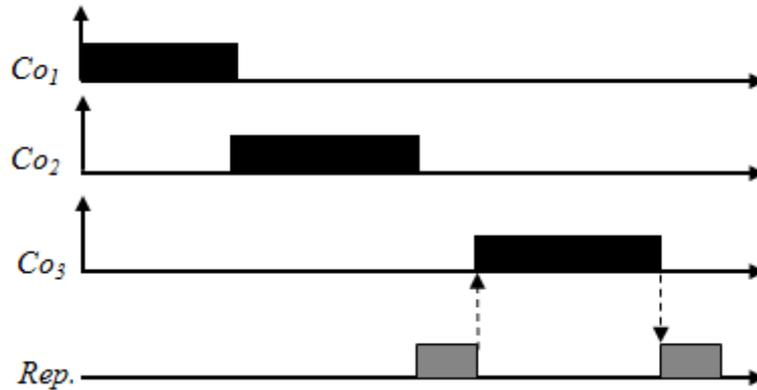
### 3.2 Replacement problem description and alternatives

Dynamic systems have requirements for on-line changes. These may imply on-line replacements, which require on-line schedulability analysis. Therefore real-time systems are sensitive to the time taken by the schedulability analysis. Any change in the system at run-time can make it not schedulable. Some approaches to provide dynamic characteristics to the system like loading, unloading and reconfiguring components have been described in Chapter 2 .

In the presented framework, extra tasks are included for performing the operations needed for component replacement as loading and unloading. The following are the basic building blocks of the proposed approach:

- *Use of framework tasks* to load, unload, and reconfigure components. If a background priority task is used to replace a component then this task can be preempted by a higher priority task or component. The state of the system would render then unstable. The state of the component may be only partially transferred to the new instance or only some of the component bindings updated. Figure 11 shows an erroneous execution of a component replacement where a component interrupts the management

task that is in charge of performing the component replacement operations. The framework task has a lower priority and will be interrupted by any other higher priority task. Execution of `Component 3` breaks the execution of the replacement. In this erroneous execution the new component instance may be in use, but the state may not be completely copied or bindings may not be completely reconfigured.



**Figure 11:** Erroneous replacement execution. Replacement task interrupted by component execution.

- A *slack time server* can be useful reserving some CPU time for system management tasks. Some works try to ensure that the incoming aperiodic task can be executed or rejected [4][5]. But these works do not guarantee that some tasks that need to be executed will, in fact, execute. A component replacement can be delayed but not rejected if it is required, for instance, to correct a component that failed. The use of slack time is not valid when the management task requires that its execution is not interrupted. The slack time server does not guarantee that enough time will be available to complete the task without being interrupted.

Time servers are not designed either to assure that these sporadic tasks are executed atomically nor in time when they have a hard deadline [6]. Time servers also reduce CPU time for mission tasks as they usually execute at highest priority. Schedulability analysis must take these sporadic tasks into account. This can be done in several ways. One of these ways is to reserve a limited amount of resources for such tasks.

- *Raising the priority of the management tasks* avoids that they are interrupted. Their assigned execution time still has to be adjusted so that they have enough time to complete their job. For example, a component replacement must not be stopped or interrupted. Else, they have the same problem as in the previous alternative.

The alternative proposed in this work takes into account all of these problems. It is possible to apply safe component replacements as described in the following chapters. This safe replacement is applied without interrupting the component's execution, which ensures a correct execution of the system.

The complete component replacement process is split in three different tasks for loading,

replacing, and unloading components. Every task will perform its corresponding operations. This separation of tasks is due to the following reasons:

- The *loading* and *unloading* tasks are not time bounded. So they execute with background priority without affecting the rest of mission tasks in the system. These tasks must also be able to load or unload components independently of component replacements.
- The *replacement* task has very strict timing requirements to ensure the correct replacement of components. This requires the replacement task to be scheduled according to the characteristics described below.

After a component replacement has been requested, the three tasks are executed in an ordered manner (loading, replacement, unloading) with the corresponding scheduling requirements. While several different load and unload tasks may execute concurrently, the replacement task will execute periodically at predefined instants to perform component replacements sequentially.

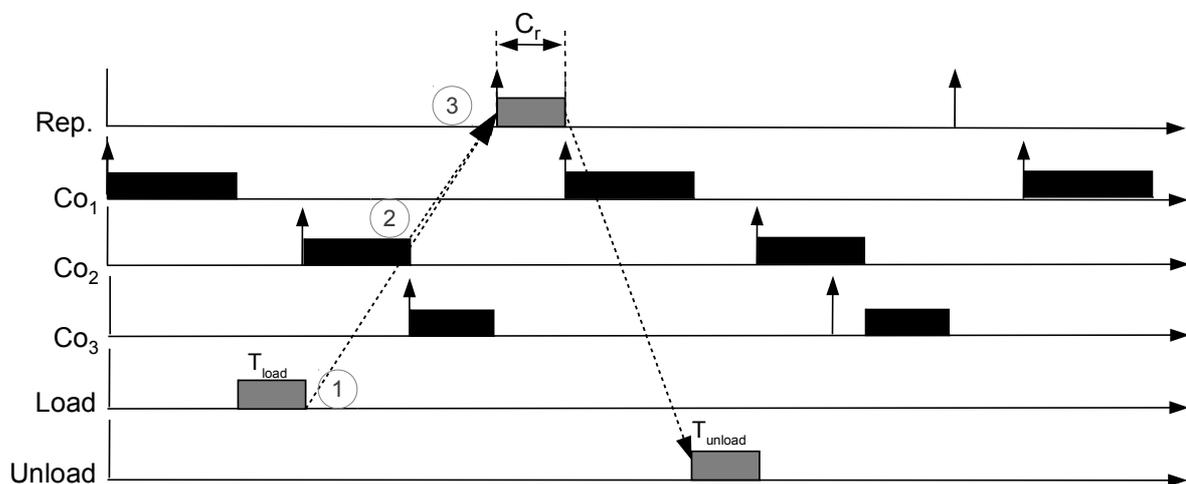


Figure 12: Component  $Co_2$  replacement

As described before, a periodic component replacement task is added to the system to assure that component replacements are correctly performed in time. Figure 12 shows an example of component replacement. The figure includes three running components ( $Co_1$ ,  $Co_2$ ,  $Co_3$ ), and the framework tasks (*Loading*, *Unloading* and *Replacement*). The replacement task can be released after the occurrence of some specified events, marked in the figure:

1. First, the new component is *loaded*. A low level priority is assigned to this framework task as it can be interrupted and the operating system is in charge of making it to not interrupt mission tasks.
2. Then, the component to be replaced has *finished its execution*, it must not start its execution again before the component replacement is performed. Then, the component replacement task is executed with maximum priority. In Figure 12, the component  $Co_2$  is replaced by a new instance (i.e., a new version of the same component).
3. The replacement task is *released at specified instants* where no other mission tasks are

executing because the time slot is reserved for such task. This is considered a safe instant for a component replacement.

The load and unload tasks are executed in background priority as they can be interrupted. The replacement task *Rep.* is released only after the described three events occur. This component replacement task is added to the schedulability analysis. If the system is schedulable then the component replacements can be safely applied.

As noted here, load, unload and replacement task must execute at different time instants due to scheduling requirements. So these operations are considered different separated tasks. The whole replacement process cannot be performed by a single task.

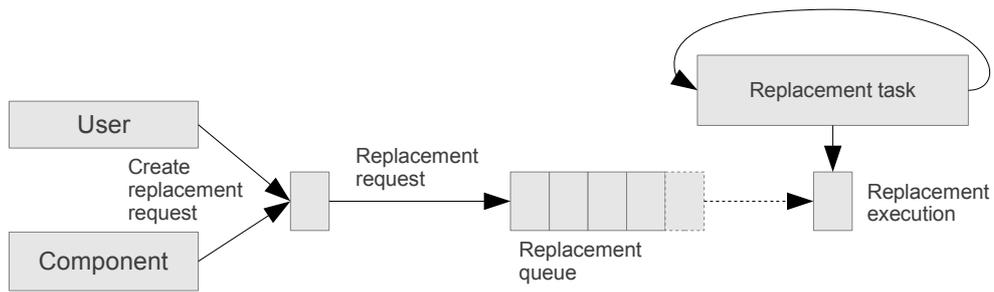
Many replacements can be requested at the same time, but a minimal inter-arrival time (MIT) is expected between replacement requests. This is due to the schedulability of the replacement task. While many loading and unloading tasks can run concurrently with background priority without interfering the execution of other tasks, the replacement task must execute timely with reserved processor time.

Making use of the CCSL semantics [144] of MARTE, some constraints can be applied to describe the timing requirements of the component replacement task:

- $T_{load}$  precedes  $Rep$     A replacement can only be performed after the new component has been loaded; when the corresponding loading task has finished its execution.
- $Rep$  precedes  $T_{unload}$     The old component can only be unloaded after the replacement has completed.
- $Co_2$  precedes  $Rep$     To be sure that the component is not running at the same time that the replacement task and it does not interrupt the component execution

Additional constraints will be detailed later in the description of the different provided replacement models.

The replacement task is supposed to work using a replacements queue (see Figure 13). The user or any other component may request a component replacement. Then, the request is placed in a queue. The replacement task is in charge of performing such replacements at the specified time instants. The replacement to perform is obtained from the queue. In case that the queue is empty, no action is performed by the replacement task.



**Figure 13:** Replacement task model

***Discussion and limitations***

A real-time component-based system is described here, with two different kind of components, their composition, connections, and temporal characterization. This system model is the base for the framework to be detailed in next chapters. The target of the framework is to provide the capability to safely replace components with real-time characteristics at run-time.

This solution is designed to attend a maximum number of replacement in an interval of time, supporting a minimum inter-arrival time of replacement requests. Replacement requests can be received at any time. Component loading and unloading tasks can be applied concurrently, but the replacement task can only be executed according to the described restrictions to ensure a safe time-bounded replacement.

The proposal has other limitations. One of them is the capability to replace only one component at a time. This replacement model is not designed to completely reconfigure an application if the structure of the application is modified, i.e., the number of connections between components change.



## Chapter 4

# Strategies and algorithms for dynamic real-time replacement of components

---

In this chapter, it is described a framework that enables run-time component replacement with real-time requirements. The framework contains: 1) A replacement model, 2) An on-line acceptance test for new components and replacements and 3) An method to calculate on-line the WCET value of new components. This model relies on the usage of framework tasks for loading, unloading, and replacing components.

In general, the execution time of some of the framework operations is not known a priori. For example, the time required to install a component depends mostly on the number of classes that it contains and their size. The time required for activating the component and loading of additional resources (as files, images, ...) besides classes is considered in this model by adding it to the loading time of the component classes.

There are some issues that have to be considered in the presence of dynamic behavior. Platforms that allow dynamic code downloads have unpredictable component installation times. It may happen that the installation time of a given component varies significantly depending on the moment when it is installed. This is due to the number and nature of the code, i.e., classes, that have to be instantiated and used, which varies during the lifetime of the system.

The problem in a dynamic system is that static offline analysis cannot be applied if the complete system configuration and possibilities are not known at design time. Some characteristics of the dynamic system considered here are:

- Components are added, removed, and replaced in the system at run-time.

- The time required for the framework task to complete replacements is not known until run-time, as it changes with every system update.
- The method provided in this work calculates the required time to execute the replacement task every time the system components are updated.
- It is assumed that replacements occur at safe points. This is ensured by the provided calculations.

Examples of calculation of computation time of components are provided in [9] and [14], where calculation of WCET is performed at design time. In a dynamic environment where components are added and removed at run-time, the WCET needs to be calculated during the system execution.

This work describes a model for processor time reservation for scheduling component replacements. It has a low-bound requirements, using the minimum time required so that the minimum time is lost when no component replacements are needed. The time required for different components in the system to be replaced is shared; a limited number of replacements can be performed at a given instant of time, but resources assigned are also adjusted, implying a minimal impact in the schedulability of the rest of system tasks.

The framework has to fulfill its main target, that is to provide the capability to replace components at run-time. The characteristics of this framework are the following:

- Replacements only take place at safe execution points to keeping the real-time schedulability of active components.
- Transparent real-time component replacement. The schedulability of the system must not be affected and running threads do not have to be aware of component replacements.
- Existence of framework tasks to perform management operations. Besides the component replacement itself more tasks are necessary in the framework to undertake a component replacement. These operations are loading, instantiating, destroying and unloading components.
- Generality. The component model used is generic enough to be usable in any scenario but also incorporates parameters to support the framework tasks, included the component replacement.
- Simple schedulability model. Complex schedulability models, i.e., multiple threads in a component, derive in complex schedulability analysis that require large computation times that are not applicable on-line. Therefore, we restrict the model while making it usable and analyzable.
- Single threaded components. Active components only have one task.
- Calculation of the WCET of active components. This calculation includes time of calling passive operations because it depends on the invoked methods of passive components. The framework will calculate, at run-time, the WCET of active components to determine

their schedulability.

- Calculation of the replacement time. It is also provided a way to calculate the time required to correctly undertake a component replacement while assuring the schedulability of the system. Component characteristics, as component bindings, are considered to perform such calculation. Computation times of component operations are provided when the component is loaded.
- Reentrant passive components. Passive components are not shared by active components or their operations are considered reentrant. This simplifies the the response time calculation for the schedulability analysis, while the model and calculations are still valid.

### 4.1 Component model

Component composition greatly depends on the type of component. For instance, components are usually connected through channels whereas services typically bind to other services through some data flow link [27] or service operations invocation. According to [28][29] components can be classified as *port-based* or *operation-based* depending on the component interface and their bindings.

- In *port-based* models, data is pushed through a connection from one component to another. The execution of the component can be started by a trigger sent by another component or synchronously through a clock.
- *Operation-based* model relies on the use of interfaces. Interface operations execution is usually started by the invocation of such operations from another component. For instance, in Java this means a method call or invocation.

Components used in this work are operation-based. They make available a set of operations to be used by other components to provide higher level functionality.

#### *Component elements*

Only the essential elements for a component to work in a real-time environment are considered in this component model. The following are the required component elements that are considered in the framework:

- **State:** The state of a component is defined by the internal values of the component attributes that are the working data for the component itself. The state can be modified due to different reasons, i.e., the execution of a functionality of the component through an external request.
- **Bindings:** They are the connections or links between components; a binding connects two components in a way that one component can make use of the operations of the other component through its interfaces. For simplicity only the term binding will be used in the rest of this work.

- **Timing attributes:** Components have a temporal model that contains a set of parameters related to the temporal execution and behavior that enable the schedulability analysis of the system and component replacements.

The two elements that are directly affected during a component replacement are the bindings and the state of the replaced component:

- The *bindings* between the components are a key element to be managed during a replacement. Such bindings must be set up prior to the component start up and execution. Likewise, bindings must be readjusted in the event of a component replacement.
- The *state* is the other element of the component that can potentially be modified in a replacement [18]. The goal is to achieve a real-time component replacement schedulability framework transparently, i.e., the system execution will not be affected by these updates. As a consequence, if the component has a state then, this state has to be transferred to the new component so that the latter can continue execution in a transparent way for the system.

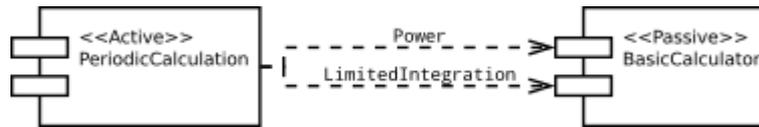
For this reason, the presented component-replacement schedulability framework comprises the component elements, their timing parameters, and a model for determining the replacement time bound.

Regarding the real-time execution nature of components, they must be separated in two different categories: *passive* and *active components*.

- *Passive components* only contain data and invocable code in the form of operations. No active execution entity (i.e., thread or task) is contained in a passive component. The worst case execution time (WCET) of every operation is considered for timing analysis purposes.
- *Active components* are such that contain a real-time thread or task and can, therefore, execute autonomously. Active components can invoke operations of other components making use of their public interfaces.

In a dynamic environment where components are loaded, unloaded and replaced, the time to complete the execution of the thread code may vary constantly [16]. The execution conditions change at run-time and not only depending on the deployment of the component on different hardware platforms. The time required to complete the execution of the operations of other components also affects the execution time of the active component. This framework provides a method to calculate the total WCET of a component thread execution at run-time to be able to apply a schedulability analysis and an acceptance test for that component in the system.

An example of passive and active components is shown in Figure 14. There is an active component `PeriodicCalculation`, which makes calculations at specified time instants and makes use of a passive component name `BasicCalculator` which provides some arithmetic operations in its interface. `PeriodicCalculation` makes use of two operations in `BasicCalculator` for its own purposes.



**Figure 14:** Example of active active component making use of passive components operations

Table 6 shows the temporal information of the `BasicCalculator` passive component. Its interface contains two operations `Power` and `LimitedIntegration`, and their computation times are 2 and 3 ms respectively.

Table 7 presents the information corresponding to the active component `PeriodicCalculation`. The computation time of its real-time thread is 5 ms without including the time required by the invoked methods. It invokes the `Power` and the `LimitedIntegration` methods in the `BasicCalculator` component 4 and 1 times respectively.

Component	Operation	$C_l^{pass}$ (ms)
BasicCalculator	Power	2
	LimitedIntegration	3

**Table 6:** Example of computation time of passive component operations

Component	Computation Time (ms)	Operation	Invocation Times (n <sub>l</sub> )
PeriodicCalculation	5	Power	4
		LimitedIntegration	1

**Table 7:** Example of computation information of active component

With the information provided in these tables, the WCET of the active component can be calculated. As an example, Equation (14) is applied here to obtain the following result:

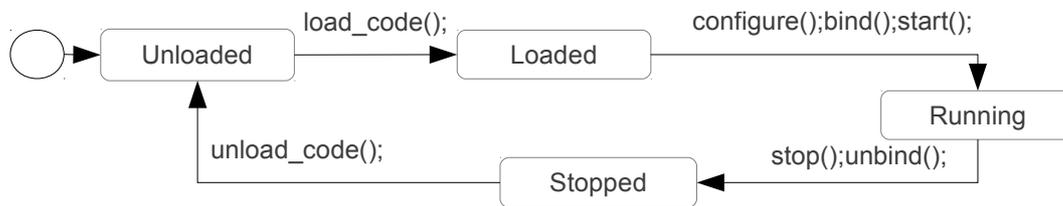
$C_{PeriodicCalculator} = C_{PeriodicCalculator}^{act} + \sum_{l=1}^n (C_l^{pass} * n_l) = 5 + (4 * 2 + 3 * 1) = 14 ms$ . No execution delays are contemplated in this example.

### Component life cycle

During their execution life time, components undergo different states as represented in Figure 15. Every change of state is achieved performing several operations on the component:

- *Unloaded*: The code has not been loaded to main memory. Temporal information about the component is still unknown.

- *Loaded*: The component code is put into the main memory and it becomes ready to execute, but not yet executed nor included in the scheduler. A component is loaded when a new component load or replacement is requested. A user or any other running component may request the load or replacement of a component.
- *Running*: The operations to perform before the component starts to run are to configure (i.e., setting up initial values), and to bind the component to existing components. The execution of the component can be started after performing those operations. The difference between active and passive components is that active components contain an execution thread that is started while passive components only receive operation invocations.
- *Stopped*: The component is stopped, bindings are eliminated and it stops being available for other components. The code is remains in memory. From here the component can be unloaded to return to the initial state.



**Figure 15:** Component life cycle

The state of a component when it is *stopped* is equivalent to its state when the component code has just been *loaded*. It has been differentiated to represent the complete life cycle of the component. The other reason to represent it as a different state is that once components are stopped components are supposed to not be started again. The transition from *stopped* to *running* is not considered here. This is the case in the Java specification where a stopped thread cannot be started again, as described later in Chapter 5 .

In the next section, the different framework tasks in charge of performing these operations are described. The operations and states described here are also refined according to the framework needs.

***Real-time properties***

Temporal properties are assigned to active components. Each component is executed by a real-time thread. This idea simplifies the model while being, at the same time, realistic. As an example, this can be the case of a software component that executes the sampling of some sensor to gather environmental data; it is a simple code provided in a self-contained unit as a component. Nevertheless, multiple components can still be composed to create more complex behaviors. Therefore every component only requires the timing parameters corresponding to a single real-time thread:

- $C_i$  or worst case execution time of component  $i$ .

- $T_i$  or activation period of component  $i$ .
- $D_i$  or execution deadline for component  $i$  at every activation period.
- $P_i$  or priority of the component  $i$ .

Where  $i$  indicates the number of task or component.

Since we consider components that are activated periodically, we use the cyclic task model [28] and rate monotonic scheduling (RMS) [29] for the temporal analysis of the system replacements. It would also be possible to consider earliest deadline first (EDF) [28], but the schedulability model would add some extra complexity if the necessary restrictions are not kept [30].

## 4.2 Framework tasks and their operations

To support the replacement of real-time components at run-time, a set of basic operations are provided by the framework. The framework contains three tasks in charge of performing such basic operations. This framework implements the following tasks, of which only the component replacement task needs to be implemented, with isolation guarantees, as an atomic operation:

- *Loading task*: The component code has to be placed in memory in order to be used. It is usually brought from a secondary memory or even a remote server. The component requires some previous steps before starting to work. Once the component is loaded into memory, it may need to be instantiated, configured, its state and bindings initialized and its thread, in case of an active component, started.
- *Unloading task*: Once the component is not needed in the system its thread has to be stopped and its resources freed. These resources are memory and CPU time. Active components are unscheduled. The code of the component is cleaned from the memory because it is not going to be used any more in the life of the system. It is differentiated here the process of destroying a component and unloading a component. There is no need to unload a component if it is going to be used again in the system later.
- *Replacing a component*: Exchanges a component in execution with a new instance that will execute in its place. The replacement is applied seamlessly for the rest of the system. This task makes use of the same operations than the loading and unloading tasks, i.e., to rebind, start, and stop components..

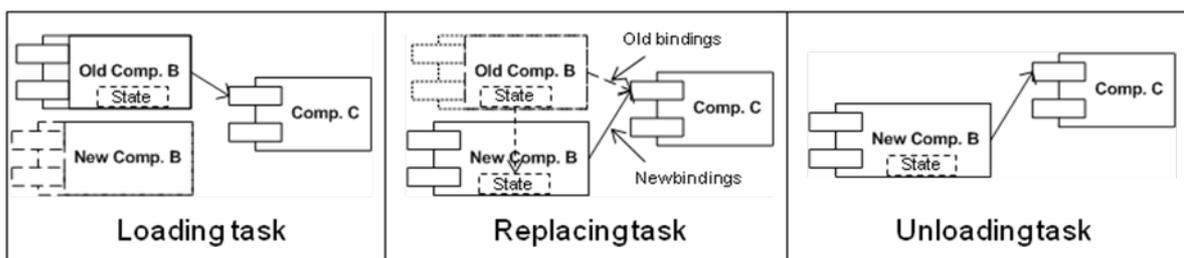


Figure 16: Framework tasks overview

The framework provides an execution infrastructure for running the described tasks (see

Figure 16). These tasks are included in the schedulability analysis in order to receive enough resources to run without interfering the mission tasks, i.e., preventing the threads of the active components from meeting their deadlines.

Following, the different framework tasks and their operations are described in more detail.

### ***Loading task operations***

Here are detailed the operations needed since the component is loaded until it is finally started:

- ***Code loading:*** The code of the new component is stored in memory. This may require to download the code from a remote node or only to read it from a local secondary storage.
- ***Acceptance test:*** A schedulability analysis algorithm is applied to check if the system is feasible with the new component temporal parameters. The component will be rejected or accepted depending on the possibility for it to be accommodated in the system and its temporal properties preserved. Resources are reserved for the component if it is accepted, as processor time or memory for its execution.
- ***Component instantiation:*** The new component is created and initialized with default parameter values. For instance, in an object-oriented oriented language it represent creating a new instance and executing a constructor method.
- ***Binding:*** The new component is connected to already existing components by means of their interfaces.
- ***Start up:*** Component execution is started. An active component then starts its own execution thread. Passive component may receive incoming requests as soon as bindings are set up.

### ***Unloading task operations***

Component unloading requires to perform the following operations to eliminate a component from the system:

- ***Stopping:*** An active component stops its execution thread. Previously reserved resources are freed. Processor time reserved for the component is freed and available for other component executions.
- ***Unbinding:*** The component is disconnected from the rest of components of the system. Connections are cleared.
- ***Code unloading:*** the component code is unloaded from memory, dereferenced and the component is not reachable.

The time required to complete each of these operations is unknown. It depends on the size of the component code, number of bindings, etc. Different approaches can be used to solve this problem. One of them is to limit the processor time assigned to framework tasks to load or

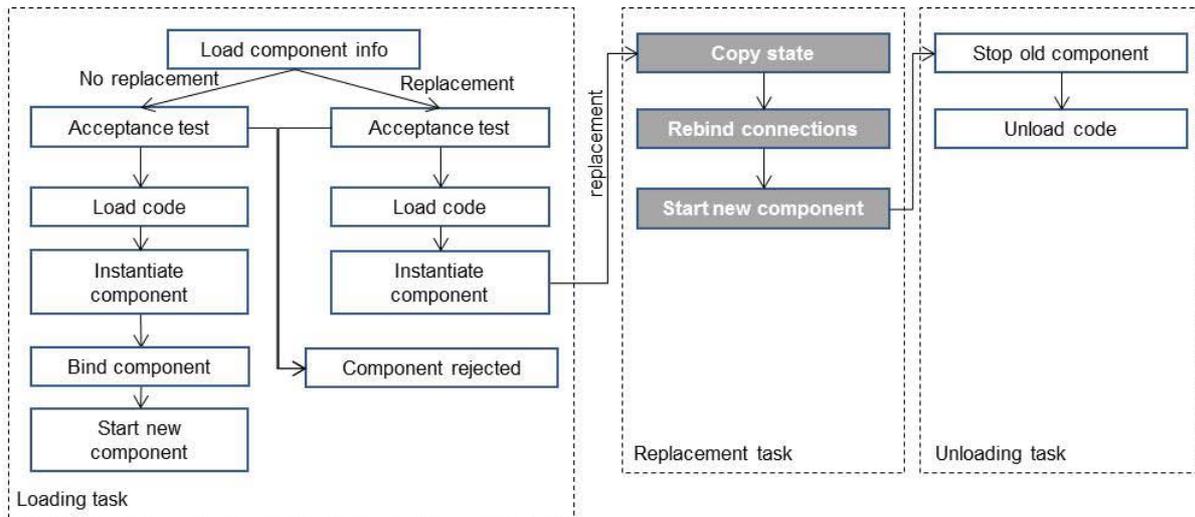
replace a component assigning them a deadline to complete their work so that the schedulability of running tasks is not affected. This can be implemented through operating system timers and processor usage accountability. In the approach presented here, loading and unloading tasks are supposed not to interfere with the execution of the mission tasks while running in background, so no deadline is imposed for them. The operating system is in charge of isolating the execution of these tasks so that, for instance, avoiding that the load of a component into memory interferes the execution of a running component. For this reason, components undergo an admission test in the system that determines whether the system is schedulable or not, i.e., determines if all components will meet their deadline.

The execution time of framework tasks is always included in the schedulability analysis. Component loading and unloading are not critical tasks. They are correctly managed by the operating system by setting their priority to the minimal priority available in the system. A component replacement is a critical task. So, it requires calculating its computation time to be correctly scheduled and not being interrupted.

### ***Replacement task operations***

Figure 17 summarizes the complete process phases for a component replacement. It requires the loading tasks been executed previously and the unloading task executed afterward. All the replacing task operations marked in gray are critical in the sense that the replacing task must not be interrupted. This task has to be executed atomically to avoid erroneous executions, i.e, no other concurrent operations are allowed while they are active. The timing of loading and unloading tasks does not affect the schedulability of the mission tasks (i.e., of the components). They have lower priority and are not assigned a real-time deadline.

Before making the actual component replacement an admission test has to be performed that determines whether the component reconfiguration is schedulable in the system. In this test, not only the real-time scheduling characteristics of the component have to be tested but also the scheduling characteristics for the replacement task to be able to replace this component.



**Figure 17:** Component Replacement Process

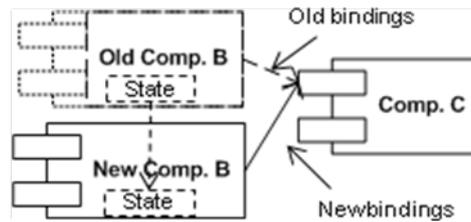
If the new component requires a replacement then it has to be taken into account. If the event that the required replacement time is greater than the time initially reserved for the execution of its replacements, the new component will have to be rejected if not enough CPU time can be reserved for such replacement.

If enough time can be reserved for the new component task and its replacement, then the component replacement is accepted. This acceptance test is described later in this section.

It can be noticed in Figure 17 that the behavior of the loading task changes depending on whether the component is going to replace a previously existing one or if a new component is going to be added to the platform. In case of a component load with the purpose of replacing a running component, the new component will be started after the replacement.

Supposing a rate monotonic scheduling system, the available processor time to accomplish the component update can be known. The free processor time for the update would start when the task execution is completed until the task is scheduled again. This free time has to be enough for the component update to be accomplished. This will also affect the rest of the tasks in the system with lower priority. But it would be interesting to schedule this component update in a way that the smallest possible number of components are affected. Two different replacement models are proposed for this purpose in the next section.

When components are created or stopped, creating new bindings and eliminating them may not be as critical as changing component connections at run-time as these changes may alter the component execution and result in unreliable outputs or failures due to deadline misses. Beside the time-budget required for components to execute, it is required to calculate the time needed for a component to be replaced. A timely execution of this replacement is mandatory to keep the safety of the execution of the components avoiding execution interference. A wrong replacement execution produces functional and non-functional errors in the system. Reserving resources for component replacement also allows the framework to provide a QoS assurance during the process.



**Figure 18:** Component replacement

Replacing a component consists of two main steps (see Figure 18):

- *State transfer*: Copy the state from the old component to the new component.
- *Rebinding*: Replace the bindings of the component.

Aspects to take into account in rebinding are:

- *The passive or active nature of components*; Passive components are components without an execution thread, so they do not have direct influence in schedulability analysis. However, their operations have a temporal effect in the active component's execution time, and hence in the overall schedulability.
- *Number of connected components*; this directly influences the time taken to rebind those components. Usually, this time is linearly proportional to the number of bindings since it is calculated as the time required to change a binding multiplied by the number of bindings (  $C_{rebinding} = C_{binding} * n_{bindings}$  ).

Following, we provide the basic parameters of the framework that allow calculating the time to update the binding of a component.

Let us suppose a group of components that are released for execution at the same instant  $t_0$ ; also, the component replacement task starts its execution at  $t_0$  with an execution time of  $C_r$ . Then, the replacement task will have to complete its mission (i.e., stop the old component, start and bind the new component) before the component is scheduled for its next execution. This will be its deadline  $D_r$ .

The replacement process will be schedulable if it completes before the given deadline  $D_r$ . If the schedulability analysis renders feasible adding a new task with a WCET of  $C_r$  then the rebinding task is feasible without affecting the execution of the involved components. The execution period of such task ( $T_r$ ) will be calculated later.

The calculation of the time required to accomplish the component replacement can be done similarly to the work in [132] [133]. Rasche et al. propose to block the execution of the application or the affected component, load the new components, reconfigure the application, and unload the unused components. Then the application is restarted:

$$C_{repl} = C_{load} + C_r + C_{unload} \quad (16)$$

where:

$C_{repl}$  – total time to replace a component, including the loading of the new component and the

unloading of the old component

$C_{load}$  – time to load and initialize a new component

$C_r$  – time for replacement – time required to exchange the old component for the new one, transferring the state and rebinding the components

$C_{unload}$  – time to stop and uninstall the old component

Loading and unloading tasks ( $C_{load}$  and  $C_{unload}$ , respectively) are already described in the previous section and they do not need a deadline because their execution can be correctly handled assigning them the lowest priority in the system.

In this model, the execution of the component replacement task is not to be interrupted. Time where components are not working is used to replace bindings. Proposals like [132] are based on halting application at run-time for reconfiguration. But in a real-time environment, blocking the application execution is not acceptable. Deadlines of component threads would not be met.

In this work, two alternative strategies to avoid blocking the execution of components are proposed to overcome the described problem. While  $C_{load}$  and  $C_{unload}$  times does not need to be bounded,  $C_r$  requires reservation of processor capacity to ensure that the component execution is not disturbed. The number of bindings is given by the number of dependencies of the component plus the number of running components that depend on the one to be replaced. Analyzing the minimum time required to accomplish the component replacement can be done based on:

$$C_r = C_{state} + n * C_{bind} + C_{start} + C_{stop} \quad (17)$$

Where:

$C_r$  – is the total time required for the component replacement

$C_{state}$  – is time required to transfer the state from the old component to the new one

$C_{bind}$  – is the time to create and setup every binding (being  $n$  the total number of bindings of the component)

$C_{start}$  – is the time required to start the execution of the new component

$C_{stop}$  – is the time required to stop the execution of the old component

The time required to transfer the state of the component can be previously known or calculated from analysis of the code based on the component's size and structure. The time required to update every binding can also be known previously by the platform. Therefore, following this component replacement protocol, the complete time required to accomplish the critical section of the reconfiguration is bounded and known. Given that the execution of the component replacement task is not to be interrupted, time where components are not working is used to reconfigure bindings.

The target of this model is to keep a *replacement task* in charge of performing component replacements in a safe way. The framework reserves processor capacity for the replacement task to ensure it is not disturbed. This means that the task execution is isolated for execution interferences of other tasks from the framework itself and from the mission tasks.

For the replacement task to be executed and not disturbed, it has to execute with the highest priority to not affect the schedulability of mission tasks in the system. Enough CPU time has to be reserved for the replacement task to execute without schedulability issues.

We assume that component replacements take place at safe execution points. Such instants in time were described generically in the previous chapter. Specific safe times are also described in the next section for every proposed replacement strategy.

The available CPU time to accomplish the component replacement can be bounded. For a periodic task, the free processor time for a replacement would start when the task execution is completed until the task is scheduled again. The computation time of the task ( $C_i$ ) plus the time for its replacement ( $C_i^r$ ) must not be greater than the period of the task ( $T_i$ ) for it to be schedulable. Equation (18) represents this restriction.

$$C_i + C_i^r \leq T_i \quad (18)$$

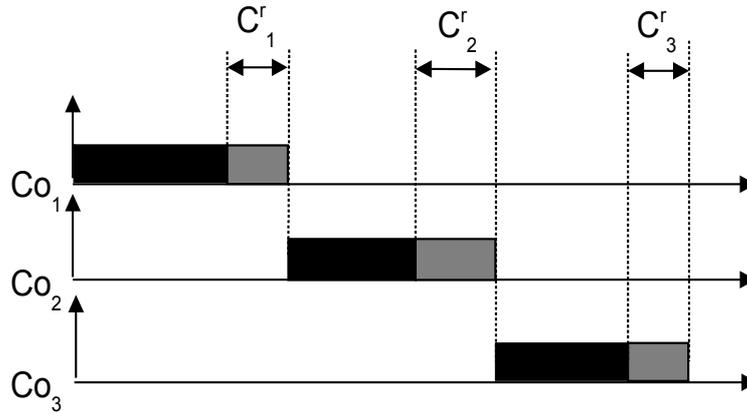
### **4.3 Strategies for component replacement**

Two different strategies are proposed here to accomplish component replacements depending on the available processor time and the type of system. The first strategy for each component, a replacement time is provisioned for each period (i.e., for each execution). This approach has been named *pessimistic replacement* model.

The other model has been named *selective replacement*. This kind of replacement model allows to configure the number of replacements that can be applied during a specified time interval. This time application dependent.

#### ***Pessimistic replacement***

Reserving processor bandwidth for every task  $i$  so that a replacement is guaranteed for every execution requires to provision additional time  $C_i^r$  for such replacement as part of the total execution time of the task,  $C_i + C_i^r$  in a way the task execution deadline  $D_i$  is not affected. If an update is requested by the task, it takes place right after the normal execution of the task (after  $C_i$ ), and before it is activated again ( $T_i$ ). The priority assigned to the replacement task is the same of the thread of the component to be replaced ( $P_i^r = P_i$ ) to assure that the update is performed in time.



**Figure 19:** Independent time reservation for component replacement

Applying semantics given by the CCSL of MARTE, the timing constraints of this replacement model can be given by  $C_i \text{ alternatesWith } Co_i$ . It indicates that the replacement task for component  $i$  ( $C_i^r$ ) necessarily executes after every execution of the component  $Co_i$ . It executes with the same periodicity, but the execution time of the replacement does not affect the execution the component and its deadline. No other constraints apply between replacements or tasks here.

To calculate the consequences of the *replacement task* on the schedulability of the whole system, the exact response time analysis of Lehoczky et al. [31] is applied. This test calculates the exact response time of every task of a system taking into account the interference generated by higher priority tasks. The formula provided by Lehoczky et al. is the following:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad (19)$$

Equation (19) adds the cost of the analyzed task  $i$  plus the cost of the  $n$  tasks with higher priority, which are executed in the  $R_i$  time interval. The computational cost and activation period of higher priority tasks are indicated by  $C_j$  and  $T_j$ , respectively. The response time,  $R_i$ , is then calculated with Equation (19) in an iterative way until the resulting value converges, i.e.,  $R_i$  has the same value at the left and right side. If it converges for every task in the task set, then this task set is schedulable. This equation has to be applied in order, from the task with the highest priority to the one with the lowest priority.

In the pessimistic model, the time to replace component at every execution is considered. Then a modification of Equation (19) is proposed here to calculate the interference of the time required to replace every component after its execution. Then, the response time analysis of the task set would be given by Equation (20) as follows:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (C_j + C_j^r) \quad (20)$$

Equation (20) represents the response time of task  $i$  where the time requiring to replace every

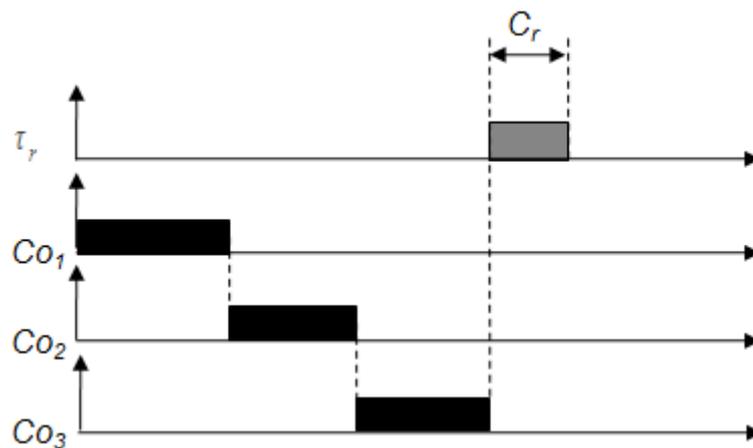
component ( $C_j'$ ) is added to calculate its interference in lower priority tasks. Replacement time for task  $i$  is not included because it does not affect the deadline of these tasks whereas the replacement time of upper priority tasks does affect the response time of this task as the replacement code execution has the same priority as the corresponding task.

Figure 19 shows the execution sequence in the described component model by reserving a time slot for replacing every component at the end of every execution. This model is given by Equation (20). As observed in this figure, it is pessimistic in case that the component replacements are not applied at every task execution; if this happens, not all the reserved time for replacements,  $Co_{r1}+Co_{r2}+Co_{r3}$ , will be consumed.

Hard real-time applications with high need for continuous of immediate component replacements may apply this replacement model. This model requires additional processor time. It usually represents increasing the CPU power of the embedded system, and so, increasing the costs. But being able to perform system updates by applying component replacements at run-time without stopping the system is usually preferred. The cost of stopping and embedded real-time system to be updated can be enormous compared to the cost of using processors with higher capacity.

**Selective replacement**

Although it is very safe, the above model may represent a considerable waste in reserved CPU time in case that the component replacements are not applied at every task execution. So, an alternative component replacement model is proposed, which represents a compromise solution. A smaller CPU time can be reserved but reducing the number of component replacements that can be applied. The time reserved for component replacements may be shared with other framework tasks to make use of free CPU time when no component replacements are requested. But such possibility is not contemplated in this work.



**Figure 20:** Shared time reservation for component replacement

The properties of this task will be updated according to the modifications of the system. In the pessimistic model each component has its own replacement characteristics. These characteristics

will not change during the life of the component. In the selective replacement model those characteristics are grouped in a single task. The characteristics of the task will have to be recalculated for every modification of the system. The methods to calculate such properties are also described here. The following is the list of modifications that affect the parameters of the replacement task:

- **Addition of a new component:** The WCET of the replacement task is modified. If the time required to replace the new component is greater than the actual WCET of the replacement task then the WCET of the replacement task is increased accordingly. The execution period may also be modified proportionally if it is based on a CPU utilization factor, as described later in this chapter.
- **Removal of a component:** The WCET and period are also modified. The largest computation time for a replacement is used as the WCET of the replacement task.
- **Replacement of a component:** As in the previous two cases the WCET and period are modified accordingly to the properties of the old and new components in the replacement.

The selective replacement model relies on the existence of a single framework task named “replacement task  $\tau_r$ ”. This task centralizes all replacements of all components. The effect the CPU time reserved for this task in the schedulability analysis has to be considered. This new task is added to the whole task set, and is executed with a higher priority than mission tasks so that it is not interrupted. If this task is interrupted then its deadline will not be fulfilled. The replacement task is described by a computation time ( $C_r$ ) that should be enough to perform any component replacement, a period of execution ( $P_r$ ), a deadline ( $D_r$ ) and a priority ( $P_r$ ). The priority of the replacement task is strictly higher than the priority of any component thread. Although it does not have to be higher than the priority of the operating system tasks ( $P_{OS}$ ) so,  $P_i < P_r \leq P_{OS}$ . Being  $P_{OS}$  the slowest priority of all operating system tasks. This is the case always since operating system tasks execute in privileged mode.

To guarantee that enough processor time is reserved in this thread, its computation time has to be equal to the largest time needed for any running component update (see Equation (21)).

$$C_r = \max(C_1^r \dots C_n^r) \quad (21)$$

The effect of the execution of this task with the highest priority in the schedulability analysis of the task set is as follows:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil C_j + I_r \quad (22)$$

Where  $I_r$  represents the interference created by the reserved time for component's replacement as:

$$I_r = \left\lceil \frac{R_i}{T_r} \right\rceil C_r \quad (23)$$

From the start of the system, a time server for component's replacement is added with a  $T_r$  period, a  $C_r$  worst execution time, and a deadline  $D_r = C_r$ .

Figure 20 shows the shared replacement task  $\tau_r$ . This replacement model is given by Equation (22). Only one replacement can be applied when this task is scheduled. Considerable time may be gained for component tasks and the number of component replacements in the system can be tuned modifying the activation period of task  $\tau_r$ .

Soft real-time applications are greatly benefited from the use of this replacement model. As described before, an large amount of time is gained with the selective replacement model compared to the pessimistic model. Component replacements does not need to be applied immediately. If several replacements are requested at the same time, they can be selectively applied in order according to the application priorities. The priorities for component replacements and the period of execution of the replacement task are, then, application dependent.

The period of the new task ( $\tau_r$ ) will be determined by the CPU time reservation or desired Minimum Interarrival Time (MIT) of the component updates. This time depends on application-based requirements. Two ways to calculate  $T_r$  are proposed here.

Processor time reserved for component replacements can also be shared with other management tasks. When no component replacements are performed, this free processor time will be used by other tasks with background priority. Background priority tasks are expected to execute on free processor time. A percentage of processor time ( $U$ ) can be assigned for all of these tasks (including the replacement task). Then, this task's period is given by:

$$T_r = \frac{100 * C_r}{U} \quad (24)$$

On the other hand, a minimum number of updates can be expected in the platform at run-time during a specified time. This number will be given by the MIT of component replacements in the system. This *minimum inter-arrival time* will directly determine the period of this task. If we want to reserve enough processor time for a number of replacements,  $n_{replacements}$ , in a fixed period of time,  $t_{period}$ , then the minimum inter-arrival time, and hence the period of the replacement task, will be given by:

$$T_r = \frac{t_{period}}{n_{replacements}} \quad (25)$$

Different values in task periods in the system generate interferences in the execution sequence of such tasks making the 100% of CPU time to not be reached or not schedulable execution sequences. A generalized technique to avoid this problem is the harmonization of such periods. This means adjusting the period value of different tasks that have a close period. This reduces execution interferences between different tasks and simplifies the schedulability analysis.

Once a period value ( $T_r$ ) is obtained, it should be *harmonized*. Harmonizing the period of the replacement task means adjusting the value so that it matches the period of existing tasks in the system or its harmonic values.

Multimedia applications can apply this method without problems. Usually multimedia applications consist of several component working sequentially at the same execution period, i.e., rendering images at the same frame rate. The replacement task period can be adjusted to a multiple value of the period used by the multimedia components. This greatly simplifies the schedulability analysis and reduces the interferences generated by this task.

Again, the constraints language provided by MARTE can be used to represent the timing properties of the replacement task regarding the component tasks in the system. Additionally to the load and unload of the new and old component respectively, new constraints are added to the replacement process compared to the initial constraints set provided in Section 3.2:

$T_{load}$ precedes $C_r$	A replacement can only be performed after the new component has been loaded
$C_r$ precedes $T_{unload}$	The old component can only be unloaded after the replacement has completed
$C_i$ precedes $C_r$	To be sure that the component is not running at the same time that the replacement task and it does not interrupt the component execution
$T_r = \text{idealclock discretizedBy } \omega$	The period of the replacement task ( $T_r$ ) is given by calculated time $\omega$ .
$C_r = C_i \text{ sampledOn } T_r$	The replacement task executes after the execution of the component $i$ ( $C_i$ ) to be replaced, at fixed time instants given by $T_r$ .

Here it is noted a limitation of the Clock Constraint Specification Language of MARTE. In the last constraint the execution of the replacement task is referred to the execution of any component execution. No constraint is provided by MARTE to express that the execution of the replacement task is performed on a specific component request. Component requests arrive aperiodically and different components are requested to be replaced at every request. A scheduling analysis tool based on MARTE would not be able to correctly apply the schedulability analysis based on this generic constraints. Anyway these tools are oriented to off-line schedulability analysis and no support for run-time analysis is provided.

#### 4.4 WCET calculation

As described in the previous chapter, the WCET of every execution of active component threads depends on the computation time of the passive component operations invoked. The WCET is calculated in the following cases:

- *A new active component is loaded or replaced:* Its WCET is calculated making use of the passive components bonded to it.

- *A new passive component is loaded or replaced:* The WCET of all the active components bonded to it is calculated.

Computation time of component operations is provided together with the component when it is loaded. The computation time of the invoked operations ( $C_{j,l}^{pass}$ ) is added to the computation time of the active component thread ( $C_i^{act}$ ), taking into account the number of times that the operation is invoked ( $n_{j,l}$ ). The way to calculate the complete execution time of the active component thread can be expressed as:

$$C_i = C_i^{act} + \sum_{k=1}^m (C_{j,l}^{pass} * n_{j,l}) \quad (26)$$

Where:

- $C_i$  is the WCET of the active component thread
- $C_i^{act}$  is the computation time of the thread without including operation invocations
- $m$  is the number of invoked operations by the active component thread
- $C_{j,l}^{pass}$  is the computation time of the operation  $l$  in the passive component  $j$  invoked by the active component thread.
- $n_{j,l}$  is the number of times the operation  $Op_{j,l}$  (operation  $l$  in component  $j$ ) with computation time  $C_{j,l}^{pass}$  is invoked by the active component thread.

Operations of passive components can also make use of operations provided by other passive components. The WCET of operations can also be computed using the same equation. The computation time of the invoked operations is added to the computation time of the operation in the passive component itself.

Table 4 and Table 5 in Section 3.1 describe the information provided by components when loaded, for the framework to be able to calculate the WCET of every loaded component making use of Equation ((26)). This calculation will be performed for every component addition or replacement in the system.

#### **4.5 Replacement acceptance test**

The framework provides an acceptance test that determines if the addition of a new component to the system can take place without degrading the performance of the rest of components. For real-time components, degradation means that component execution deadlines are not met. In our model, an additional requirement is included that refers to the capability to replace components reserving processor time for the replacement task. For this reason, a schedulability test has to be passed based on equations described in section 4.3. Processor time required by the component replacement task is calculated using Equations (17) and (18). Depending on the replacement model used, Equations (20) or (22) are applied respectively for the pessimistic or selective replacement model. If the system is still schedulable after introducing those changes, then the component addition or replacement is accepted. Figure 21 shows the acceptance test for the replacement task.

```
TS = TS - {O}
TS = TS + {P}
Cpx = Cstart,p + np*Cbind + Cstop,p
Cr = max (C1x, ... Cpx, ... Cmx)
rc = calculate_RTA_schedulability(TS);
if (rc == true) then
    accept_component();
else
    reject_component();
    TS = TS - P;
    TS = TS + O;
end;
```

---

**Figure 21:** Acceptance test for replacement

Task set ( $TS$ ) is modified eliminating the old component task ( $O$ ) and adding the new component task ( $P$ ). The time required to update the new component is calculated ( $C_p^x$ ) and the replacement task modified accordingly. The schedulability test is then applied. The `calculate_RTA_schedulability()` method calculates the response-time ( $R_i$ ) of every task and compares it to the corresponding deadline ( $D_i$ ). If no task misses its deadline then the system is schedulable and the method returns `true`. Else it returns `false` indicating that the task set is not schedulable. If the test is passed the replacement is accepted, else the replacement is rejected and the task set restored to its previous state.

Accepting the component with the `accept_component()` method represents that the process described in the Figure 17 is performed. The loading task will continue performing its operations. The component code is loaded, the component is instantiated, its bindings configured and its execution started. In case of a replacement then the replacement task will continue the process when executed. If the component is rejected then its information is discarded and the loading or replacing process is aborted.

Usually, the number of connections of a component will not change. So, the time required for a component update will not change either. The difference in acceptance or rejection will mainly come from the worst case execution time of the component. If it is not greater than the one of the component to be replaced (and the number of bindings is the same) the algorithm could be simplified and the component update automatically accepted.

### 4.6 Discussion

Two model to replace real-time components have been provided in this chapter. The usability of these replacement models depends on the type of application on which they are going to be implemented. The pessimistic replacement model is designed for applications with immediate replacement needs at the cost of high processor time reservation for replacements. The selective

replacement model is more flexible. It allows to adjust the desired processor time for replacements. Then, replacements are not so immediate and have to be prioritized if more than one is requested at the same time.

The basic elements of these replacement models are:

- *The required time to replace a component.* It is based on the component model and its structure.
- *The replacement task and its characteristics.* Based on the replacement model and processor time reserved for replacements.

To be able to perform the replacements in the system, two other operations are also necessary:

- *The calculation of the WCET:* The computation time of components is not completely known until they, and all the components on which they depend, are loaded on memory. The WCET has to be calculated at run-time.
- *The replacement acceptance test:* Every new component has to pass an acceptance test. In this case the acceptance test also takes into account the replacement task, which is in charge of replacing running components.

Replacement process and its phases are designed to replace only one component at a time. Even y time is reserved to replace more than one component when the replacement tasks is executed, components are replaced independently.



# Chapter 5

## Framework specification

---

Due to the extended use of Java programming language, a number of component models have been elaborated for it. Beside component models, a Real Time Specification for Java (RTSJ)[13] has been proposed with the intention to bring real-time characteristics to the language, overcoming some important challenges to predictability as the use of a virtual machine and a memory garbage collector, which affect negatively the predictable execution of the real-time systems.

To support dynamic execution in a real-time system, a number of additional characteristics should be considered. Dynamic behavior need to be limited and controlled to achieve predictability at run-time. Predictability allows maintaining the system execution safe while allowing components to be replaced without affecting the execution of other running components.

For a component framework to provide dynamism not only component loading and unloading must be supported. A suitable component replacement strategy is also required. This strategy has to fulfill the following requirements to be implemented in the component model:

- The replacement task must not be interrupted while it is in execution. This task must run at the highest priority available in the system so that no component can force it to be preempted. That is one of the problems addressed in this work.
- The component model must enable the use of mechanisms to avoid concurrent execution of the component and the replacement process. Data would get corrupted or incoherent while being copied. This could lead to memory incoherency.
- The specification of a component must allow that component provide their temporal information to maintain the schedulability of components and the replacement task. The

execution of components depends on the rest of components in the system. The total time of a component to complete its execution depends on the time of other components to complete its operations. If the implementations of the running components can change at run-time, then, computation time of every component can only be calculated at run-time.

This chapter describes the implementation in Java of a framework that provides safe replacement of components at run-time while keeping the correct real-time execution of the running components in the system. It also specifies the operations and methods that allow the loading, unloading components at run-time. Loads and replacements are only accepted if the schedulability of the system and the capability for the new component to be replaced can be guaranteed. This component framework elaborates the component model described in the previous chapter.

This specification of a component is focused on three main targets:

- The specification of the component to be used in the framework: It has to integrate the needed characteristics to be correctly managed by the framework.
- The framework itself: The replacement task implementation description is remarked.
- The registry: To handle internally all of the components running in the system and their dependencies.
- The implementation of the framework tasks: It is described how the different required tasks as loading, unloading and replacing (see Figure 16 in Section 4.2) are implemented in the framework.

It is also shown the capability to provide real-time characteristics to a dynamic platform as OSGi. An integration of the framework and the OSGi platform is specified here with some objectives:

- This integration is as transparent as possible for the programmer. Components are loaded in bundles and registered as any other service in the OSGi platform.
- The use of real-time characteristics by the OSGi platform is also transparent. The OSGi source code is not modified to handle the real-time characteristics of components. A service tracker is used to invoke the framework tasks as loading and unloading.

## **5.1 Component implementation**

This section describes the specification of the framework in Java language. Components are implemented by Java classes. Classes in Java can provide several different interfaces at the same time, each interface for a different purpose by implementing a number of interfaces. This is the realization of multiple inheritance. Interfaces in Java are a set of operations (called *methods*).

Components provided in this framework are specifically service components. This is due to the use of Java as implementation language and that this language is based on the use of interfaces. Services provided by components are based, then, in Java classes interfaces.

In this specification two kinds of interfaces are considered according to their purpose:

*Composition* and *configuration* interfaces.

### *Composition interface*

An application is assumed to be designed by binding components together. Components offer interfaces to be binded to other components. This is the purpose of the *composition interface*. Java objects are able to provide several different interfaces, but for simplicity only one provided interface per component is considered in this framework. This also simplifies the composition of components. Similarly to *dynamic services* in OSGi only one service is provided by components although they may require several interfaces to work. Although this framework is OSGi-oriented, it makes no specific distinction between component provided interfaces. This allows to handle different methods from different interfaces in the same component as if they belonged to a single interface.

Composition of components is made through the handling of *dependencies*. A dependency is a binding from an active component to a passive component and the operation invoked in such passive component. The framework makes use of the `setDepend` method in the configuration interface to setup each dependency of the component. Components will make use of the interface provided by the received components. The component expects only one kind of component interface per dependence.

### *Configuration interface*

The configuration interface is provided by the component for the framework to manage the component life cycle. The framework makes use of it to deploy the component in the platform, transfer the state from one component to another, set up dependencies, and to start and stop it. This interface is common for all the components and the defined methods in this framework are in Figure 22.

---

```
public interface Component {
    public Object getState();
    public void setState(Object state);

    public void setDepend(Dependence dep, Component comp);

    public boolean start();
    public void stop();
}
```

---

**Figure 22:** Basic component configuration interface

---

The configuration interface provides methods for the following operations:

- Start and stop the component when it is instantiated or unloaded. The method `start`

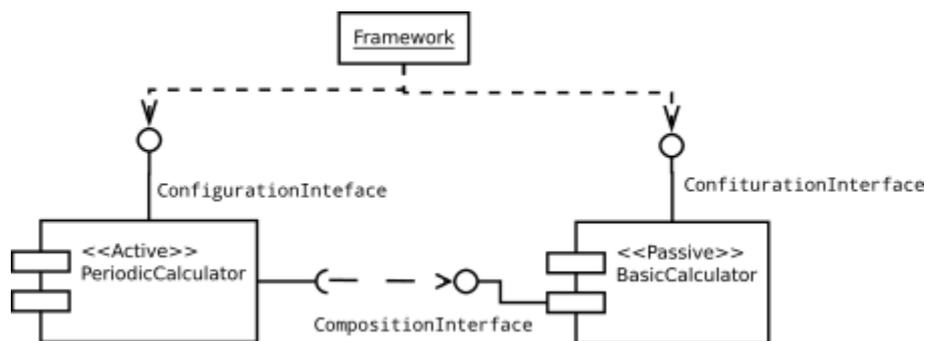
expects the component to initialize some configuration values if needed and the active component thread is started. The method `stop` makes the component finalize its thread execution and to free resources.

- To bind the component to already existing components in the platform., the `setDepend` method is used. It requires the information about the dependency and the component to be connected to.
- Copy the component state. This interface provides the methods `getState` and `setState`. They are used in the replacement process to be able to copy the internal working parameters of the component at the moment of the replacement.

The fastest way to transfer the state of the component is by translating the variable content of the component in a way that is easy to be handled. In this case, an unspecified Object. This gives freedom to the implementer to use a specific object according to the component's needs. Implementations of the `setState` and `getState` methods have to be provided by the programmer, saving the state of the object and restoring it.

The implementation of the `setDepend` method is facilitated by the use of reflection capabilities of Java, which allows providing a generic implementation of the method so that the component has no need to provide an implementation for this method. The object `Dependence` includes the name of the method to receive the component. A more detailed description of the `Dependence` object is given in following sections. This method has to be implemented by the programmer and the internal state or variables of the component set accordingly. If no method is found with the name described in `Dependence` then an exception is raised.

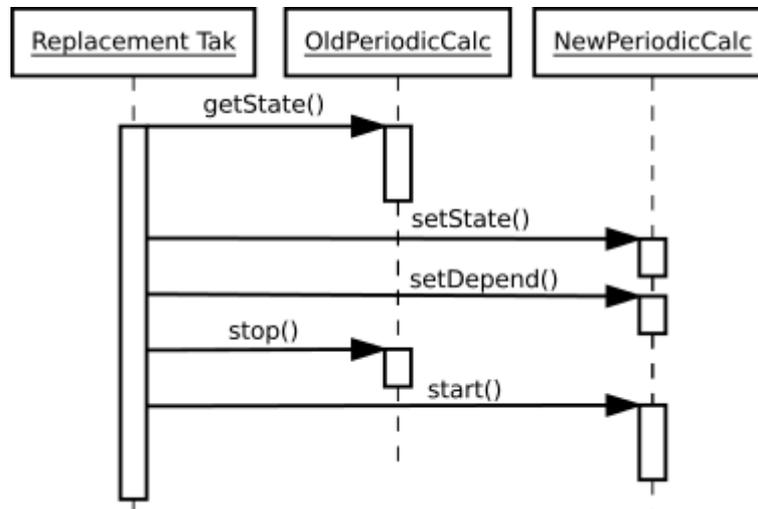
An example of composition and configuration interfaces is shown in Figure 23. Every component has a configuration interface that is manipulated by the framework. Composition interfaces are used to bind components and compose the application functionality.



**Figure 23:** Example of configuration and composition interfaces

Figure 24 shows an example of usage of the configuration interface by a framework task. In this case the replacement task uses the configuration interface of the old component to be replaced and the new component that will substitute it. The current state of the old component `OldPeriodicCalc` is obtained and then delivered to the new component, `NewPeriodicCalc`.

Then, the dependencies of the new instance are configured. In this case, the same reference to `BasicCalculator` that `OldPeriodicCalc` was using should be provided to `NewPeriodicCalc`. Finally the `OldPeriodicCalc` instance is stopped and the `NewPeriodicCalc` instance started.



**Figure 24:** Example of usage of the configuration interface by the replacement task

## 5.2 Framework

The framework description is split here in several parts:

- Component description
- Replacement task implementation
- Registry model

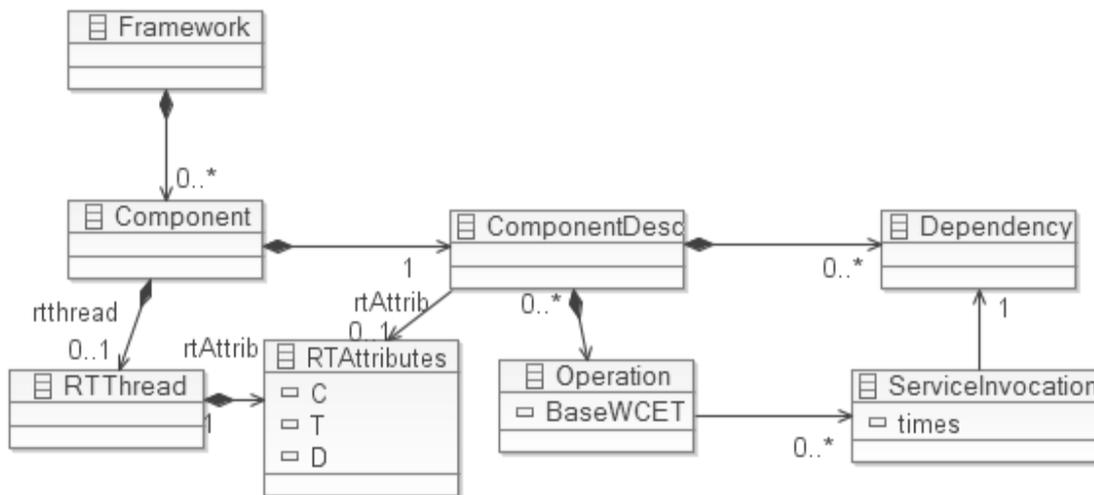
### 5.2.1 Component description model

Several classes are elaborated that contain the required information on temporal properties to apply the WCET calculation at run-time. These classes are represented in Figure 25. Active components contain an `RTThread` class with the code of the execution thread of the component. The temporal properties are in the `RTAttributes`, which contains the worst case execution time ( $C$ ), the period ( $T$ ) and the deadline ( $D$ ) of the thread. Additional parameters as thread priority may be used extending this class, depending on the specific scheduler used by the framework.

The `ComponentDesc` is used to describe the rest of characteristics of the component. These characteristics are the component *dependencies*, i.e., those components that interoperate with it, and the *operations* of the provided interface of passive components. The `Dependency` classes only contain the name of the components that the described component needs to work. The framework this class to bind component when there are loaded or replaced.

The `Operation` class contains the operation name and its specified WCET value, named `BaseWCET`, of the provided operation ( $C^{pass}$ ) as described in Table 4. This class is also used to

represent the temporal characteristics of active component threads. The framework supposes that the operation specifies the temporal characteristics of an active component thread if the operation name is “run”. This name is used for that purpose because it matches the `run` method of thread in Java and no other method should have the same name in the component. Then, the `BaseWCET` represents the  $C^{act}$  value of the active component. To describe the temporal information of this operation, as specified in Table 5, the `Operation` class makes reference to the operations it invokes with the `ServiceInvocation` class, which includes the number the number of times these operations are invoked ( $n_{i,j}$ ) and the dependency component to which these operations belong.



**Figure 25:** Component Description

Table 8 shows a representation of how Table 6 and Table 7 information is decomposed and distributed in the framework classes. Both `PeriodicCalculator` and `BasicCalculator` components are detailed. The active component `PeriodicCalculator` has an operation named `run`. This operation indicates that this component is an active component. The value indicated in the next column (5 ms) is the execution time of its execution thread ( $C^{act}_i$ ) without including operation invocations (`BaseWCET`). Following columns contain the information about the component dependencies and invoked operations. The `Dependency` class indicates that `PeriodicCalculator` depends on the `BasicCalculator` component. The framework will be able to create the corresponding bindings accordingly. The invoked methods (`Power` and `LimitedIntegration`) and the number of times they are invoked by the `run` operation are indicated in the `ServiceInvocation` class.

The `BasicCalculator` component contains in its description two `Operation` objects with their name ( $Op_{j,i}$ ) and their ( $C^{pass}_{j,i}$ ). These operations are `Power` and `LimitedIntegration`, and they do not make use of other component operations. So, this component has no dependencies.

<i>Component</i>	<i>Operation</i>		<i>ServiceInvocation</i>		<i>Dependency</i>
	<i>name/Op<sub>j,l</sub></i>	$\frac{BaseWCET/C^{ac}}{t_j/C^{pass}_{j,l}}$	<i>Op<sub>j,l</sub></i>	<i>n<sub>j,l</sub></i>	
Periodic Calculator	run ( <i>thread</i> )	5 ms	Power	4	BasicCalculator
			Limited Integration	1	
Basic Calculator	Power	2 ms			
	Limited Integration	3 ms			

**Table 8:** Example of component timing information in framework classes

To support the real-time execution of threads in active components the `RTThread` is implemented extending the `RealtimeThread` class provided by RTSJ. `RTAttributes` are, then, used by the methods provided by `RealtimeThread` and the corresponding scheduler.

Actually a `FRealtimeThread` has also been created to manage the schedulability of the thread and its relationship with the scheduler class implemented in this work (`Fscheduler`). `RTThread` inherits the characteristics of `RealtimeThread` through `FRealtimeThread`. These classes are described later in this chapter.

The framework provides a simple interface for a component to assign the code of threads that must be executed every time the thread is invoked. Precisely, code is included in the `RTThread` implementing the `execute` abstract method. This method will be invoked according to the real-time provided parameters. A `wrapup` method is also provided for the thread to be cleanly stopped and unscheduled (see Figure 26).

---

```

public abstract class RTThread extends FRealtimeThread {

    private boolean active = true;

    public RTThread(PeriodicParameters pp) {
        ...
    }

    public void run() {

        while (active == true) {

            execute();

            waitForNextPeriod();
        }
        wrapup();
    }

    protected abstract void wrapup();
    protected abstract void execute();

```

---

---

```

    public void setActive(boolean active) {
        this.active = active;
    }
}

```

---

**Figure 26:** RTThread class implementation

RTThread simplifies the creation of real-time threads. A mechanism is implemented to control the way to stop the thread. The thread is started through the invocation of the `start` method (inherited from `RealTimeThread`) and keeps running while the value of an internal attribute named `active` is `true`. The mechanism provided to stop the component is the use of the `setActive` method with a `false` value as argument. The internal `active` value will be updated with the received argument. The `setActive` method will be invoked by the `stop` method of the `Component` class to stop its execution.

The `PeriodicParameters` class belongs to RTSJ and it is generated making use of the attributes of the `RTAttributes` class. It describes the scheduling parameters for the scheduler to apply the schedulability analysis.

## 5.2.2 Replacement task

The framework also requires an infrastructure to apply component replacements. The core of this infrastructure is a real-time thread, named `ReplacementTask`, in charge of applying component replacements at specified time instants (e.g., when the component to be replaced is not being executed). This thread executes at predefined instants as it is included in the scheduler with timing parameters as described in Section 4.3, according to the selective component replacement model.

A queue is implemented to store replacement requests until they can be attended by the replacement task (see method in Figure 27). Code to add requests to the queue is enclosed in a synchronized block to avoid race conditions.

---

```

Vector<Request> requests_queue = new Vector<Request>();

public void putRequest(Request uo) {
    synchronized (requests_queue) {
        requests_queue.add(uo);
    }
}

```

---

**Figure 27:** Method to enqueue replacement requests

The replacement task is executed according to its timing configuration and the `run` method of the thread executed (see Figure 28). The first request in the queue is extracted to be performed every time the task is executed. If the queue is empty then no action is performed in that execution. Code is enclosed in synchronized blocks to avoid race condition and the concurrent execution of other operations on the same queue elements.

A boolean variable named `replacementsActive` is used to indicate if the replacement task is active. The replacement task can be stopped using a method named `stopReplacements()` that sets the `replacementsActive` value to `false` and makes this thread to stop its execution.

---

```
public void run() {  
  
    while ((replacementsActive == true) || (requests_queue.size() != 0)) {  
        Request order = null;  
        synchronized (requests_queue) {  
            if (requests_queue.size() != 0) {  
                request = requests_queue.firstElement();  
                requests_queue.removeElementAt(0);  
            }  
        }  
        if (request != null) {  
            synchronized (request) {  
                request.apply();  
            }  
        }  
        waitForNextPeriod();  
    }  
}
```

---

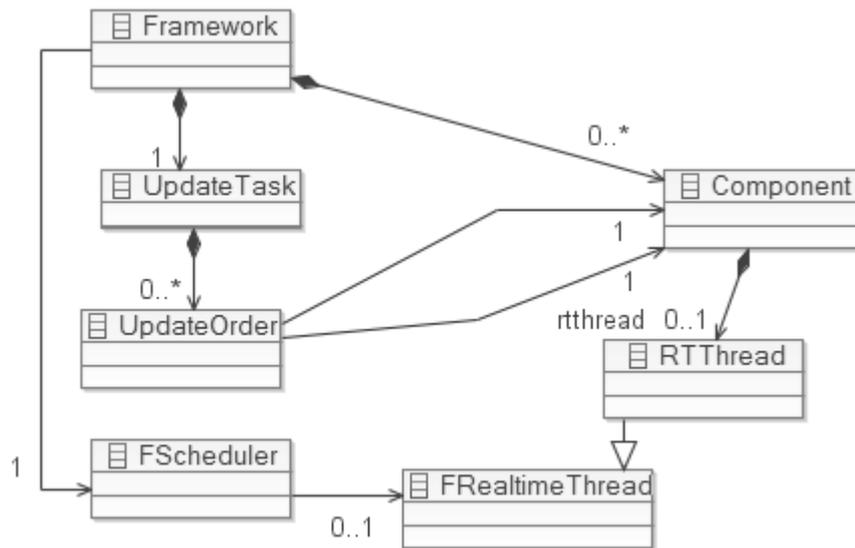
**Figure 28:** Replacement thread execution

---

The code to apply the replacement is in the specific `Request` class implementation. As can be noted, this code is implemented so that different requests could be performed by the `ReplacementTask`. Different requests can be implemented extending the `Request` class and implementing different `apply` methods. The code corresponding to the component replacement request is detailed later in this chapter.

The scheduling of the replacements task is fully controlled by the framework. RTSJ classes have been extended (see Figure 29) to have control over the execution of every real-time thread in the system, included the replacement task. The following classes have been implemented for that purpose:

- `FScheduler` is a scheduler based on the `PriorityScheduler` provided by the RTSJ specification due to the use of a priority-based scheduling based on RMS technique. It provides the task set management and an acceptance test implementation. Any test could be applied.
- `FRealtimeThread` extends `RealtimeThread` class from the RTSJ specification, providing some characteristics like adding and removing the thread from the scheduling task set of the scheduler.

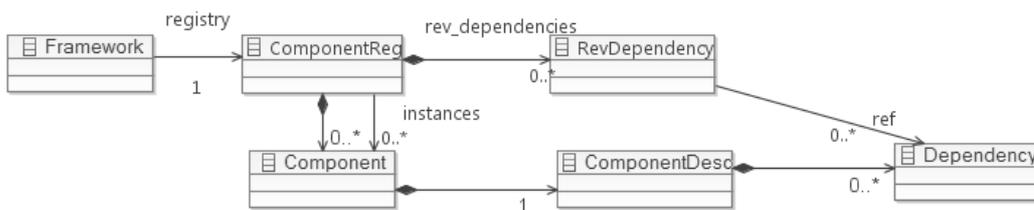


**Figure 29:** Framework architecture for the replacement task

Later it will be described how this replacement task is used, with a detailed description of the operations it performs.

### 5.2.3 Component registry

One of the main elements of the framework is a *component registry*. It is designed to hold the information about every component instance running in the system as well as its dependencies. It also contains information about loaded and/or running components. Actually, the registry is in charge of storing the information about running component instances and not the `Framework` class (see Figure 30). The information held by the registry is used by the framework task as component loading, unloading and replacement.

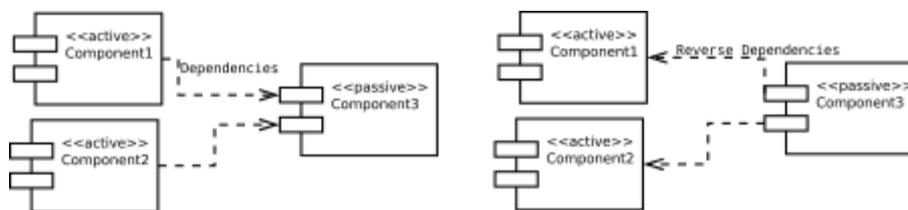


**Figure 30:** Component Registry Model

The `ComponentReg` is the class implementing the component registry. It contains the following information:

- A list of *registered components* in the system. A component can be loaded and registered without the need to be started. The registry has all of its information although it is not running.

- At any time instant it also contains the list of running components. That is a list of running *instances* of already *registered components*. In the Java language classes have to be instantiated to be executed. Components are configured when instantiated initializing their state.
- Besides the information about component dependencies included in the installed components list, a list of *reverse dependencies* (*rev\_dependencies*) is stored in the registry (see Figure 30 and Figure 31). This list is useful in the WCET calculation algorithm described later to calculate at run-time the WCET of every component service. Reverse dependencies are used to find dependencies in reverse order, for instance, from the replaced component to those that make use of it. Reverse dependencies list is updated every time a new component is added to the system or when a component is replaced. It reduces the amount of time to find the components that depend on a specified one.



**Figure 31:** Dependencies in reverse order

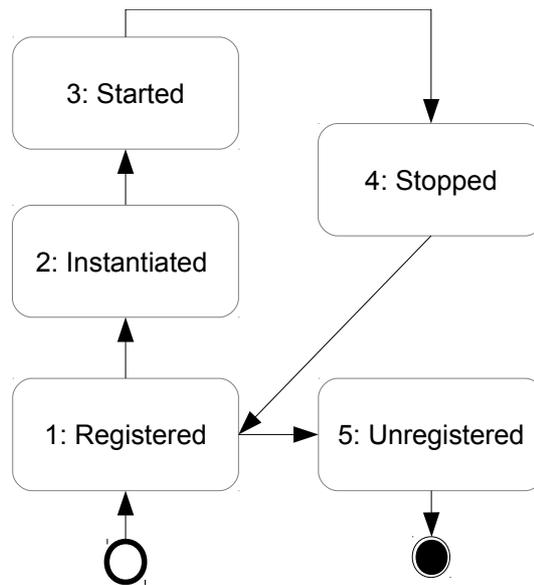
Figure 31 shows an example of dependency of components. If `Component3` is replaced, then the WCET of `Component1` and `Component2` must be recalculated. The reverse dependencies list is used to find components that depend on the replaced one. Reverse dependencies can only be added to the list when components new components are instantiated and their dependencies configured. For instance, when the `Component2` is configured and its dependency on `Component3` is set then the reverse dependency of `Component3` on `Component2` is added to the reverse dependencies list.

## 5.2.4 Framework tasks

The tasks of the framework are loading, unloading and replacing of components at run-time.

The main complexity when loading and unloading components in a Java environment is the use of *classloaders*. There are existing tools able to manage the load and unload of code into and out of memory using classloaders. So, this task is reserved for a different framework. The OSGi bundles system is an example of framework able to use Java classloaders to load and unload Java classes from files, and to correctly manage Java packages imported and exported in every bundle.

In this framework the loading and unloading of a component is limited to the registry and unregistry of a component class and its information into the framework. Classes are already loaded into memory using a different framework able to deal with classloaders as the OSGi one.



**Figure 32:** Component life cycle in the Java specification

The life cycle that the component follows in the Java specification of the framework is shown in Figure 32:

1. The component class is *registered* in the framework. All the information for the calculation of the WCET and its schedulability is provided.
2. The component is *instantiated* and its dependencies set, bonded to other components, but not yet started.
3. The component instance is *started*, starting its real-time thread.
4. The component instance is *stopped*. The real-time thread is stopped and its resources freed. The component instance itself stops being used. The component class is still registered and a new instance could be created.
5. The component class can be *unregistered* from the framework. This means that the framework will force the running instance of the unregistered component to stop if it is still running.

### **Loading**

As stated before, the action of loading and unloading the code on memory is delegated to existing frameworks able to cover those needs. After the load of the component classes, the component can be added to the framework registering it and its description. Once the component has been registered, it can be instantiated. Components can be instantiated recursively according to their dependencies. Components on which this component depends are instantiated if they have been previously registered but not yet instantiated.

---

```

Instantiate (Component):
    if (Component in registry.getInstances())
  
```

---

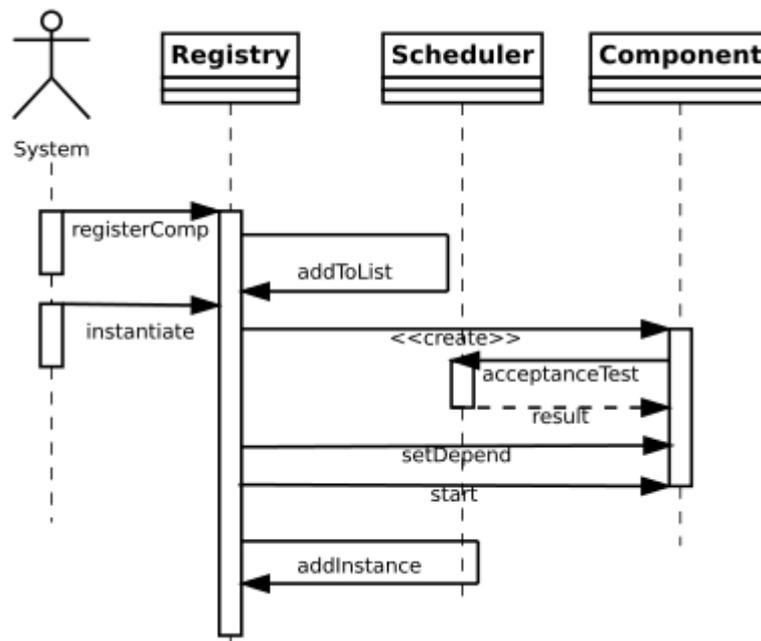
```
        return registry.getInstance(Component)
    end if
    if (Acceptance_test(Component) == false)
        return null;
    end if
    comp = new Component
    dependencies = registry.getDependencies(Component)
    for all D in dependencies do
        inst = Instantiate(D)
        if (inst == null)
            return null;
        else
            comp.setDepend(D, inst)
        end for
    if (comp.start() == true)
        registry.addInstance(comp)
        return comp
    else
        return null
    end if
```

---

**Figure 33:** Component instantiation process

Algorithm in Figure 33 shows how component dependencies are instantiated if needed. Only if all dependencies match, the component execution is started and returned.

A sequence diagram representing the instantiation process is shown in Figure 34. When a component is registered, then the information about the component is stored in the registry. Adding a component to the registry consists of adding its information to the components list. Nothing more is done until the component is instantiated. When the registry is requested to instantiate the component, an object is instantiated from the component class, its dependences are set and the instance started.



**Figure 34:** Sequence diagram of component registering and instantiating

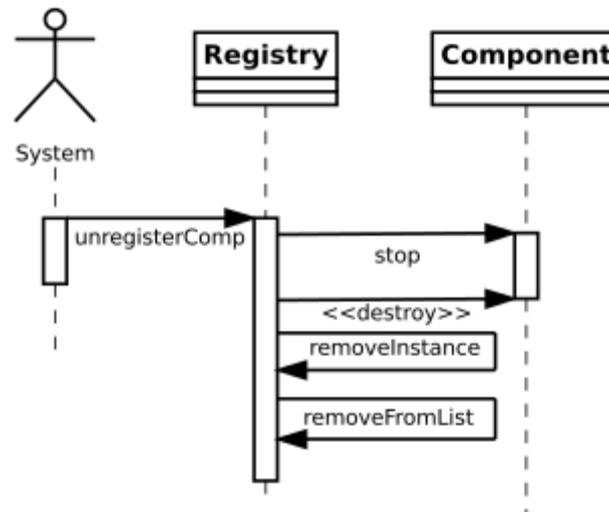
Another key element in the component instantiation is the acceptance test. The system must fulfill the following conditions described in the acceptance test (Section 4.5):

- The component has to be schedulable in the actual state of the system. It can not make other running component to miss their deadlines.
- The component has to be replaceable. To be able to replace this component, the replacement task has to keep being schedulable after updating its scheduling parameters.

The acceptance test basically consists of updating the WCET of affected components by the loaded or replaced component and modifying the updating task characteristics accordingly. Then a schedulability test is applied by the scheduler. If the modifications are schedulable then the component instantiation or replacement is accepted.

### ***Unloading***

Component unloading consists of stopping the component's execution and removing its thread from the scheduler. Then dependencies and reverse dependencies are removed as well as the component instance from the instances list of the registry. Figure 35 shows the sequence diagram corresponding to the component unloading process.



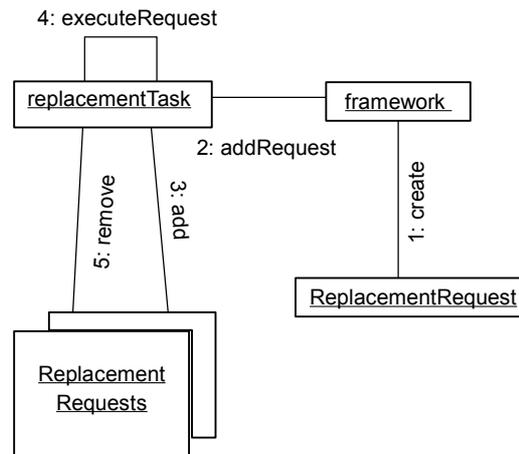
**Figure 35:** Sequence diagram of component unregistering

The current approach does not unregister the components on which this component depends. Each registered component has to be registered specifically. To be unregistered, it should be checked if there are running component instances that also depend on every dependency of this unregistered component.

### ***Replacing***

As described in Section 4.2, the component replacement task has special timing requirements. The real-time scheduling characteristics of this task are modified every time a new component is accepted in the framework.

Unlike registering and unregistering, the replacement of a component is performed only at safe instants (see Section 3.2), with the parameters already described above. Figure 36 represents the complete process. Once the component replacement is requested to the framework it creates an update order and passes it to the update task. The update task adds it to the pending update tasks queue until its next scheduled execution.



**Figure 36:** Replacement task collaborative diagram

All the steps represented in Figure 36, except the replacement execution represented by step 4, are not performed in a real-time thread. Free CPU time is used for that purpose. This is possible because this part of the process has no timed restrictions. The replacement process itself is performed at safe instants: when the replacement task is scheduled for execution and the component to be replaced is not executing. Given that the execution of the task is correctly scheduled (CPU time is reserved), and it is done with highest priority then neither this thread is disturbed by any other component nor other component execution is disturbed by this thread execution.

---

```

Replace (OldComponent, NewComponent) :
    oldInstance = registry.getInstance (OldComponent)
    newInstance = new NewComponent
    newInstance.setState (oldInstance.getState ())
    dependencies = registry.getDependencies (NewComponent)
    for all D in dependencies do
        inst = registry.getInstance (D.getComponent ())
        if (inst == null)
            return null;
        else
            newInstance.setDepend (D, inst)
        end if
    end for
    rev_dependencies = registry.getRevDependencies (NewComponent)
    for all R in rev_dependencies do
        dep_component = registry.getInstance (R.getComponent ())
        dep_component.setDepend (R.getDepend (), newInstance)
        newInstance.setState (oldInstance.getState ())
    end for

```

---

**Figure 37:** Replacement process

Figure 37 describes the component replacement process. The new component is instantiated with the information stored in the registry. Then, the state is copied from the old instance to the

new one. Then, the bindings are set. First, the components on which this component depends are set. For simplicity, dependencies are supposed to be already instantiated. In other case this would require to reconfigure all of them. Only the essential steps of instantiating and configuring one component are represented here. Reconfiguring components depending on this one represents redundant code in the figure.

After the dependencies are set, then, the reverse dependencies list in the registry is used to find the components that make use of this component and their dependencies are updated. At this moment the new component has replaced the old one.

### 5.2.5 Calculation of WCET of component operations

Previously to any component instantiation in the system, the concrete worst case execution time of the new component operations has to be calculated. This WCET is unknown until run-time because the concrete implementations of the service components used in the system are only known at run-time. Similarly, when a component is replaced, the budget time required by the components that depend on the replaced component needs to be recalculated.

The required information for such calculation is described in Section 4.4 and modeled for the framework in Section 5.2.1. This information is used before every component is added to the system to calculate the WCET of the affected components. As described in that section, the timing information for the active component thread is provided as an operation of the component named “run”. The WCET of all the component operations, be active or passive component, is calculated. The WCET of the active component thread is differentiated by its name.

---

```
CalculateWCET(Component) :
  operations = registry.getComponent(Component).getDescription().getOperations()
  for all O in operation do
    invocations = O.getInvocations()
    WCET = O.getBaseWCET()
    for all S in invocations do
      WCET = WCET + S.getTimes() * S.getServiceOperation().getWCET()
    end for
    O.setWCET(WCET)
  end for
  rev_dependencies = registry.getRev_Dependencies(Component)
  for all R in rev_dependencies do
    calculateWCET(R.getComponent)
  end for
```

---

**Figure 38:** Calculation of WCET of component operations

Some elements of the algorithm in Figure 38 have been simplified for comprehensibility reasons. In the case of reverse dependencies, there is no need to recalculate the WCET of all the operations in the component but only the affected by the modified binding. Similarly the information contained in the `Dependency` class and its management is simplified for presentation reasons.

### 5.3 OSGi integration

This section shows the capability to provide real-time capabilities to a dynamic service platform. Real-time component's replacement is provided to the dynamic platform without minimal or no additions to the service platform.

Although the framework specification is OSGi independent and can be used independently from it, some details have been restricted in the framework for simplification to easy integration.

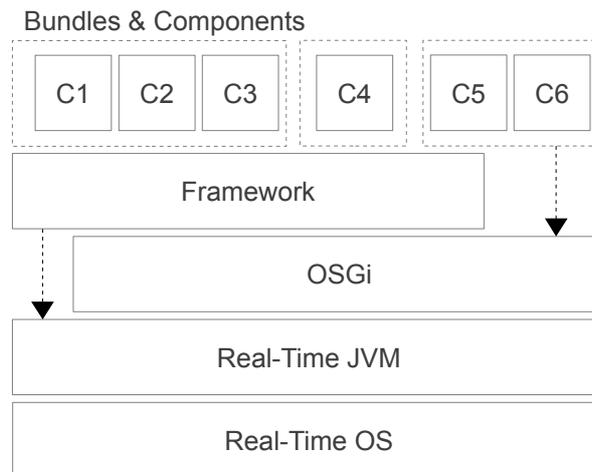
- Interfaces provided by components are named *services* to assimilate the nomenclature used in the OSGi framework.
- The number of *services* provided by every component is limited to one. The number of services required by the component are not limited.

As described in Section 2.5, OSGi has dynamic capabilities from which the framework specification gets benefited:

- Dynamic load of Java classes at run-time: Classes are packaged in a file (*jar* file) named *bundle* which is loaded at run-time.
- Dynamic unload of Java classes: *bundles* are also unloaded at run-time from memory.
- Management of components: OSGi bundles can contain an Activator class able to manage components provided by any bundle, registering or unregistering them from the OSGi register.

The specified framework allows adding real-time characteristics to the component in the system:

- Transparently adding support for real-time components: The use of a *Service tracker* allows intercepting real-time component registrations to also register them in our real-time framework.
- Adding real-time component replacement without altering their execution, which OSGi is not able to do without stopping and restarting components.



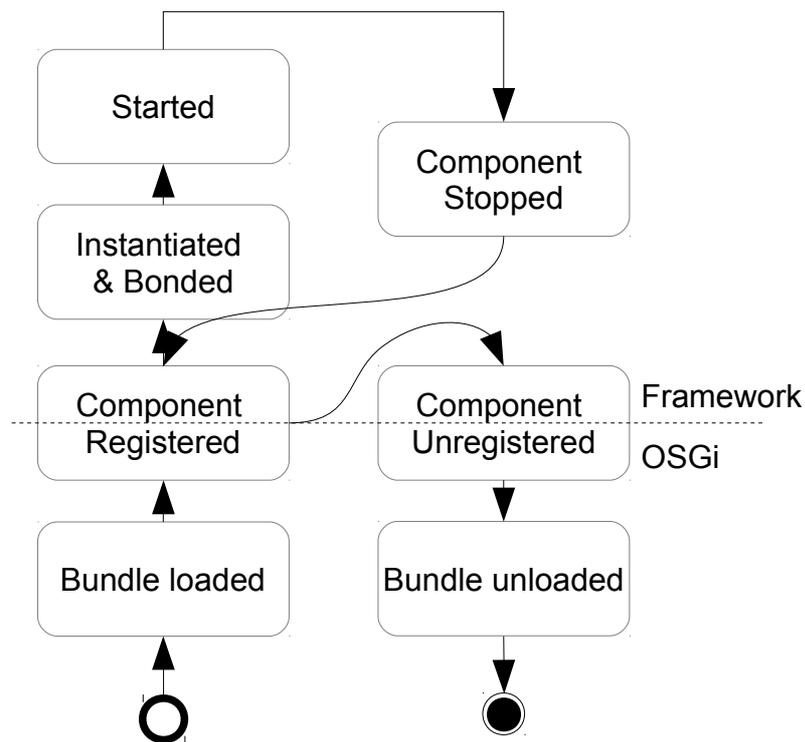
**Figure 39:** System architecture after the framework and OSGi integration

After the integration of the framework and the OSGi platform, the system architecture is as represented in Figure 39. It contains the following elements:

- A Real-time Operating System. Different Java Virtual Machines may require to work on an OS with real-time capabilities or not, but real-time support is required for the system to have a scheduler with low latency precision.
- A Real-Time Java VM. This means the virtual machine must provide access to the RTSJ for the framework to make use of it. This VM must also support the execution of the OSGi platform. There are some incompatibilities found, as the use of the RTSJ reference implementation<sup>20</sup> with any OSGi implementation. The RTSJ reference implementation only supports JavaSE version 1.3 while any OSGi implementation requires at least JavaSE version 1.5.
- An OSGi implementation. Only a small modification in one of the configuration files is required as described later in this chapter for the bundles to have access to the RTSJ API provided by the Java VM.
- The Framework. The framework itself is set up as an OSGi bundle that can be installed and uninstalled in the platform, providing all of its implemented characteristics. Beside being a bundle that makes use of the OSGi platform support its functionality is based in the RTSJ API provided by the Java VM.
- Finally bundles containing components are installed in the platform. They make use of the services provided by the OSGi platform as well as the real-time characteristics provided by the framework.

---

<sup>20</sup> <http://www.timesys.com/java/>



**Figure 40:** Component life cycle within OSGi platform

Figure 40 shows the life cycle of a component when the framework is integrated with the OSGi platform. The process slightly changes although the internal workings of the framework are still the same. The main characteristics are:

- Now OSGi allows loading the component code in memory and unloading it. Components are packaged into bundles, which are managed by OSGi.
- The component registration in the framework is joined to the OSGi component registration. The mechanism is described later. A component is registered in the OSGi platform and transparently also in the framework. The component is also unregistered in both places before the code is unloaded.
- The rest of the component life cycle keeps unchanged inside the framework once a component is registered.

### ***The framework as an OSGi bundle***

The first step to integrate the previously implemented framework into the OSGi platform is to generate a bundle suitable for installation in the platform. The other alternative is to generate a modified implementation of the OSGi platform which already includes the real-time framework. This alternative represents generating a new implementation of every OSGi existing implementation. In Section 2.5 are enumerated some of the existing OSGi implementations. This last alternative would represent generating a new modified implementation for every original OSGi implementation.

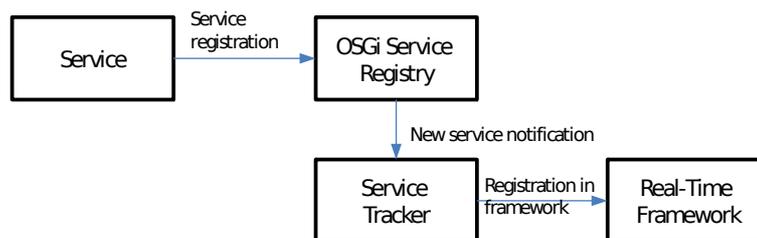
Generating an installable bundle results in a more generic solution that can be installed on any of the described OSGi implementations. The main requirement is still the use of a Java Virtual Machine with real-time support (see Section 2.5.1).

For any other bundle to use the framework the packages implemented by the bundle have to be exported to the rest of the system. Similarly, the real-time packages with the RTSJ implementation have to be imported.

<b>Export-Package:</b> framework, framework.component, framework.register, framework.tests	<b>Import-Package:</b> javax.realtime; <i>resolution:=optional</i> , org.osgi.framework; <i>version="1.3.0"</i>
<b>Figure 41:</b> Imported and exported packages in bundle MANIFEST file	

### OSGi Service trackers

The OSGi platform has some mechanisms that allow the programmer to track the services that are added, modified or removed from the platform. Service trackers also offer a filter so that the service tracker is only notified when a specific type of service is added or removed.



**Figure 42:** Service registration notification with Service Tracker

Service tracker offers the following interface as shown in **Figure 43**. Every method is invoked when the corresponding action occurs in the system:

- Addition of a service: a new service is registered in the OSGi platform with a set of properties. The properties list is provided as a Dictionary of elements in the form of key and value. A service reference is received in that case.
- Removal of a service: a service is unregistered from the platform.
- Modification of a service: the tracker is notified if the set of properties is modified.

```

public class Tracker extends ServiceTracker {

    public Tracker(BundleContext context, Filter filter,
                  ServiceTrackerCustomizer customizer) {
    }
  
```

---

```
@Override
public Object addingService(ServiceReference reference) {
}

@Override
public void modifiedService(ServiceReference reference, Object service) {
}

@Override
public void remove(ServiceReference reference) {
}
}
```

---

**Figure 43:** Service Tracker interface

Real-time components can be identified when added to the system using the service tracker and the corresponding *filter*. The filter is an object facilitated to the service tracker. The parameters of the filter will identify the real-time components registered in the platform. There are two possibilities to identify a real-time component that belongs to the framework:

- The first possibility is to identify the `Component` class belonging to the framework. The detection of the `Component` class is safer than the other option because it assures that the component provides the required configuration interface for the framework
- The other possibility is to filter using the properties provided with the component when registered. This possibility is more flexible because it allows to use the properties before the component is registered in the framework, and it also allows handling components implemented using classes different to the `Component` class if the framework is extended.

The filter based on properties is finally used here, because of its flexibility. For a new real-time component addition to be intercepted it is required to include a property named `RTDESC`. As a value for this attribute a `ComponentDesc` class instance is expected.

Components can be registered in the platform as normal OSGi services and transparently managed by the real-time framework. An example of new component registration in the system follows in Figure 44. Once the addition of a new component is detected then, the corresponding actions can be applied, as registering the component in the real-time framework.

```
FibServ fib = new FibCompImp();
ComponentDesc desc = new ComponentDesc( ... );

Hashtable<String, Object> prop = new Hashtable<String, Object>();
prop.put("RTDESC", desc);
context.registerService(FibServ.class.getName(), fib, prop);
```

---

**Figure 44:** Real-time component registration in framework

---

### *Tracker implementation*

To provide a transparent integration of the real-time framework with the OSGi platform there are two alternatives:

- The first alternative is to modify the source code of OSGi. Modifying the source code, the framework registry can be integrated into the OSGi registry. Components and its dependences could be managed inside OSGi. It may require a considerable amount of work to perform such modifications in the OSGi source code, and these modifications would only work in an specific implementation of OSGi.
- The other alternative is to use a *Service Tracker*. This is a class described in the OSGi specification[103] that allows being notified for every change in the OSGi registry. The use of this tracker allows managing the framework real-time components without having to access the OSGi source code and avoiding the need to use a specific OSGi implementation with included real-time characteristics.

In the following Figure 45 the code to intercept the registration of a new component is included. The component and its real-time description are obtained from the service reference provided by OSGi. Then, this information is used to register the component into our framework.

```
public Object addingService(ServiceReference reference) {
    ComponentDesc rtdesc = (ComponentDesc) reference.getProperty("RTDESC");
    Component comp = (Component) context.getService(reference);

    boolean rc = Framework.registerComp(comp, rtdesc);
    if (rc == false)
        ...
    return comp;
}
```

---

**Figure 45:** Code to intercept a new component addition

---

The equivalent code to remove a component from the framework is in Figure 46. In both cases the code makes use of the `Framework` class to register and unload the corresponding component. When the component is unloaded it has to be registered again in the framework before using it again. It is implemented in this way because the component is also being unregistered from the OSGi platform, probably from a bundle activator that is being unloaded from the platform.

---

```

public void remove(ServiceReference reference) {
    Framework.unregisterComp(((Component) context.getService(reference))
        .getName());
}

```

---

**Figure 46:** Code to intercept a new component removal

A component replacement cannot be easily implemented without introducing contradictions in the OSGi semantics of components registration. The service registry in OSGi allows having multiple active implementations of the same service at the same time. A registration of a new implementation does not represent to substitute a previous existing implementation. The best practices in OSGi suggest using multiple bundles to provide service implementations. One bundle provides the service interface while different bundles provide different implementations. For instance one bundle provides the specification interface of the service (i.e., a fibonacci calculation service), while different bundles provide different implementation classes of such service (i.e., an implementation based on recursion and another one based on the use of vectors with precalculated values). This allows multiple implementations registered in the platform at the same time. They are differentiated by the provided parameters when registered.

Changing the implementation of a service usually consists of unloading the bundle corresponding to the active component and loading or activating the new implementation. This approach is not acceptable in the scope of this work where the new implementation has to be ready before stopping the old implementation.

The only acceptable approach in this case is the direct use of the framework to replace an existing component. The name of the old implementation is provided to the registry with the new implementation. The framework is in charge of replacing the existing implementation of the component as described in the Section . The following code in Figure 47 represent an example of component replacement.

---

```

prop = new Hashtable<String, Object>();
prop.put("RTDESC", framework.tests.streaming.Render2.desc);
Render2 render2 = new Render2();
ComponentReg.getInstance().replace(Render.class.getName(), render2);

```

---

**Figure 47:** Code to replace a component

The third method provided by the service tracker, that is `modifiedService`, invoked when the properties of the service are modified. This method is not used in the actual implementation. A change in the properties of the component may affect to basic aspects. One of them is the calculation of the WCET modifying the services invocations. This change is not acceptable without changing the component's implementation. That would represent a component's replacement, which should be made as indicated above. The other aspect is a change in the schedulability parameters as the periodicity or deadline. This would represent a simplified component replacement where the component is not actually replaced, but new schedulability tests are applied. This case is not implemented because the main target in this work are component replacements.

### ***Framework component bindings management***

Although OSGi provides its own component model as *Declarative Services* the integration of the framework with that component model is not easy. One of the issues is the already described management problem of the component replacement.

The other problem is the management of components bindings. The Declarative Services (DS) subsystem in OSGi is able to manage service bindings by looking for required services of a component according to declared properties and to bind them. This binding system should be revamped if was to be used by the real-time framework. Mayor changes would have to be applied to the source code of the OSGi implementation to intercept DS bindings to handle references in the framework to recalculate WCET of every services invocation and schedulability analysis. By now this task is applied by the implemented real-time framework and only real-time components are handled by it.

### ***OSGi classpath from VM provided to bundles***

Other of the issues in running OSGi on a real-time Java Virtual Machine with RTSJ support is to let bundles to access RTSJ classes. By default OSGi implementations detect the JVM version they are running on and configure their classloaders to be able to load the available classes in every Java profile. To do that a text file is included in the implementation release with all the available java packages per profile.

Java VMs with RTSJ support include a `javax.realtime` package that can be imported by Java applications running on those VMs. The name of this package has to be included in the java profiles of the OSGi implementation. This will make OSGi classloaders to make this package available to installed bundles. Bundles that require the use of the RTSJ characteristics have to import this package in the corresponding classes and in the `MANIFEST.MF` file.

The framework is implemented in a way that most of the bundles containing a real-time component do not need to import the packages corresponding to RTSJ. The code in the framework encapsulates by default the use of `waitForNextPeriod` method. Avoiding the knowledge or use of RTSJ implementation details as much as possible.

## ***5.4 Predictability consideration on Java language***

One of the main issues regarding real-time implementations in the Java Virtual Machine is the management of the memory and the garbage collector[32]. The unexpected execution of the garbage collector breaks the scheduled execution of real-time threads. Many solutions are provided to minimize the effect of the execution of the garbage collector in real-time systems [33–35].

RTSJ provides some alternatives to avoid the use of the garbage collector, like de use of no heap threads in which no object can be allocated in memory during the execution of the thread. This way the garbage collector can be preempted by this thread assuring that no memory inconsistencies are produced. The other alternative provided by RTSJ is the use of scoped memory, where the thread data is allocated. This memory scope is limited and freed when the execution of the thread ends, but its management is usually extremely complicated. An *immortal*

*memory* is also provided, where threads manually manage allocation and reuse the allocated data. This memory is never freed, so the execution of the garbage collector is not needed to recycle this memory area.

Given that the use of the garbage collector is difficult to be avoided, especially when previous existing Java code is to be reused, other alternatives have to be applied to deal with the garbage collector. The alternative applied in this framework is based on [36].

A new task is incorporated in the framework designed to handle the execution of the garbage collector. The objective of this thread is to reserve CPU time in the scheduler for the garbage collector to completely recycle all the needed memory. This task can be tuned with the values obtained according to [36]. The implementation of this task is specific to the Java virtual machine on which this framework is executed, so it is considered out of the scope of this work.

The specialization of the Component class to make use of `NoHeapRealtimeThread`, `ScopedMemory` and `ImmortalMemory` is still possible but not yet implemented.

## 5.5 Summary

Several objectives in this chapter have been achieved:

- In this chapter, a specification of the framework described in the previous chapter is provided. This specification is based on the the Java language.
- Real-time characteristics of the framework are addressed making use of RTSJ. Extensions are made to classes provided by RTSJ.
- The required extensions to RTSJ are identified to include the extra functionality of a component replacement framework that are not contemplated by RTSJ.
- The WCET of components is calculated at run-time before they are accepted and scheduled for their execution.
- A *replacement task* is included, able to replace components in a bounded time, at run-time without interfering the execution of any running component.
- Some aspects of Java that determine unpredictability problems are evaluated. The execution of the garbage collector and its unpredictable generated interference in schedulability is one of the problems. The other is the use of a virtual machine that adds more unpredictable execution times.
- This specification is made to be used in generic environments with reduced number of modification of previous existing code. Specialized classes provided by RTSJ can be used to reduce the predictability problems of the Java language, but it would increase the incompatibility with already existing code.
- The complete framework has been integrated in the OSGi platform. The framework is packaged as a bundle and a service tracker is used to transparently connect both platforms. The capability of loading and unloading of code is provided by OSGi, the real-

time characterization of services and the capability to safely replace components is provided by the framework.



# Chapter 6

## Validation

---

This chapter includes the results of various experiments in which the developed component replacement models are applied to prove the validity of the proposed ideas and concepts. Generic tests are applied to evaluate the results of applying the proposed component replacement models on the schedulability of task sets. The performed tests are:

- An analysis of the results of adding the component replacement capability to a previously existing real-time system. This affects the number of threads in the system and their schedulability.
- A comparison of the difference in schedulability of a system when the pessimistic or the selective replacement models are applied.
- A specific multimedia scenario is set up that simulates a real case, to test how this proposal can be applied to automate component replacements and to achieve specific replacement times on soft real-time systems.

The proposed algorithms have been implemented in Java. Schedulability tests do not make use of the RTSJ specification. Schedulability tests are not affected by the use of Java as results only depend on the parameters of tasks and applied algorithm for the schedulability analysis.

RTSJ is applied in the multimedia scenario, where the complete framework implementation is used. The execution of the replacement task and the multimedia tasks are directly affected by the use of Java, the low performance of the virtual machine and the garbage collector. This low performance increases the execution times of tasks compared to their implementations in languages that make no use of virtual machine as C or Ada. This does not invalidate the obtained results as the execution time of the proposed tasks that, although the variability generated by the virtual machine, are still bounded in time, providing certainty in the schedulability of the system.

## 6.1 Schedulability tests

One of the main questions of a real-time systems designer is if this proposal can be applied to their system or any already deployed system. This proposal affects the *schedulability* of a real-time system as it reserves processor time for new tasks. It is important to know if the system can hold the same amount of components or tasks. It is also important to know the *overhead* generated by the replacement task and the acceptance tests in the running system to know the available processor time when this proposal is implemented. The last important value to take into account is the usability of this proposal in terms of number of component *replacements* that actually will be accepted at run-time. The effect of adding the presented component replacement models to a dynamic real-time system is evaluated in this section in terms of:

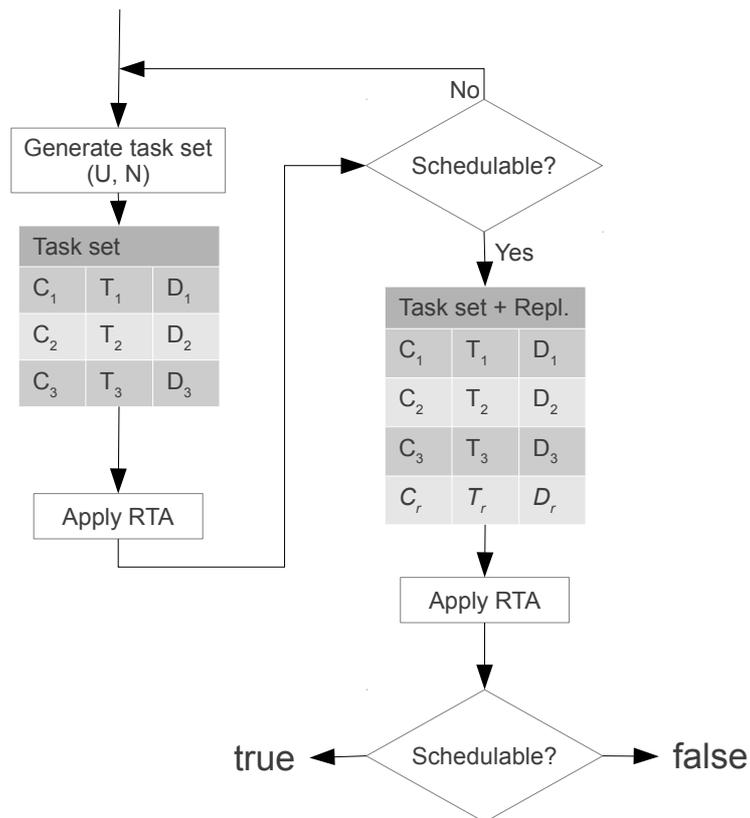
- *Schedulability*. This is indicated by the number of tasks that are still schedulable after the addition of the replacement task to an existing system.
- *Overhead*. This is indicated by the CPU utilization by the tasks set when a replacement task is incorporated in the system.
- The number of accepted or rejected *replacements*. This is the percentage of replacements accepted or rejected after they are requested.

### *Elaboration of schedulability tests*

Tests are performed generating a schedulable task set using the *UUniFast* algorithm [145] with several different processor load values. The *UUniFast* algorithm generates a task set with uniformly distributed processor loads to simulate a generic scenario with different kinds of tasks and different characteristics, as execution time ( $C$ ) and period ( $T$ ). The deadline assigned to the tasks is the period ( $D = T$ ). This algorithm is used in other works as a [146] to evaluate acceptance tests and the schedulability of the generated task sets. The exact *response time analysis* [147] is applied for the schedulability tests (see Equation (7)). For simplification purposes, it is assumed that there are no shared passive components that may generate execution delays.

Task sets are generated for various ranges of processor loads and different number of tasks. It is assumed that every task corresponds to an active component with their WCET already calculated. It is assumed that the execution time corresponding to the operations of passive component is already included in such WCET.

After finding an schedulable task set with the selected parameters (processor load and number of tasks), then the replacement task is added to the system. The schedulability of the system is tested. The number of task sets that stop being schedulable is considered a measure of the interference generated by the addition of the replacement task to a real-time system. The test is applied to 100 schedulable task sets with the same processor load and number of components. The number of failing task sets is obtained.



**Figure 48:** Schedulability tests

Figure 48 shows how schedulability tests are performed. Task sets are generated according to the received parameters, that is, the processor utilization ( $U$ ) and the number of tasks in the system ( $N$ ). Task sets are randomly generated according to the *UUniFast* algorithm until a schedulable task set is found. The schedulability of the task set is determined applying the Response Time Analysis (RTA). Then the corresponding replacement task is added to the task set and the schedulability test performed again. If the system is schedulable then it returns **true**, else returns **false**.

---

```

for U in 1 to 100
  for N in 1 to 100
    tests[U,N] = 0;
    for t in 1 to 100
      if (schedulability_test(U,N) == false)
        tests[U,N] = tests[U,N] + 1;
      end if;
    end for;
  end for;
end for;

```

---

**Figure 49:** Example of schedulability tests application

An example of how complete schedulability tests are performed is in Figure 49. For a given range of processor utilization ( $U$ ) and a number of components ( $N$ ), a fixed number of 100

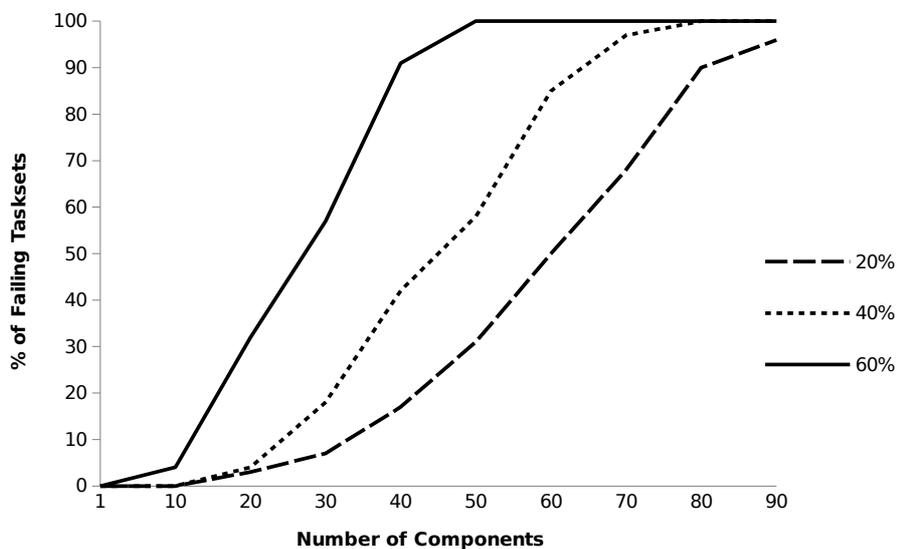
schedulability tests are performed to obtain representative results. Every test returning false is a miss in the application of the replacement task. These misses are accounted and represented in next subsections.

### *Pessimistic replacement model*

The first replacement model to be tested is the pessimistic one. To perform this test the addition of the replacement task is simulated increasing the execution time ( $C$ ) of every task applying the Equation (17) with an average value of 4 bindings by component. Then, the schedulability tests are performed with every generated task set. Equation (20) is applied here to perform such schedulability tests.

Task sets are generated with different values of processor utilization and number of components. The occupancy levels vary from 10% of CPU usage to 100%, and the number of components (represented by tasks) varies from 1 to 100. Different CPU loads are represented in Figure 50.

Figure 50 shows the average of failing task sets for 20%, 40% and 60% processor occupancy. With 50 components, the maximum occupancy level is 60%. The number of failing task sets rapidly increases as the number of components is increased too. Other occupancy levels are not shown because their schedulability start failing as soon as the number of components starts increasing.



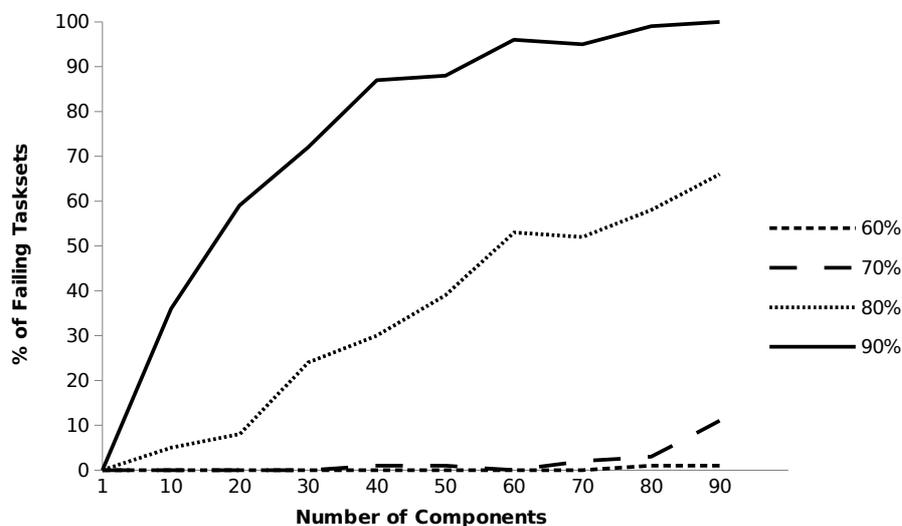
**Figure 50:** Pessimistic replacement model. Percentage of failures regarding processor load and number of components

### *Selective replacement model*

To increase the number of task sets where the component replacement task can be applied while reducing the number of failures, the selective replacement model is applied. This replacement model corresponds to Equations (22) and (23).

After the task set is generated the replacement task parameters are calculated; Equation (21) is used to calculate its cost ( $C_r$ ), and the period ( $T_r$ ) is calculated using the Equation (24) with a desired 5% of processor time reserved for component replacements. This replacement task is added to the task set and the schedulability test are performed applying the Equation (22). Results are shown in Figure 51.

It is appreciated that the amount of failing task sets is enormously reduced. At a 60% of CPU usage the addition of the replacement task barely affects the schedulability of the task sets. At 80% the system schedulability seems to be gradually reduced as the number of components is increased. At 90% of processor load the schedulability of the system is significantly affected as the number of components increases. This effect seems to be closely related to the number of components and the processor load, given that other processor loads are almost not affected.



**Figure 51:** Selective replacement model. Percentage of failures regarding processor load and number of components

The difference between the pessimistic and the selective mode can be appreciated. The pessimistic replacement model allows replacing every component at any time when it executes but at the cost of greatly reducing the schedulability of the system. The selective model allows adjusting the number of replacements that can be performed in a period according to the specific application needs. The number of schedulable systems when this model is applied determines that the application of this model is really feasible.

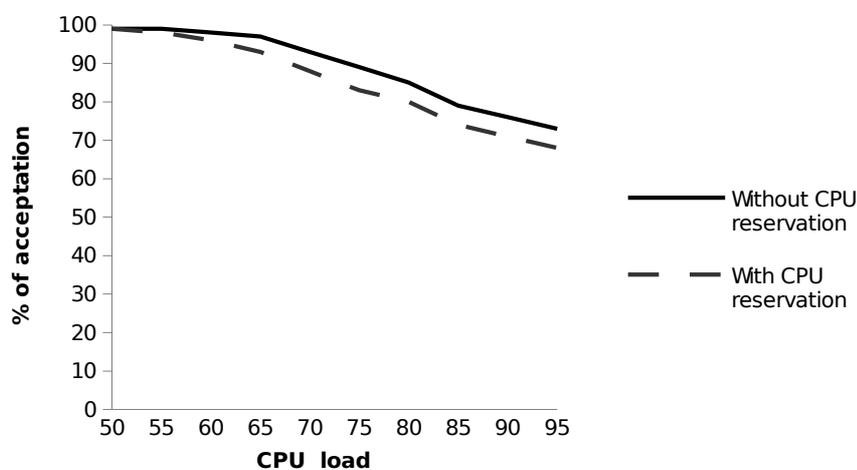
### *Acceptance of replacements*

A variation of this test is presented in Figure 52 to show a different point of view of the interference generated by the addition of the replacement task to the system. Two task sets are maintained with the same tasks. One of them includes the replacement task while the other does not. Task sets are initially empty and components are added to the task set one by one. The replacement task is updated according the characteristics of the added task. Once a new task is added to the set the acceptance test is performed. Components are continuously added to the

system while it is schedulable. The final result is the number of components that can be added to the system before it stops being schedulable.

Figure 52 shows the effect in the percentage of schedulable tasks including the replacement task. Task sets generated range from a processor load of 50% to 95%. For the Response Time Analysis Equations (19) and (50) are applied. Equation (19) is applied for the acceptance test that does not include the replacement task.

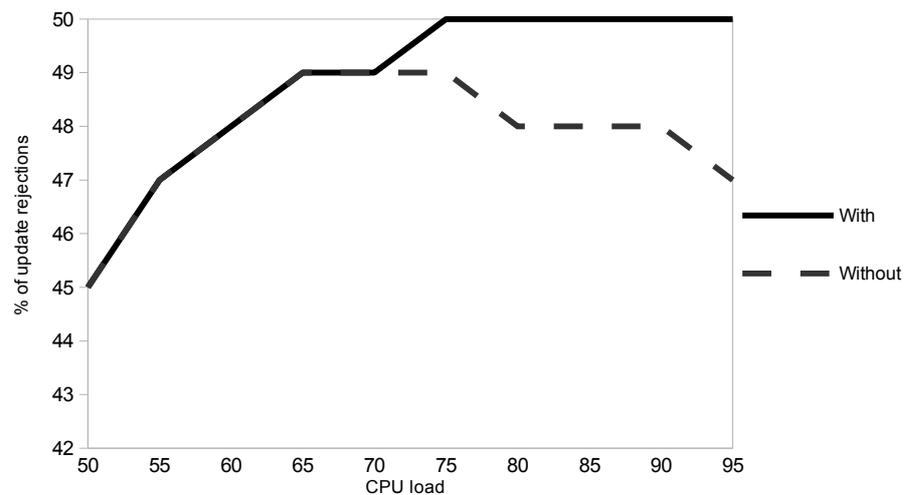
Even at 95% of occupancy the difference in the number of accepted tasks is never greater than the 5% compared to the same task set without processor time reserved for component replacements. This is a small difference. This means that the interference generated by the addition of the reference task is also very small.



**Figure 52:** Acceptance percentage of tasks with and without CPU time reservation for component replacement

An acceptance test of component updates was also created. The previous tests measured the effect of reserving processor time for the component replacement in the execution of a task set and on the effect that failures have on a given task set. The next test measures the number of rejected component replacements in a system that is continuously being updated, and it focuses on the effect of the acceptance test of a single component replacement.

In this case, the system already has a set of running tasks and new components are created to update already existing components in the system. Two component sets are maintained throughout the execution: one with a component replacement task and another without it. A running component to be replaced is randomly selected.



**Figure 53:** Rejection percentage in execution with & without processor reservation for component replacement task

The same selected component is replaced in both sets. The resulting number of accepted component replacements in both cases is compared. Figure 53 shows the percentage of rejected new component updates in the system. In higher CPU loads than 70% half of the component updates are rejected. This effect is due to the fact that components are generated randomly. Half of them will use more processor time than the one to be updated and half of them not.

The *anomaly* of rejections decreasing when no replacement task is included in the task set is due to the free processor time that it is not used by this updating task. This means that actually the processor load of this task set is lower than the other one, so more updates can be accepted compared to the other set.

There is also some difference between the results of reserving or not reserving processor time for the component replacement task. This means that a safe component replacement can be achieved at a very small cost.

### **Replacement times**

To test the time required by the execution of the replacement task, components are randomly generated and replaced. The number of component bindings varies from 2 to 5. For these tests, component data is modeled as an object, and the copy of the state is implemented as a copy of the object to the new component instance. Only the time required to replace a component is measured. The replacement task and the testing process are the only running threads in the system. Components are stopped to reduce interferences in the replacement execution.

**Table 9:** Generic tests replacement times

Average replacement time	0.242 ms
Maximum replacement time	1.559 ms

Table 9 shows that the average time needed for a component replacement is 0.242 ms. It will

vary according to additional complexities of the used component framework, as the method used to copy the state or the size of the component state.

### *Acceptance test times*

The same tests were used to calculate the time required by the implementation for the acceptance test. Beside the RTA tests, a *Response-Time Upper Bound* (RUB)[35] has also been tested. RUB significantly reduces the computation time of the schedulability test at the cost of accepting a smaller number of components in both cases (with and without the component replacement task). Given that most of the acceptance test cost is due to the schedulability test, the use of RUB reduces the execution time of the component acceptance process.

**Table 10:** Component update Acceptance Test Times

Average acceptance test time (RTA)	552 ms
Average acceptance test time (RUB)	71 ms

Table 10 shows the difference in the use of RTA and RUB to test the schedulability. The average time for the acceptance test using RTA is 552 milliseconds, while for RUB it is 71 milliseconds. Significant amount of time is saved using RUB, as predicted, due to its lower computational cost. This also represents some loss of precision and slight increase in the number of rejected components [35].

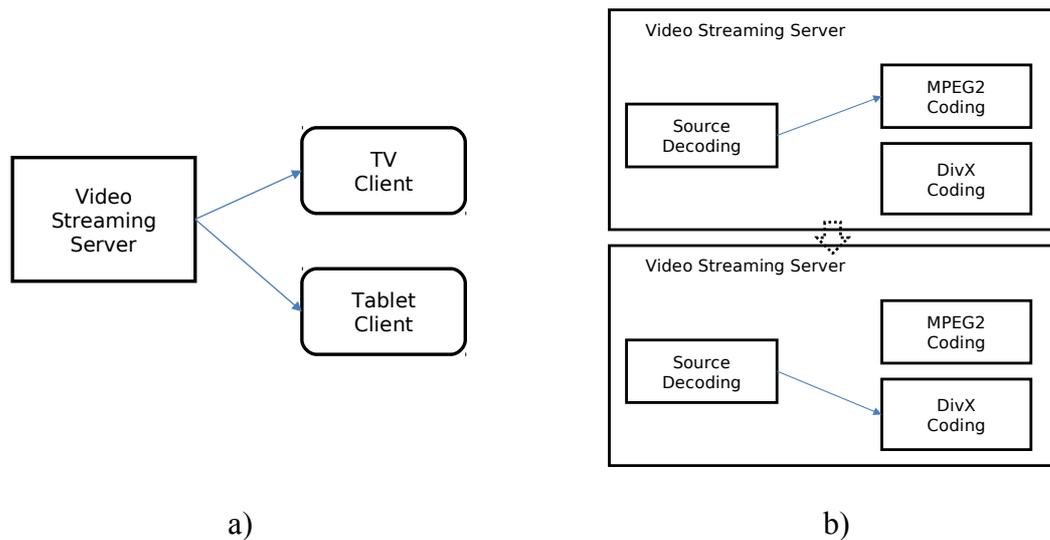
## **6.2 Multimedia scenario implementation**

A multimedia scenario has been created to test the framework implementation using RTSJ and the proposed replacement task. This scenario is based in the use of a video streaming application in a server. This server should be able to:

- Provide video streaming to every rendering client in a domestic environment (i.e., TV, smartphone or tablet).
- Transcode the video source to different video formats depending on the client device to play the content (see Figure 54a).
- Change of target client at run-time without stopping the video streaming, adapting the video format to the format required by the new client.

The objective is that a user can change at any time the device used to watch the video content. This change is done through the replacement at run-time of the component in charge of transcoding to the video format supported by the corresponding target device. A source video file decoding component and two video coding components are used for this scenario. A MPEG2 and a DivX video format coding components are used to send video to a TV device or a tablet device respectively.

A component replacement is required when the target device and the content format are changed. Besides redirecting the video to the new device, the content coding is changed at run-time. Video reproduction is continued from the same point in time as source decoding component is not stopped. This is possible due to the characteristics of the replacement process. The replacement is performed on time without affecting the execution of the rest of components of the system.



**Figure 54:** Domestic scenario (a) and Video Streaming Server components (b)

Table 11 provides scheduling information of used components in the streaming server. Generating video at a rate of 24 frames per second the periodicity of components will be 50 ms. Each component is, then, assigned a periodicity of 50 ms. The source video decoding task is estimated to work in a range of 3 and 4 ms. A worst case execution time of 4 ms is assigned to that task. Similarly a worst case execution time of 3 ms is assigned to the MPEG2 encoding task and also 3 ms to the DivX encoding task. The deadline for tasks is set to the same time as its periodicity, 50 ms. This implementation is deployed in an Intel Core2 Duo 2.2Ghz CPU working as video server.

**Table 11:** Scheduling parameters of tasks (ms)

Name	$C_i$	$T_i$	$P_i$	$D_i$
Decoding	4	50	3	50
MPEG2	3	50	2	50
DivX	3	50	2	50
<i>Repl. task</i>	5	50	1	50

To test the time required by the execution of the component replacement, the three components are loaded into the system and continuously replaced. Coding components are being replaced 100. A replacement request is sent every 2 seconds of work. No state copy is applied in this case because the coding components have to generate a new complete frame every time it they are executed and the information specific to a code is useless for the other one.

Replacement task parameters are previously adjusted to an execution periodicity of 1 second. A worst execution time of 5 ms is assigned to the update task be able to measure the time that the implementation actually takes to replace the component. A 5 ms deadline is also established. The times actually needed to replace the components are in the Table 11. The Jamaica Java Virtual Machine has been used to provide real-time support for Java, using the RTSJ specification API.

In Figure 55 it is detailed the code to execute the multimedia application test. Initially both the input decoder component and a MPEG2 rendering component are registered in the OSGi platform. The service tracker described in Section 5.3 is in charge of adding both components to the framework registry and activating them. A 3 seconds delay is inserted before the stress test is

started. The stress test consists of executing 100 consecutive replacements of the MPEG2 and DivX components. A 2 seconds delay is inserted between replacement requests.

As described in Section 5.3, direct access to the framework registry is used for the replacement requests as the OSGi API does not provide support for replacements.

```

Hashtable<String, Object> prop = new Hashtable<String, Object>();
ComponentReg registry = ComponentReg.getInstance();

prop = new Hashtable<String, Object>();
prop.put("RTDESC", streaming.tests.components.Input.desc);
Input data_input = new Input();
context.registerService(Input.class, data_input, prop);

prop.put("RTDESC", streaming.tests.components.MPEG2Render.desc);
Render render = new MPEG2Render();
context.registerService(MPEG2Render.class, render, prop);

try {
    Thread.sleep(3000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

for (int i = 1; i < 100; i++) {

    render2 = new DivXRender();
    registry.registerComponent(DivXRender.class,
        DivXRender.desc);
    registry.replace(MPEG2Render.class.getName(), render2);

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    render = new MPEG2Render();
    registry.registerComponent(MPEG2Render.class,
        Render.desc);
    registry.replace(DivXRender.class.getName(), render);

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

**Figure 55:** Multimedia scenario testing

Replacement times, shown in Error: Reference source not found, get increased in a real scenario regarding those obtained in the previous tests in Table 9. The time required to replace the components is affected by the execution of running components and the execution of the garbage collector of the virtual machine. Replacement times are bounded assuring a valid execution of the tests.

**Table 12:** Multimedia scenario replacement times

Average replacement time	0.738 ms
Maximum replacement time	2.001 ms
<b>Replacement deadline</b>	<b>5.000 ms</b>

### 6.3 Summary

The interference generated by the inclusion of the replacement task architecture and its execution is tested with different results:

- The passive replacement model highly impacts in the schedulability of the system. The time required to be able to apply a replacement after the execution of every component is only applicable if the processor load or the number of active components is small.
- The impact of the selective replacement model on the schedulability can be adjusted. As shown in the tests, a 5% of processor capacity reserved for the replacement task has a small impact in the system. The number of components to be eliminated for the system to be schedulable is also minimal.
- The on-line execution of the acceptance test is still costly. The application of methods to reduce the time required to accept a replacement request is encouraged. A possible optimization is comparing the timing parameter of the of the new and old components, and accepting the new component if the new execution time ( $C_i$ ) is smaller or equal to the old one. Period and deadline of old and new components should maintain their values to avoid the schedulability analysis.

The replacement task can be directly applicable on soft real-time systems as multimedia applications. The multimedia application supports the variability generated by the virtual machine and the garbage collector.

It is noticed that the execution time of the replacement task varies depending on the conditions of the system. The execution of components make the garbage collector to be activated and interfere in the replacement task.



# Chapter 7

## Conclusions

---

Finally, general conclusions of the developed work are presented in this chapter. Benefits of the described proposal are summarized. Future works are also proposed.

### **7.1 General conclusions**

Merging real-time system with dynamic systems requires to provide mechanisms that enable the determination of the time bounds of the different execution phases of system processes to achieve safe execution during normal operation and also during transitions. In this work, it is provided a method for real-time systems to support component replacements at run-time. A replacement described as the seamless substitution of a running component instance by a new instance in its place. This represents a major contribution to real-time systems where dynamism is reduced to minimum and the most extended approach for dynamism is the use of mode changes. This component replacement allows provide the capability to, not only updating and fixing errors of running components without stopping the platform, but also providing new functionality not previously contemplated.

When using component technology, the time bounds for component switching must be calculated by providing adequate timing properties to the components in an efficient component model. In this work, safety is provided to component's replacement by offering a simple component model that integrates the required temporal properties to enable schedulability analysis of the system; it mainly integrates the component replacement time in the overall timing costs calculation of the system as a way to reserve processor time for safe scheduling of the replacements.

After comparing different proposals with real-time component-based characteristics and different levels of dynamism, it is determined that none of them offers a complete of safe

solution for component replacements at run-time. A complete solution is proposed here composed of the following parts:

- A simple model for components containing common elements to most of the component frameworks. Real-time properties of the components are also contemplated.
- A component replacement method. This method is designed to be bounded in time and based in the known characteristics of the components, i.e., their internal state and bindings to other components. This replacement has to be performed without interruptions by the execution of components to maintain the correct execution of the replaced component.
- Two strategies for component replacement scheduling. One of the replacements scheduling approach allows replacing every component every time it is executed at the cost of using large amounts of processor time. The other approach allows replacing components at specified safe time instants. This time reserved for replacements can be adjusted to the application needs.

For component replacements to be performed, other elements are incorporated in the framework:

- A component loading task. Components have to be put on memory, available for the replacement task to replace running components.
- A component unloading task. To free resources eliminating old stopped components from memory.
- A method to calculate the WCET of component tasks at run-time. The WCET of component tasks depends on the running components at a given instant of time. So it has to be calculated at run-time.
- An acceptance test that performs schedulability analysis based on the WCET of components at run-time. It also takes into account the reserved time for the replacement task.

The described framework has been implemented in Java. RTSJ has been used to provide support for real-time threads. Although the Java platform introduces some performance issues related to the use of virtual machine and the garbage collector, the implementation is designed so that the execution of tasks as the replacement task are bounded in time. This allows providing safe replacements in spite of the low performance of Java code.

The implementation of the framework in Java also introduces new complexities in the component life cycle. A component registry is implemented. Components are registered before the framework can make use of them. The main characteristics of the Java implementation are:

- A configuration interface is added to the component for the framework to operate with it. It allows the framework starting, stopping and binding the component.
- A real-time replacement task is implemented to execute according to the provided parameters as period, processor occupation, etc.

- Loading and unloading tasks are basically represented by the registering and unregistering operations. Loading and unloading code of the component in and out of memory will be implemented by the OSGi platform.
- Other mechanisms are implemented, as the WCET calculation and the acceptance test. This code is partially implemented by the registry, the component classes and the real-time scheduler.

Finally, this framework is integrated in the OSGi platform. This is a well-known dynamic framework where components can be loaded, unloaded and replaced at any time, with a high level of dynamism. The OSGi platform provides the capability to load and unload code at run-time. Component's code can be loaded making use of these mechanisms. No deadlines have to be applied. This work shows that real-time characteristics can be made compatible with certain level of dynamism.

The validity of this proposal is proved through a set of tests. Generic tests are applied to test some general characteristics of the proposed replacement strategies:

- The number of schedulable tasks in a real-time system is important. The *schedulability* tests show that the reduction of schedulable tasks in the system after the introduction of the proposed method is small. This is affected by the processor time reserved for component replacements. So, this can be adjusted depending on the application needs.
- As described above, the *overhead* introduced in the system by the replacement task is small. It is known and manually adjustable.
- The usability of the system is measured in the *number of replacements* that will can to be performed in the running system. Replacements depends on the schedulability of the new components to replace running components. If the new component has the same timing properties than the component to be replaced (same period, execution cost and deadline) then, the replacement will be accepted. The overhead of the replacement task affects the total number of running tasks, but not the number of replacements.

A specific multimedia scenario is created to test the framework in a configuration close to a real environment. In this case it is a soft real-time scenario. A video server where components are replaced at execution time are set up. This shows the capability of the system to work without being stopped and replacing components seamlessly.

Although it shows that Java is a low performance language even making use of RTSJ, obtained results are valid and successful. This is also due to the use of RTSJ, which guaranties that tasks are correctly scheduled and the replacement task is not interrupted by components because it has a higher priority than components tasks.

In this work is has been shown that it is possible to implement a components framework in Java able to provide dynamism to a real-time system. Beside the WCET calculation at run-time of active components, the main contribution of this work is the capability to replace components that make use of this framework at run-time in a safe manner and transparently, without interrupting their execution.

## 7.2 Future works

It has been shown that the safe replacement of components in a real-time environment is possible. New research lines can be started from this point. Some of them represent enhancements to the described contributions and others are complementary.

- Implementation of the framework in C for performance comparison. The addition of the described strategies for component replacement to an existing real-time environment based on a language that is typically related to real-time systems. This framework may be implemented in languages as C, C++ or Ada. This could prove the reliability of the proposal in hard real-time systems.
- Use of membranes to simplify and reduce replacement times. Membranes can be used to encapsulate the component data and code. This approach may reduce the need to perform rebindings as the component code is not directly connected to the other components. If the component data is also managed by the membrane then the state transfer can also be avoided. The time for a component replacement is reduced to swap the old component code inside the membrane by the new code.
- Support for dual characterization of components as active and passive components, and their repercussion in schedulability of the system. In the proposed model, active and passive components are considered exclusive. Modeling an active component, with an executing thread, and able to provide operations that can be invoked by other components has to be analyzed.
- Design and support replacement of more than one component at a time. The current proposal is designed to replace only one component at a time. If the available free processor time allows it, multiple components may be replaced in a single execution of the replacement task. As described in previous chapters, this replacement task can also be shared with other tasks or operations that may be applied if no replacement is requested.
- Components with internal scheduling do not match their time slot or execution with the one assigned by the general scheduler. In the current proposal components are designed to contain a single task. The implementation of this task is supposed to follow the given timing parameters of the component. Else, the execution of the task may not have finished when a replacement is performed. The new component task will not continue from the same execution point, generating wrong results. Actually, the execution of the task has been interrupted.
- Integration of a QoS Manager or a QoS architecture. This proposal could be integrated in a more complete QoS architecture, as a part dedicated to application reconfiguration and component replacements at run-time.

# Chapter 8

## Conclusiones

---

Finalmente, en este capítulo se presentan las conclusiones generales sobre el trabajo desarrollado. Se realiza un resumen de los beneficios de la propuesta descrita. También se proponen líneas de investigación futuras.

### **8.1 Conclusiones generales**

La unión de sistemas de tiempo real con sistemas dinámicos requiere proveer mecanismos que permitan determinar los límites de tiempos de las diferentes fases de ejecución de tareas en el sistema para alcanzar una ejecución segura durante la operación normal del sistema así como durante transiciones. En este trabajo se aporta un método para soportar reemplazo de componentes en tiempo de ejecución para sistemas de tiempo real. Se entiende por un reemplazo la substitución de forma transparente de la instancia de un componente en ejecución por una nueva instancia, que ocupará su lugar. Esto representa una contribución importante a los sistemas de tiempo real, donde el dinamismo está reducido al mínimo y el modelo de dinamismo mas extendido es el uso de cambios de modo. El reemplazo de componentes permite aportar la capacidad, no solo actualizar y corregir errores de los componentes en ejecución sin parar la plataforma, sino también incorporar nueva funcionalidad no contemplada anteriormente.

Al usar tecnologías basadas en componentes, los limites de tiempos de ejecución para el reemplazo de componentes deben ser calculados. Los componentes deben aportar sus correspondientes tiempos de ejecución como parte de un modelo de componentes eficiente. En este trabajo se aporta seguridad en el reemplazo de componentes mediante un modelo de componentes simple que integra las propiedades temporales que permiten aplicar un análisis de planificabilidad sobre el sistema. Principalmente se integra el tiempo para reemplazo de componentes en el cálculo de costes de tiempo del sistema como una manera de reservar tiempo

de procesador para que los reemplazos sean correctamente planificados.

Después de comparar diferentes propuestas con características de tiempo real, basadas en componentes y con diferentes grados de dinamismo se ha determinado que ninguna de ellas ofrece una solución completa y segura para el reemplazo de componentes en tiempo de ejecución. Una solución completa se propone aquí, compuesta de las siguientes partes:

- Un modelo simple de componentes que contiene elementos comunes a la mayoría de marcos de componentes. También contempla las propiedades de tiempo real de dichos componentes.
- Un método de reemplazo de componentes. Este método está diseñado para estar limitado en tiempo y basado en el conocimiento de las características de los componentes, como su estado interno y sus conexiones con otros componentes. Este reemplazo tiene que ser aplicado sin interrupciones por la ejecución de componentes para mantener la correcta ejecución de los componentes reemplazados.
- Dos estrategias para la planificación de los reemplazos de componentes. Una de las aproximaciones permite el reemplazo de cada componente cada vez que éste se ejecuta, con el coste de ocupar una gran cantidad de tiempo de procesador. La otra aproximación permite reemplazar componentes en momentos seguros específicos. El tiempo reservado para estos reemplazos puede ser ajustado según las necesidades de la aplicación.

Para que los reemplazos de componentes puedan realizarse otros elementos son incorporados en el marco de componentes:

- Una tarea para la carga de componentes. Los componentes tienen que ser cargados en memoria para que la tarea de reemplazo de componentes pueda reemplazar los componentes en ejecución.
- Una tarea para la descarga de componentes. Para liberar recursos eliminando de memoria antiguos componentes ya parados.
- Un método para calcular el peor tiempo de cómputo de las tareas de los componentes en tiempo de ejecución. El tiempo de cómputo de los las tareas de los componentes depende de los componentes en ejecución en un instante de tiempo dado. Por eso debe calcularse en tiempo de ejecución.
- Un test de aceptación que realiza análisis de planificabilidad basados en los peores tiempos de cómputo de los componentes en tiempo de ejecución. También tiene en cuenta el tiempo reservado para la tarea de reemplazo de componentes.

El marco de trabajo descrito ha sido implementado en Java. Se ha utilizado Java para soportar el uso de hilos de tiempo real. Aunque la plataforma Java introduce algunos problemas de rendimiento relacionados con el uso de una máquina virtual y un recolector de basura la implementación está diseñada para que la ejecución de tareas como la tarea de reemplazo puedan estar limitadas en tiempo. Esto permite la aplicación de reemplazos de forma fiable a pesar del bajo rendimiento del código en Java.

La implementación del marco de componentes en Java también introduce nuevas complejidades en el ciclo de vida de los componentes. Se ha implementado un registro. Los componentes son registrados antes de que el marco pueda hacer uso de ellos. Las principales características de la implementación en Java son:

- Se ha añadido una interfaz de configuración a los componentes para que el marco de trabajo pueda operar con ellos. Este interfaz permite al marco arrancar, parar y enlazar componentes.
- Se ha implementado una tarea de reemplazo en tiempo real para ser ejecutada de acuerdo a unos parámetros dados, como una periodicidad, una ocupación de procesador, etc.
- Las tareas de carga y descarga de componentes están básicamente representadas por las operaciones de registro y desregistro. La carga y descarga de código de los componentes de la memoria es implementado por la plataforma OSGi.
- También se implementan otros mecanismos, como el cálculo del peor tiempo de cómputo y el test de aceptación de componentes. Este código está parcialmente implementado por el registro, las clases de los componentes y el planificador de tiempo real.

Finalmente, este marco de trabajo está integrado con la plataforma OSGi. Ésta es una extendida plataforma dinámica donde componentes pueden ser cargados, descargados y reemplazados en cualquier momento, con un alto grado de dinamismo. La plataforma OSGi aporta la capacidad de cargar y descargar código en tiempo de ejecución. El código de los componentes es cargado haciendo uso de estos mecanismos. Este trabajo muestra que características de tiempo real se pueden hacer compatibles con cierto grado de dinamismo.

La validez de esta propuesta es probada a través de un conjunto de tests. Se aplican tests genéricos para comprobar algunas características generales de las estrategias de reemplazo propuestas:

- El *número de tareas planificable* en un sistema de tiempo real es importante. Los tests de planificabilidad muestran que la reducción del número de tareas planificables en el sistema tras la introducción de los métodos propuestos es pequeña. Esto depende del tiempo de procesador reservado para reemplazo de componentes. Por lo tanto, esto puede ser ajustado dependiendo de las necesidades de la aplicación.
- La *sobrecarga* introducida en el sistema por la tarea de reemplazo es pequeña. Es conocida y configurable.
- La *usabilidad* del sistema se mide en función del número de reemplazos que el sistema pueda llevar a cabo durante su ejecución. Los reemplazos dependen de la planificabilidad de los nuevos componentes que reemplazarán a componentes en ejecución. Si los nuevos componentes tienen los mismos tiempos de ejecución que los reemplazados (mismo periodo, tiempo de cómputo y plazo de ejecución) entonces el reemplazo será aceptado. La sobrecarga de la tarea de reemplazo afecta al número total de tareas en ejecución, pero no el número total de reemplazos.

Un escenario específico de multimedia ha sido creado para probar el marco de trabajo en una

configuración cercana a un entorno real. En este caso es un escenario de tiempo real flexible. Se ha configurado un servidor de video donde los componentes son reemplazados en tiempo de ejecución. Esto muestra la capacidad del sistema de trabajar sin ser parado, mediante el reemplazo transparente de componentes.

Aunque este escenario muestra que Java es un lenguaje de bajo rendimiento incluso haciendo uso de RTSJ, los resultados obtenidos son validos y satisfactorios. Esto en parte es debido al uso de RTSJ, que garantiza que las tareas son correctamente planificadas y la tara de reemplazo no es interrumpida por componentes debido a que posee una prioridad mayor que las tareas de los componentes.

En este trabajo se ha desmostrado que es posible implementar un marco de componentes en Java capaz de aportar dinamismo a sistema de tiempo real. Además del cálculo del tiempo de cómputo en el peor de los casos en tiempo de ejecución de los compontentes activos, la mayor contribución de este trabajo es la capacidad de reemplazar componentes que hacen uso de este marco en tiempo de ejecución de una manera segura y transparente, sin interrumpir su ejecución.

## **8.2 Trabajos futuros**

Se ha demostrado que los reemplazos de componentes de forma segura en entornos de tiempo real son posibles. Nuevas lineas de investigación pueden ser comenzadas desde este punto. Algunas de ellas representan mejoras con respecto a las contribuciones realizadas en este trabajo y otras son complementarias:

- La implementación del marco de trabajo en C para realizar comparaciones de rendimiento. El añadir las estrategias de reemplazo de componentes descritas a un entorno de de tiempo real ya existente basado en un lenguaje típicamente relacionado con sistemas de tiempo real. Este marco puede estar ya implementado en lenguajes como C, C++ o Ada. Esto aportaría información sobre las posibilidades de esta propuesta en sistemas de tiempo real crítico.
- El uso de membranas para simplificar y reducir los tiempos de reemplazo. Se pueden usar membranas para encapsular los datos y código de los componentes. Esta aproximación puede reducir la necesidad de realizar reconexiones ya que el código del componente no está directamente conectado con otros componentes. Si los datos de los componentes son gestionados por la membrana también la transferencia de estado puede ser evitada. El tiempo para reemplazar un componente es reducido al intercambiar sólo el código de componente dentro de la membrana por el nuevo código.
- Soporte para una caracterización dual de los componentes como activos y pasivos, así como su repercusión en la planificabilidad del sistema. En el modelo propuesto se consideran exclusivamente componentes activos y pasivos. La capacidad de modelar un componente activo, con hilo de ejecución, y capaz de aportar operaciones que puedan se invocadas por otros componentes todavía tiene que ser analizada.
- Diseño y soporte para el reemplazo de mas de un componente a la vez. La propuesta actual está diseñada para realizar el reemplazo de un único componente cada vez. Si el

tiempo de procesador disponible lo permite, múltiples componentes podrían ser reemplazados en cada ejecución de la tarea de reemplazo. Tal y como se ha descrito en los capítulos anteriores, el tiempo de la tarea de reemplazo también puede ser compartido por otras tareas u operaciones que puedan ser aplicadas si no se han realizado peticiones de reemplazo.

- Componentes con planificaciones de ejecución internas pueden no coincidir con sus tiempos de ejecución asignados por el planificador general. En la propuesta actual los componentes están diseñados para contener una única tarea. Se asume que la implementación de dicha tarea sigue los parámetros de tiempo del componente. En caso contrario la ejecución de la tarea puede no haber finalizado en el momento de realizar su reemplazo. La tarea del nuevo componente no continuará desde el mismo punto de ejecución, generando resultados incorrectos. En realidad la ejecución de la tarea habría sido interrumpida.
- Integración de un gestor o arquitectura de calidad de servicio (QoS). Esta propuesta puede ser integrada en una arquitectura QoS mas completa, como una parte dedicada a la reconfiguración de la aplicación y el reemplazo de componentes en tiempo de ejecución.



## Chapter 9

### Bibliography

---

- [1] M. García-Valls, P. Basanta-Val, and I. Estevez-Ayres, “Real-time reconfiguration in multimedia embedded systems,” *Consumer Electronics, IEEE Transactions on*, vol. 57, no. 3, pp. 1280–1287, 2011.
- [2] G. Lipari and E. Bini, “A methodology for designing hierarchical scheduling systems,” *Journal of Embedded Computing*, 2005.
- [3] M. García-Valls, A. Alonso, and J. de la Puente, “Mode change protocols for predictable contract-based resource management in embedded multimedia systems,” in *Embedded Software and Systems, 2009. ICESS’09. International Conference on*, 2009, pp. 221–230.
- [4] I. Estévez-Ayres, P. Basanta-Val, M. García-Valls, J. A. Fisteus, and L. Almeida, “QoS-aware real-time composition algorithms for service-based applications,” *Industrial Informatics, IEEE Transactions on*, vol. 5, no. 3, pp. 278–288, Aug. 2009.
- [5] R. Mall, *Real-Time Systems: Theory and Practice*. 2009.
- [6] M. Ben-Ari, *Principles of concurrent and distributed programming*. 2006.
- [7] J. A. Stankovic, “Real-time computing,” *Invited paper, BYTE*, pp.(155-160), 1992.
- [8] E. J. Bruno and G. Bollella, *Real-time Java programming: with Java RTS*. Prentice Hall PTR, 2009.
- [9] D. Isovich and G. Fohler, “Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints,” in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, 2000, pp. 207–216.

- [10] K. K. G. Shin and P. Ramanathan, “Real-time computing: A new discipline of computer science and engineering,” *Proceedings of the IEEE*, vol. 82, no. 1, pp. 6–24, 1994.
- [11] J. A. Stankovic and K. Ramamritham, “What is predictability for real-time systems?,” *Real-Time Systems*, vol. 2, no. 4, pp. 247–254, 1990.
- [12] L. Sha, T. Abdelzaher, and A. Cervin, “Real time scheduling theory: A historical perspective,” *Real-time systems*, 2004.
- [13] K. Tindell, “Fixed priority scheduling of hard real-time systems,” 1994.
- [14] A. Bernstein and P. K. Harter Jr, “Proving real-time properties of programs with temporal logic,” *ACM SIGOPS Operating Systems Review*, vol. 15, no. 5, p. 11, 1981.
- [15] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking for real-time systems,” in *Logic in Computer Science, 1990. LICS’90, Proceedings., Fifth Annual IEEE Symposium on e*, 1990, pp. 414–425.
- [16] T. U. Eindhoven and D. R. Magnificus, *Predictability in Real-Time Software Design*, no. september. 2005.
- [17] R. Muntz and E. Coffman Jr, “Preemptive scheduling of real-time tasks on multiprocessor systems,” *Journal of the ACM (JACM)*, vol. 17, no. 2, pp. 324–338, 1970.
- [18] T. P. Baker, I. T. Programming, A. Shaw, T. H. E. C. Executive, and R. Periodic, “The Cyclic Executive Model and Ada,” 1988.
- [19] A. Burns and A. Wellings, “Real–Time Systems and their programming languages,” pp. 1–47, 1989.
- [20] C. Locke, “Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives,” *Real-Time Systems*, 1992.
- [21] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.
- [22] P. Pedreiras, “Supporting flexible real-time communication on distributed systems,” 2003.
- [23] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithm: Exact characterization and average case behavior,” *Real Time Systems Symposium.*, pp. 166–171, 1989.
- [24] R. E. Schantz and D. C. Schmidt, “Research advances in middleware for distributed systems: State of the art,” *systems: the state of the art: IFIP 17th World*, 2002.
- [25] K. Nilsen, “Issues in the design and implementation of real-time Java,” vol. 1, no. 1, p. 44, 1996.
- [26] A. D. Stoyenko, “The evolution and state-of-the-art of real-time languages,” *Journal of Systems and Software*, vol. 18, no. 1, pp. 61–83, 1992.
- [27] Phillip A. Laplante, *Real-Time Systems Design & Analysis*, 3rd Ed. WILEY-INTERSCIENCE, 2004, p. 505.

- [28] E. Schweitz and N. Raleigh, “Real-Time Languages,” *Citeseer*, pp. 1–7, 1995.
- [29] J. A. Stankovic and R. Rajkumar, “Real-time operating systems,” *Real-Time Systems*, vol. 28, no. 2, pp. 237–253, 2004.
- [30] W. Cedeño, P. A. Laplante, and W. Cedenó, “An Overview of Real-time Operating Systems,” *Journal of the Association for Laboratory Automation*, vol. 12, no. 1, pp. 40–45, Feb. 2007.
- [31] I. of E. and E. Engineers, “Information technology--Portable operating system interface (POSIX).: C language. System application program interface (API),” 1990.
- [32] S. Baskiyar and N. Meghanathan, “A survey of contemporary real-time operating systems,” *INFORMATICA-LJUBLJANA-*, vol. 29, no. 2, p. 233, 2005.
- [33] L. Abeni, “Integrating multimedia applications in hard real-time systems,” *Real-Time Systems Symposium, 1998.*, 1998.
- [34] L. Abeni, “Resource reservation in dynamic real-time systems,” *Real-Time Systems*, 2004.
- [35] L. Abeni, L. Palopoli, and C. Scordino, “Resource reservations for general purpose applications,” *IEEE Transactions on*, 2009.
- [36] M. García-Valls, A. Alonso, and J. A. de la Puente, “A dual-band priority assignment algorithm for dynamic QoS resource management,” *Future Generation Computer Systems*, vol. 28, no. 6, pp. 902–912, 2012.
- [37] M. M. H. P. van den Heuvel, R. J. Bril, S. Schiemenz, and C. Hentschel, “Dynamic resource allocation for real-time priority processing applications,” *2010 Digest of Technical Papers International Conference on Consumer Electronics (ICCE)*, pp. 67–68, Jan. 2010.
- [38] M. Garcia-Valls, A. Alonso Muñoz, J. Ruiz, and A. Groba, “An Architecture of a QoS Resource Manager for Flexible Multimedia Embedded Systems,” in *Proceedings of 3rd International Workshop on Software Engineering and Middleware. LNCS*, 2003, vol. 2596.
- [39] L. Palopoli, T. Cucinotta, and L. Marzario, “AQuoSA—adaptive quality of service architecture,” *Software: Practice and ...*, no. April 2008, pp. 1–31, 2009.
- [40] P. Clements, “A survey of architecture description languages,” *Proceedings of the 8th international workshop on*, 1996.
- [41] N. Medvidovic and R. N. Taylor, “A classification and comparison framework for software architecture description languages,” *Software Engineering, IEEE*, 2000.
- [42] K.-K. Lau and Z. Wang, “Software Component Models,” *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 709–724, Oct. 2007.
- [43] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav, “Composing components and services using a planning-based adaptation middleware,” in *Software Composition*, 2008, pp. 52–67.
- [44] P. C. Clements, “From subroutines to subsystems: Component-based software

- development,” *American Programmer*, vol. 8, pp. 31–31, 1995.
- [45] OMG, “CORBA Component Model Specification,” *Management*, no. April, p. 350, 2006.
- [46] OMG, “Common Object Request Broker Architecture (CORBA) Specification,” 2011.
- [47] E. Bruneton, T. Coupaye, and J. B. Stefani, “Recursive and dynamic software composition with sharing,” in *Proc. Seventh Int’l Workshop Component-Oriented Programming (WCOP ’, 2002*, vol. 02.
- [48] R. Van Ommering, F. van der Linden, J. Kramer, and J. Magee, “The Koala component model for consumer electronics software,” *Computer*, vol. 33, no. 3, pp. 78–85, 2000.
- [49] E. Bruneton, R. Lenglet, and T. Coupaye, “ASM: a code manipulation tool to implement adaptable systems,” *Adaptable and extensible component systems*, vol. 30, 2002.
- [50] C. Escoffier, R. S. Hall, and P. Lalanda, “iPOJO: An Extensible Service-Oriented Component Framework,” in *Services Computing, 2007. SCC 2007. IEEE International Conference on*, 2007, pp. 474–481.
- [51] M. Alia, V. S. W. Eide, N. Paspallis, F. Eliassen, S. Hallsteinsen, and G. A. Papadopoulos, “A utility-based adaptivity model for mobile applications,” in *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, 2007, pp. 556–563.
- [52] OMG, “UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems,” vol. 15, no. November, 2009.
- [53] I. R. Quadri, A. Muller, S. Meftali, and J. Dekeyser, “MARTE based design flow for Partially Reconfigurable Systems-on-Chips,” *Design*, 2009.
- [54] T. Arpinen, E. Salminen, T. D. Hämäläinen, and M. Hännikäinen, “MARTE profile extension for modeling dynamic power management of embedded systems,” *Journal of Systems Architecture*, pp. 1–11, Feb. 2011.
- [55] A. Charles, F. Mallet, and R. De Simone, “Time modeling in MARTE,” 2007.
- [56] C. Mallet, “Clock Constraints in UML/MARTE CCSL,” 2008.
- [57] J. Riihimäki, P. Kukkala, T. Kangas, M. Hannikainen, T. D. Hamalainen, J. Riihimäki, and M, “Interfacing UML 2.0 for Multiprocessor System-on-Chip Design Flow,” *2005 International Symposium on System-on-Chip*, pp. 108–111, 2005.
- [58] D. Garlan, “An introduction to software architecture,” *Software Engineering and Knowledge*, 1993.
- [59] R. Taylor and N. Medvidovic, “Architectural styles for runtime software adaptation,” *Software Architecture, 2009*, 2009.
- [60] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime software adaptation: framework, approaches, and styles,” in *Companion of the 30th international conference on Software engineering*, 2008, pp. 899–910.

- [61] J. Magee and J. Kramer, “Dynamic structure in software architectures,” *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 3–14, 1996.
- [62] I. C. S. S. C. Committee, “IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries,” 1991.
- [63] F. Akkawi, K. Akkawi, and A. Bader, “Software adaptation: A conscious design for oblivious programmers,” *Aerospace*, 2007.
- [64] C. Canal, J. Murillo, and P. Poizat, “Software adaptation,” *L’objet*, vol. 12, no. 1, pp. 9–31, 2006.
- [65] W. Li, “QoS Assurance for Dynamic Reconfiguration of Component Based Software Systems,” *IEEE Transactions on Software Engineering*, 2011.
- [66] J. Almeida and M. Wegdam, “Transparent dynamic reconfiguration for CORBA,” in *Distributed Objects and Applications*, 2001, pp. 197–207.
- [67] J. Kramer and J. Magee, “The evolving philosophers problem: Dynamic change management,” *Software Engineering, IEEE Transactions on*, vol. 16, no. 11, pp. 1293–1306, 1990.
- [68] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D. Hondt, “Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates,” *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 856–868, 2007.
- [69] J. Fox and S. Clarke, “Exploring approaches to dynamic adaptation,” *Evaluation*, pp. 19–24, 2009.
- [70] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, “Composing adaptive software,” *Computer*, vol. 37, no. 7, pp. 56–64, 2004.
- [71] W. L. Hirsch and C. V. Lopes, “Separation of concerns,” 1995.
- [72] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An overview of AspectJ,” pp. 327–354, 2001.
- [73] P. Maes, “Concepts and experiments in computational reflection,” *ACM Sigplan Notices*, pp. 147–155, 1987.
- [74] M. P. Papazoglou and D. Georgakopoulos, “Service-oriented computing,” *Communications of the ACM*, vol. 46, no. 10, pp. 25–28, 2003.
- [75] H. Cervantes and R. S. Hall, “A framework for constructing adaptive component-based applications: Concepts and experiences,” *Component-Based Software Engineering*, pp. 130–137, 2004.
- [76] B. Redmond, “Supporting unanticipated dynamic adaptation of application behaviour,” 2006.
- [77] J. Waldo and K. Arnold, *The Jini Specifications*. Addison-Wesley Longman Publishing Co., Inc., 2000.

- 
- [78] UPnP Forum, “UPnP Device Architecture 1.1,” 2008.
- [79] R. Hall, K. Pauls, S. Mc Culloch, and D. Savage, “OSGi In Action: Creating Modular Applications in Java,” *Manning Publications*, 2010.
- [80] K. Arnold and J. Gosling, “The Java programming language,” 2000.
- [81] P. Tyma, “Why are we using Java again?,” *Communications of the ACM*, 1998.
- [82] D. F. Bacon, P. Cheng, and V. Rajan, “A real-time garbage collector with low overhead and consistent utilization,” in *ACM SIGPLAN Notices*, 2003, vol. 38, no. 1, pp. 285–298.
- [83] A. Nilsson, T. Ekman, and K. Nilsson, “Real Java for real time-gain and pain,” in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, 2002, pp. 304–311.
- [84] A. Ive, “Implementation of an embedded real-time Java virtual machine prototype,” 2003.
- [85] I. AB., “The Cjip java microprocessor,” 2000. [Online]. Available: <http://www.imsys.se>.
- [86] D. D. S. Hardin, “Real-time objects on the bare metal: An efficient hardware realization of the Javatm virtual machine,” *Object-Oriented Real-Time Distributed*, pp. 53–59, 2001.
- [87] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, “Techniques for obtaining high performance in Java programs,” *ACM Computing Surveys (CSUR)*, vol. 32, no. 3, pp. 213–240, Sep. 2000.
- [88] G. Bollella, B. Delsart, and R. Guider, “Mackinac: Making HotSpot™ Real-Time,” *-Oriented Real-Time*, 2005.
- [89] F. Pizlo and J. Vitek, “Memory management for real-time java: State of the art,” in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, pp. 248–254.
- [90] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, “High-level programming of embedded hard real-time devices,” in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 69–82.
- [91] N. Gui, V. De Flori, H. Sun, and C. Blondia, “A framework for adaptive real-time applications: the declarative real-time OSGi component model,” in *Proceedings of the 7th workshop on Reflective and adaptive middleware*, 2008, pp. 35–40.
- [92] 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*, p. 1, 2009.
- [93] G. Coates, “Real-Time OSGi,” 2007. [Online]. Available: <http://www.osgi.org/wiki/uploads/VEG/Aonix-RT-OSGi.ppt>.
- [94] Z. Mahmood and K. Road, “Service Oriented Architecture: Tools and Technologies,” *Language*, pp. 485–490, 2007.
- [95] P. Basanta Val, M. Garcia-Valls, and I. Estevez-Ayres, “Enhancing OSGi with real time

- Java support,” *Software: Practice and ...*, vol. 28911, 2012.
- [96] C. Cheney, “A nonrecursive list compacting algorithm,” *Communications of the ACM*, vol. 13, no. 11, pp. 677–678, 1970.
- [97] T. Richardson, A. J. Wellings, J. A. Dienes, and M. Diaz, “Towards Memory Management for Service-Oriented Real-Time Systems,” in *JTRES’10*, 2010, pp. 128–137.
- [98] T. Miettinen, D. Pakkala, and M. Hongisto, “A Method for the Resource Monitoring of OSGi-based Software Components,” *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pp. 100–107, Sep. 2008.
- [99] M. Chung, “Using JConsole to monitor applications,” *Sun Developer Network*, 2004.
- [100] The NetBeans community, “NetBeans Profiler,” 2011. [Online]. Available: <http://profiler.netbeans.org/>.
- [101] “JVM Tool Interface,” 2011. [Online]. Available: <http://download.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html>.
- [102] M. Dmitriev, “Design of JFluid: A Profiling Technology and Tool Based on Dynamic Bytecode Instrumentation.”
- [103] OSGi Alliance, “OSGi Service Platform Release 4 Service Compendium Version 4.2,” 2009.
- [104] M. Pfeffer and T. Ungerer, “Dynamic real-time reconfiguration on a multithreaded java-microcontroller,” 2004.
- [105] R. W. Brennan, M. Fletcher, and D. H. Norrie, “An agent-based approach to reconfiguration of real-time distributed control systems,” *Robotics and Automation, IEEE Transactions on*, vol. 18, no. 4, pp. 444–451, 2002.
- [106] X. Zhang, S. Balasubramanian, R. W. Brennan, and D. H. Norrie, “Design and implementation of a real-time holonic control system for manufacturing,” *Information sciences*, vol. 127, no. 1–2, pp. 23–44, 2000.
- [107] R. Brennan, X. Zhang, and Y. Xu, “A reconfigurable concurrent function block model and its implementation in real-time Java,” *Integrated Computer-Aided*, 2002.
- [108] H. Kopetz, “Component-based design of large distributed real-time systems,” *Control Engineering Practice*, vol. 6, no. 1, pp. 53–60, 1998.
- [109] E. Bruneton, T. Coupaye, and M. Leclercq, “The fractal component model and its support in java,” *Software: Practice*, pp. 1257–1284, 2006.
- [110] T. Bures, P. Hnetyinka, and F. Plasil, “Sofa 2.0: Balancing advanced features in a hierarchical component model,” in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, 2006, pp. 40–48.
- [111] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas, “An efficient component model for the construction of adaptive middleware,” *Middleware 2001*, pp. 1–15, 2001.

- 
- [112] 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," *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pp. 1208–1224, 2004.
- [113] M. Vanneschi and L. Veraldi, "Dynamicity in distributed applications: issues, problems and the ASSIST approach," *Parallel Computing*, 2007.
- [114] A. Tesanovic, J. Hansson, A. Te Usanovic, D. Nyström, and C. Norström, "Aspect-level worst-case execution time analysis of real-time systems compositioned using aspects and components," *Workshop on Real-Time*, 2003.
- [115] P. Comwell and A. Wellings, "Transaction Integration For Reusable Hard Real- Time Components," pp. 166–175, 1997.
- [116] J. Stankovic, "VEST: A toolset for constructing and analyzing component based operating systems for embedded and real-time systems," *University of Virginia TRCS-2000-19*, 2000.
- [117] W. T. Tsai, Y. H. Y. Lee, Z. Cao, Y. Chen, and B. Xiao, "RTSOA: Real-time service-oriented architecture," *2006 Second IEEE International Symposium on Service-Oriented System Engineering (SOSE'06)*, pp. 49–56, Oct. 2006.
- [118] M. Panahi, W. Nie, and K. J. Lin, "A framework for real-time service-oriented architecture," in *2009 IEEE Conference on Commerce and Enterprise Computing*, 2009, pp. 460–467.
- [119] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checco, and F. Rusina, "A real-time service-oriented architecture for industrial automation," *Industrial Informatics, IEEE Transactions on*, vol. 5, no. 3, pp. 267–277, 2009.
- [120] C. McGregor and J. M. Eklund, "Real-time service-oriented architectures to support remote critical care: trends and challenges," in *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, 2008, pp. 1199–1204.
- [121] D. Verma, *Supporting service level agreements on IP networks*. Sams, 1999.
- [122] J. P. Etienne, J. Cordry, and S. Bouzeffrane, "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*, 2006, pp. 218–226.
- [123] A. Plšek, F. Loiret, P. Merle, and L. Seinturier, "A component framework for java-based real-time embedded systems," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, 2008, pp. 124–143.
- [124] 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 *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, 2004, pp. 15–22.
- [125] 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 *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, 2007, vol. 4834, no. 0410218, pp. 41–59.
- [126] R. Alur and G. Weiss, “RTComposer: a framework for real-time components with scheduling interfaces,” in *Proceedings of the 8th ACM international conference on Embedded software*, 2008, pp. 159–168.
- [127] J. Auerbach, D. F. Bacon, D. T. Iercan, C. M. Kirsch, V. Rajan, H. Roeck, and R. Trummer, “Java takes flight: time-portable real-time programming with exotasks,” *ACM SIGPLAN Notices*, vol. 42, no. 7, pp. 51–62, 2007.
- [128] T. Henzinger, B. Horowitz, and C. Kirsch, “Giotto: A time-triggered language for embedded programming,” *Embedded Software*, 2001.
- [129] M. García-Valls, I. Rodríguez Lopez, and L. Fernández Villar, “iLAND: An Enhanced Middleware for Real-Time Reconfiguration of Service Oriented Distributed Real-Time System,” *IEEE Transactions on Industrial Informatics*, no. c, pp. 1–9, 2012.
- [130] D. Isovich and M. Lindgren, “System development with real-time components,” *Pervasive Component-Based*, 2000.
- [131] L. Sha, “Dependable system upgrade,” *Real-Time Systems Symposium*, 1998.
- [132] A. Rasche and A. Polze, “Dynamic reconfiguration of component-based real-time software,” in *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, 2005, pp. 347–354.
- [133] A. Rasche and A. Polze, “ReDAC—Dynamic Reconfiguration of Distributed Component-Based Applications with Cyclic Dependencies,” in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, pp. 322–330.
- [134] M. Wahler, S. Richter, S. Kumar, and M. Oriol, “Non-disruptive large-scale component updates for real-time controllers,” in *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, 2011, pp. 174–178.
- [135] E. Schneider and F. Picioroagă, “Dynamic reconfiguration through OSA+, a real-time middleware,” *International Middleware Doctoral Symposium*, 2004.
- [136] J. Gorinsek, S. Van Baelen, Y. Berbers, and K. De Vlaminck, “Managing quality of service during evolution using component contracts,” in *ETAPS 2003 Workshop on Unanticipated Software Evolution (USE2003), Warsaw, Poland*, 2003, pp. 57–62.
- [137] G. Bollella and J. Gosling, “The real-time specification for Java,” *IEEE Computer*, vol. 33, no. 6, pp. 47–54, 2000.
- [138] H. Cervantes, “Autonomous adaptation to dynamic availability using a service-oriented component model,” *of the 26th International Conference on*, 2004.
- [139] R. Needham, “Denial of service,” *Proceedings of the 1st ACM Conference on*, 1993.
- [140] C. Szyperski, “Component Software: beyond object-oriented software,” *Reading, MA:*

*ACM/Addison-Wesley*, 1998.

- [141] E. Bondarev and M. Chaudron, “Compositional Performance Analysis of Component-Based Systems on Heterogeneous Multiprocessor Platforms,” *Software Engineering and ...*, 2006.
- [142] OMG, “Unified Modeling Language (OMG UML)-Infrastructure(V2. 1.2),” 2009.
- [143] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.
- [144] F. Mallet, R. De Simone, and L. Rioux, “Event-triggered vs . Time-Triggered communications with UML MARTE,” *2008 Forum on Specification Verification and Design Languages*, pp. 154–159, 2008.
- [145] E. Bini and G. C. Buttazzo, “Measuring the performance of schedulability tests,” *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.
- [146] E. Bini, T. H. Cha, P. Richard, and S. K. Baruah, “A Response-Time Bound in Fixed-Priority Scheduling with Arbitrary Deadlines,” *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 279–286, 2009.
- [147] G. C. Buttazzo, “Rate Monotonic vs. EDF: Judgment Day,” *RealTime Systems*, vol. 29, no. 1, pp. 5–26, 2005.