Developing Dynamically Reconfigurable Real-Time Systems with Real-Time OSGi (RT-OSGi)

Thomas Richardson

PhD

The University of York
Computer Science

July 2011

Abstract

By encompassing the concepts of Service-Oriented Architecture and Component-Based Software Engineering, the OSGi Framework enables dynamically reconfigurable Java applications to be developed. Application components can be added, removed and updated during run-time, i.e., without shutting the application down. As a result, dynamically reconfigurable Java applications developed and deployed using the OSGi Framework can maintain high levels of availability to their users even during software maintenance/evolution activities. As a consequence, the application is able to continue to operate (perhaps with some degradation in the quality of service) and thus provide some utility to its users.

Real-time systems have timing requirements in addition to functional requirements. In addition, they have software maintenance/evolution requirements much like any other software application, but are also known to have particularly high availability requirements for either safety or financial reasons. Real-time systems would therefore particularly benefit from the dynamic reconfigurability and resultant high availability offered by the OSGi Framework. However, the OSGi Framework is based on standard Java, which is unsuitable for real-time systems development. The OSGi Framework also lacks a number of features required to support dynamically reconfigurable real-time systems.

In order to address the incompatibility of using the OSGi Framework to develop and deploy dynamically reconfigurable real-time applications, the Real-Time Specification for Java (RTSJ) is investigated as a means of developing real-time OSGi applications, moreover, the OSGi Framework is extended with various features such as temporal isolation, CPU and memory admission control, garbage collection reconfiguration analysis, asynchronous thread termination using asynchronous transfer of control, and a mode change protocol. A prototype OSGi Framework for Real-Time Systems (RT-OSGi) is implemented and used to evaluate the ability for RT-OSGi applications to exhibit both real-time requirements and high availability requirements in the presence of

application dynamic reconfiguration. Furthermore, a case study is discussed, both motivating the need for RT-OSGi, and demonstrating the expressive power of RT-OSGi in a practical setting.

Table of Contents

1 Introduction 15
1.1 Real-Time Systems 15
1.2 Software Maintenance and Availability Requirements 17
1.3 Java and the Real-Time Specification for java (RTSJ) 20
1.3.1 Memory Management 21
1.3.2 Time Values and Clocks 22
1.3.3 Scheduling 23
1.3.4 Asynchrony 24
1.3.5 Resource Sharing 25
1.4 Thesis Motivation and Thesis Hypothesis 26
1.5 Thesis Contributions 27
1.6 Thesis Structure 28
2 Underlying Concepts of the OSGi Framework 30
2.1 Component-Based Software Engineering (CBSE) 30
2.1.1 Motivation for CBSE 32
2.1.2 Limitations of CBSE 33
2.2 Service-Orientation 36
2.2.1 Motivation for Service-Orientation 38
2.2.2 Limitations of Service-Orientation 39
2.3 Comparison of Service-Orientation and CBSE 41
2.4 Application to Real-Time Systems 43
2.4.1 Real-Time CBSE 43
2.4.1.1 Non-RTSJ CBSE 43
2.4.1.2 CBSE and the RTSJ 44
2.4.2 Real-Time SOA 45
2.5 Other Related Work on Dynamic Reconfiguration 48
2.6 Summary 51
3 Overview of the OSGi Framework and its Real-Time Extensions (RT-OSGi) 53
3.1 OSGi Alliance and the OSGi Framework 53
3.2 Architecture of the OSGi Framework 60
3.2.1 Security Layer 61
3.2.2 Module Layer 61
3.2.3 Life Cycle Layer 62
3.2.4 Service Layer 63
3.3 Using the OSGi Framework 65
3.3.1 Developing a Component 65
3.3.2 Controlling the Life Cycle of a Component 67
3.3.3 Registering and Obtaining Services 68
3.3.4 Services with State 71
3.3.5 Dynamic Availability 72
3.4 Applications of the OSGi Framework 73
3.4.1 Service Gateways 73
3.4.2 Other Applications 75
3.5 Motivation in Real-Time Systems 76

3.6 Real-Time OSGi (RT-OSGi) ... 79
3.6.1 Issues ... 80
3.6.1.1 Global and Local View – Priority Assignment ... 80
3.6.1.2 Worst-Case Execution-Time (WCET) Analysis ... 81
3.6.1.3 Scheduling – Dynamic Availability ... 82
3.6.1.4 No Temporal Isolation – Denial of Service Attacks ... 82
3.6.1.5 GC Reconfiguration and Reconfiguration Analysis ... 82
3.6.1.6 OSGi Framework is not Real-Time ... 83
3.6.2 Other Approaches to Supporting Real-Time Applications using the OSGi Framework ... 85
3.7 Summary ... 89
4 Temporal Isolation and Worst Case Execution-Time (WCET) Analysis .. 90
4.1 Temporal Isolation ... 90
4.1.1 Providing Temporal Isolation ... 91
4.1.1.1 Time-Slicing ... 91
4.1.1.2 Execution-Time Servers ... 92
4.1.1.3 Comparison of Time Slicing and Execution-Time Servers ... 95
4.1.2 RT-OSGi Temporal Isolation Extensions ... 96
4.1.2.1 Servers in RT-OSGi ... 98
4.1.2.2 Simulating Hierarchical Scheduling in RT-OSGi ... 109
4.2 Worst-Case Execution-Time (WCET) Analysis ... 110
4.2.1 Introduction ... 110
4.2.2 WCET Issues ... 113
4.2.2.1 Service Execution ... 113
4.2.2.2 Synchronous Event Handling ... 117
4.2.2.3 Service Factories ... 117
4.2.2.4 Component Activation and Deactivation ... 118
4.2.3 Solving the WCET Problems of OSGi ... 119
4.2.3.1 WCET Contracts ... 121
4.2.3.2 Adaptive Resource Reservation ... 124
4.2.3.3 Asynchronous Event Handling ... 126
4.3 Summary ... 128
5 Admission Control ... 130
5.1 Introduction ... 130
5.2 Component-Derived and User-Derived Life Cycle Operations ... 132
5.3 Installing Components ... 134
5.3.1 Execution-Time Server Parameter Selection ... 135
5.3.2 Schedulability Analysis ... 136
5.3.3 Priority Assignment ... 138
5.4 Updating Components ... 144
5.5 Starting Components ... 147
5.6 Stopping and Uninstalling Components ... 148
5.6.1 Controlling the Life-Time of Threads ... 148
5.6.2 Resource Reclamation ... 153
5.7 Blocking and Life Cycle Operations ... 153
5.8 Summary ... 156
6 Memory Management ... 159
6.1 Introduction ... 159
6.2 Current Real-Time Garbage Collectors ... 161
6.3 Garbage Collector Reconfiguration ... 166

6.3.1 Garbage Collection Reconfiguration Analysis ... 170
6.3.1.1 Estimating Garbage Collection Work ... 173
6.3.1.2 Calculating Garbage Collector Parameters ... 176
6.3.1.3 Estimating Garbage Collection Cycle Time ... 178
6.4 Memory Admission Control ... 179
6.5 Example of Applying Garbage Collection Reconfiguration Analysis 183
6.6 Memory Allocation Enforcement ... 185
6.7 Summary ... 192
7 Case Study: Chronic Disease Management ... 193
7.1 Introduction ... 193
7.1.1 Introduction to Short Term Acute Complications in Chronic Disease Management ... 194
7.2 Application Requirements ... 195
7.2.1 Application Reconfiguration Requirement ... 195
7.2.2 High Application Availability Requirement ... 198
7.2.3 Real-Time Requirement ... 199
7.3 Case Study Design ... 199
7.3.1 Overview ... 199
7.3.2 Sensor Components ... 201
7.3.3 Sensor Interpreter Components ... 204
7.3.4 Complex Event Processor (CEP) Component ... 205
7.4 Application of the Chronic Disease Management Application – Monitoring for Hypoglycaemia in Diabetes ... 208
7.4.1 Dynamic Reconfiguration Examples ... 210
7.4.1.1 Installing New Components – Admission Control and GC Reconfiguration ... 210
7.4.1.2 Replacing Existing Components – Effect on Application Timing Constraints and the RT-OSGi Mode Change Protocol ... 215
7.5 Summary ... 224
8 RT-OSGi Evaluation ... 226
8.1 RT-OSGi Prototype Implementation ... 226
8.1.1 Deploying RT-OSGi Applications – Apache Felix Modifications and Extensions ... 228
8.1.2 Developing RT-OSGi Applications – RTSJ Class Extensions and OSGi Manifest Extensions ... 229
8.2 RT-OSGi Evaluation ... 234
8.2.1 Comparison of Thread Response Times in a Standard JVM and in an RTSJ JVM ... 235
8.2.2 Dynamic Reconfiguration Effects on Application Availability and Application Timing Constraints ... 239
8.2.2.1 Component Installation ... 240
8.2.2.2 Component Removal ... 242
8.2.2.3 Component Replacement ... 243
8.2.3 Execution-Time Overhead of Dynamic Reconfiguration ... 249
8.2.4 RT-OSGi Pessimism and Application Schedulability ... 256
8.2.4.1 Temporal Isolation Pessimism ... 257
8.2.4.2 Server Parameter Selection Pessimism ... 258
8.2.4.3 GC Reconfiguration Analysis Pessimism ... 261
8.2.4.4 Component Replacement Pessimism ... 261

8.2.5 Backwards Compatibility with Standard OSGi and the Usability of RT-OSGi 262
8.3 Summary 264
9 Conclusions and Future Work 266
9.1 Thesis Goals and Hypothesis 266
9.2 Overall Conclusions 269
9.3 Future Work 272
9.4 Concluding Remarks 275
References 277

List of Figures

Figure 2.1 Limited Dynamic Reconfigurability of RT-SOA 47
Figure 3.1 OSGi Component State Transitions 55
Figure 3.2 Dynamic Reconfiguration in OSGi 59
Figure 3.3 Layers of the OSGi Framework 61
Figure 3.4 OSGi Overview [80] 64
Figure 3.5 Bundle Activator 66
Figure 3.6 Example of Component Lifecycle Operations 68
Figure 3.7 Obtaining Services 69
Figure 3.8 Providing Multiple Implementations of a Service 69
Figure 3.9 Examining Service Properties 70
Figure 3.10 Obtaining a Service 70
Figure 3.11 Service Factories 71
Figure 3.12 Registering Service Factories 71
Figure 3.13 Service Events 72
Figure **3**.14 Service Gateway 74
Figure 3.15 the Real-time Management Interface 86
Figure 4.1 Hierarchical Scheduling 94
Figure 4.2 Model of Cost-Overrun and Budget Replenishment 102
Figure 4.3 Constructor of the OSGiRTT class 104
Figure 4.4 OSGiPGP Skeleton Class 106
Figure 4.5 Basic Temporal Isolation Scheme of RT-OSGi. 110
Figure 4.6 Intra-JVM Service Model: 114
Figure 4.7 Inter-JVM Service Model: 114
Figure 4.8 Service Execution-Time Contract 123
Figure 4.9 A Stub Service Implementation 123
Figure 4.10 Resource Adaptation 126
Figure 5.1 Execution-Time Server Creation and Initialisation 148
Figure 5.2 the RTSJ's Interruptible Interface 152
Figure 5.3 OSGiRTT Asynchronous Thread Termination Extensions 152
Figure 5.4 Example of an Application Thread in RT-OSGi 152
Figure 5.5 Summary of Life Cycle Operation Extensions in RT-OSGi 158
Figure 6.1 Application Reconfiguration Not Reflected in GC Rate 162
Figure 6.2 Sun Java RTS Model of Garbage Collection 164
Figure 6.3 Subset of FullyConcurrentGarbageCollector class Used to Support Time-Based GC for RT-OSGi 167
Figure 6.4 Implementing Time-Based GC Using Sun Java RTS's FullyConcurrentGarbageCollector Class 170
Figure 6.5 Example of Memory Allocation Enforcement 187
Figure 7.1 Interaction between the Patient's Wireless Wearable Vital Signs Sensors, the RT-OSGi Components, and the Patient's Smartphone 200
Figure 7.2 SensorService Service Interface 201
Figure 7.3 Registering an Implementation of the SensorService 202
Figure 7.4 Discovering an Implementation of SensorService 202
Figure 7.5 The ECG Interpreter Thread's Run Method 203
Figure 7.6 ECGInterpretation Service Implementation 204

Figure 7.7 CEP Asynchronous Event Handler's handleAsycEvent Method Implementation 206
Figure 7.8 Chronic Disease Management Application Architecture 207
Figure 7.9 Steps Taken by Service Requesters during Component Replacement 220
Figure 7.10 Required Transition for the EEG Sensor Component 221
Figure 8.1 Run-Time Environment Used for the RT-OSGi Prototype 228
Figure 8.2 The RTBundle Interface of RT-OSGi 231
Figure 8.3 Additional Manifest Headers Required for Real-Time Components 231
Figure 8.4 Example of Defining the Temporal and Memory Allocation Requirements of a Component's Schedulable Objects 232
Figure 8.5 Relationship Between the RT-OSGi, RTSJ and OSGi Packages 233
Figure 8.6 Response Times of the Chronic Disease Management Application Threads in a Standard JVM and in an RTSJ JVM 237
Figure 8.7 The Possible Schedules for the Example Dynamic Reconfiguration 246
Figure 8.8 Graph Showing Changes in Execution-Time of Install Operation when the Number of Component Deployed Increases 255
Figure 8.9 Pessimism of Server Parameter Selection 260

List of Tables

Table 3.1 Priorities Assigned by Component Developers and by the System..... 81
Table 5.1 Period to Priority Range Mapping Rules 140
Table 5.2 Priority Range Reassignment.. 142
Table 6.1 Reconfiguration Analysis Notations 173
Table 6.2 Server Temporal Specification.. 183
Table 7.1 Temporal Specifications of Installed Components' Servers.............. 211
Table 7.2 Temporal Specification of Blood Glucose/Pressure Component 212
Table 7.3 Response Times for Servers.. 213
Table 7.4 Application Dynamic Reconfiguration Effect on GC Configuration 214
Table 8.1 Response Times in a Standard JVM and in an RTSJ JVM............... 236
Table 8.2 Temporal Specification of the Dynamic Reconfiguration Example Application... 240
Table 8.3 WCET of the Life Cycle Operations in both Standard OSGi and RT-OSGi... 251
Table 8.4 Execution-Time Overhead of Installing a Component with Admission Control 252
Table 8.5 Execution-Times of Sequentially Installing the Chronic Disease Management Application's Components... 255
Table 8.6 Temporal Isolation Handlers' Effect on Application Schedulability. 259
Table 8.7 Temporal Specification of Threads... 260
Table 8.8 Server Parameters and Comparison with CPU Utilisation of Threads 260

List of Equations

Equation 6-1 Calculating the Require GC Cycle Time....................................172
Equation 6-2 Estimating GC Work..175
Equation 6-3 Calculating C_{GC} 177
Equation 6-4 Determining the GC Cycle Time...179
Equation 6-5 Determining Free Memory Requirement of the Application 181

Declaration

I declare that the research work described in this thesis is original work unless otherwise indicated in the text. The research was conducted by me, Tom Richardson, between 2007 and 2011 at the University of York, with the supervision of Professor Andy Wellings. I have acknowledged external sources through bibliographic referencing. Some parts of this thesis have been published in scientific papers or have been accepted to appear in future publications.

In Chapter 4, the work on extending the OSGi Framework with cost enforcement and temporal isolation is based on the following publication **Richardson, T., et al., *Providing temporal isolation in the OSGi framework*, in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*. 2009, ACM: Madrid, Spain.**

In Chapter 5, the CPU admission control protocol and asynchronous thread termination model extension to the OSGi Framework is based on the following publication: **Richardson, T. and A. Wellings. *An Admission Control Protocol for Real-Time OSGi*. in *13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing (ISORC)*. 2010. Seville, Spain.**

In Chapter 6, the GC reconfiguration analysis, memory admission control, and memory allocation enforcement extensions to the OSGi Framework are based on the following publication: **Richardson, T., et al., *Towards memory management for service-oriented real-time systems*, in *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*. 2010, ACM: Prague, Czech Republic.**

In Chapter 7, the chronic disease management case study application is based on the following publication: **Richardson, T. and A. Wellings, On the Road to Real-Time OSGi, to Appear in International Journal of Computer Systems Science and Engineering**

Finally, the work on RT-OSGi in this thesis is summarised in the following book chapter: **Richardson, T. and A. Wellings. RT-OSGi: Integrating the OSGi Framework with the Real-Time Specification for Java, to appear in Distributed, Embedded and Real-time Java Systems, M. Teresa Higuera-Toledano, Editor, 2012:Springer**

1

Introduction

1.1 Real-Time Systems

Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behaviour of these systems depends not only on the value of the computation but also on the time at which the results are produced [1].

There are a number of applications that require real-time computing and these include: chemical and nuclear plant control, automotive applications, flight control, environmental acquisition and monitoring, telecommunication systems, multimedia systems, industrial automation, and robotics etc. Despite these many uses of real-time systems, there are a number of misconceptions about real-time computing [2], the most common misconception is that real-time computing equates to fast computing.

However, in real-time systems, the concept of time is not an intrinsic property of the computing system, but rather it is related to the environment in which the system operates. The emphasis is on computing a result within a deadline in order to remain reactive with its environment i.e. the emphasis is on predictability, as late results are of reduced value and may even be dangerous.

Such predictability is much more important than the absolute speed of the computing system. For example, the absolute speed of a turtle's reactions to stimuli from its natural environment is much slower than that of a cat, yet its reactions are as effective as those of the cat when considering the respective environments of the two [3]. Furthermore, if events are introduced into the cat's environment which evolves more rapidly than the cat can handle, its actions will no longer be effective despite the absolute speed of the cat, e.g. a cat wondering onto a road may not be able to react in time to a speeding car.

In order to ensure that real-time tasks finish execution within their deadline i.e. in order to guarantee the timing requirements of each task, a scientific methodology is required to analyse the set of tasks and to ensure that the tasks satisfy a number of constraints. Without such a methodology, tasks may have an average response time less than their deadline, that is, tasks may in the average case meet their deadlines, but this cannot be guaranteed as no bound can be placed on the worst case response time of tasks

As briefly mentioned, results computed by tasks after their deadline are of reduced value and may even be dangerous. More specifically, deadlines can be soft or hard. If a result has utility even after the deadline has passed, the deadline is classified as soft, otherwise it is hard. If at least one task has a hard deadline where a miss has catastrophic consequences i.e. has a utility less than zero, the system is safety-critical [4].

In addition to deadlines, a task also has a release time, and this is the instant of time at which the task becomes available for execution. The task can be scheduled and executed at anytime at or after its release time. End users are mostly concerned with the length of time from the release time of the task to the instant when it completes (response time) [4]. As a note, in many systems, exact response times aren't relevant provided they are within the deadline of a task. In some systems such as control systems, varying the response times (known as jitter) can cause instability or a jerky behaviour of the controlled system, and therefore this jitter should be minimised.

Tasks which are released in a regular fashion such as every *n* time units are called periodic tasks. If tasks are released in a non-regular fashion, the task is said to be aperiodic. Aperiodic tasks in which consecutive releases are separated by a minimum interarrival time (MIT) are called sporadic [5].

1.2 Software Maintenance and Availability Requirements

Software must be continually adapted else it becomes progressively less satisfactory [6]. According to Parnas [7], "The only programs that don't get changed are those that are so bad that nobody wants to use them. Designing for change is designing for success".

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or to adapt the product to a changed environment, and to modify the functionality of the software in accordance with changing user requirements [8]. These modifications are termed corrective, perfective, and adaptive maintenance respectively [9]. Examples of the need for such maintenance include [10]:

- Faults
- Customer Need
- Competition – from other suppliers in provision and the enemy in operations
- Technology Development and Change (e.g. initial cost, maturation, phase-out)
- Standards (technical, etc.)
- Efficiency
- Architectural optimization
- Obsolescence
- Architectural Decay
- Legislation/Litigation
- Culture
- Community and Industrial Relationship
- Project Changes

In terms of the percentage of the total amount of maintenance activity that occurs in software, the surveys carried out in [11] and [12] show that 18% of the maintenance activity is perfective, 60% is adaptive, and 17% is corrective. This means that, perhaps surprisingly, more of the maintenance effort is spent on adapting the software to changing user requirements e.g. adding new functionality than on correcting errors in the software. The significance of this will be discussed after software availability is introduced below.

In addition to the requirement for maintenance, software applications also have high availability requirements. Software availability can be defined as either the probability that the system will be operable at a specified instant of time (pointwise availability), or as the expected fraction of a given interval of time that the system will be operable (interval availability) [13].

Software typically executes for a phase known as mission-time [14]. This is the time in which the application should not be made unavailable because it is performing some function which is critical to meeting the user requirements. Clearly, when software is unavailable for use during its mission-time, it has no utility. For many types of software then, such unavailability has financial implications. For example, if the software responsible for processing financial transactions in an online retailer becomes unavailable, the company is unable to make any sales until the software becomes available again. As a result, the retailer lose money when ever the system is unavailable. Similarly, when real-time systems are unavailable, they provide no utility and there may be a financial or safety penalty. In these applications, outages translate directly into reduced productivity, damaged equipment, and sometimes lost lives. Therefore both non real-time and real-time systems have a very high availability requirement. However, the need for software maintenance clashes with the high availability of real-time software applications. Taking the system offline for maintenance makes it unavailable for use and has financial and possibly safety implications. Although the software may have other phases known as repair-time and switch-over time, where the software is permitted to be taken offline as its availability during these times is not of the utmost importance, there is still an issue. Delaying maintenance and switchover to the new version of the software

application until the end of mission-time and the beginning of repair and switchover-time may have severe implications because the deployed software will either be faulty, poorly optimised, or no longer meeting the user requirements until the end of mission-time, which could be a considerable length of time away. Of course, one might argue that software maintenance may require so little time that the decrease in software availability is not an issue. However, consider the case where the maintenance activity is small and only results in only 1% downtime i.e. the software availability is 99% during a week. Whilst such availability appears to be good, 99% availability means 100 minutes of downtime per week. Such downtime still incurs a significant penalty in many systems [14]. A real-time application that cannot tolerate even such small episodes of downtime for maintenance purposes (or any other purposes) is discussed in Chapter 7.

A partial solution to the effect of software maintenance on the availability of software is redundancy as a means of fault tolerance. In some applications, the need for corrective maintenance can be delayed by using fault tolerant approaches to software development such as recovery blocks [15] and N version programming [16] such that if one version of an application fails, the failure is masked by other independently developed versions of the application thus delaying the need for corrective maintenance, and in doing so keeping the application available for use. Unfortunately, in addition to the fact that running multiple versions of an application is costly, as pointed out earlier in the discussion about software maintenance, the majority of software maintenance is related to modifying the functionality of the software in accordance with user requirements changes rather than on correcting software errors. Such fault tolerant schemes do not solve the issue of having to take the system offline to modify the architecture of the software application to accommodate new functionality. Hence fault tolerance is not discussed further in this thesis.

As redundancy is only partially of use for corrective maintenance but not of use for perfective and adaptive maintenance, a more general solution is required for the problem of performing software maintenance without making the software unavailable for use. Hence this thesis focuses on mechanisms to support software

evolution/maintenance in general rather than on fault tolerant approaches to masking failures. One solution is to develop and deploy software applications using Service-Oriented Architecture (SOA) and Component-Based Software Engineering (CBSE), the benefits of which are dynamicity and modularity respectively and are discussed further in Chapter 2. The OSGi Framework [17] (discussed further in Chapter 3) is a Java based run-time environment for deploying component-based and service-oriented Java applications.

The significance of the OSGi Framework for solving the conflict between maintaining a high level of software availability and the need for maintenance/evolution of the application is that OSGi allows for dynamic reconfiguration. Dynamic reconfiguration allows the functionality of the application to be modified during run-time thus keeping it available for use. It is possible to add new functionality to the software application, update existing functionality of the application, and remove functionality from the application, thus allowing for all types of software maintenance to occur without having to make the application unavailable for use as is traditionally required.

1.3 Java and the Real-Time Specification for java (RTSJ)

Whilst the dynamic reconfigurability offered by OSGi enables non real-time software applications to undergo maintenance/evolution and remain available for use, it is not possible to use OSGi to do the same for real-time systems.

As discussed, the OSGi Framework is based on standard Java as are target applications. However, Java is unsuitable for the development of real-time systems for a number of reasons related to: memory management, time values and clocks, scheduling, asynchrony, and resource sharing.

Several attempts have been made to extend the Java language to be more appropriate for the development of real-time systems by addressing the above issues. The most successful attempt to define a real-time version of Java has been the Real-Time Specification for Java (RTSJ) [18] produced by the Real-Time for

Java Expert Group (RTEG). The RTSJ defines a set of extensions to the Java virtual machine and the class libraries that facilitate real-time programming by enabling the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints (also known as real-time threads).

The aforementioned issues with standard Java in the context of real-time systems are discussed below along with the enhancements to standard Java provided by the RTSJ.

1.3.1 Memory Management

The run-time implementations of most programming languages provide a large amount of memory (called the heap) so that the programmer can make dynamic requests for memory allocation (for example, to contain an array whose bounds are not known at compile time).

In standard Java, all objects are allocated on the heap. The heap is managed by a garbage collector, which reclaims areas of the heap that are no longer being referenced [19]. Such automatic memory management benefits application developers by providing pointer safety, memory leak avoidance, and generally leaving the developer free to concentrate on writing the application logic rather than being concerned with writing memory management routines.

Historically, garbage collections are performed while the application program is halted; a process termed Stop-The-World (STW) [20]. With a STW garbage collector, the application program experiences a garbage collection as a pause in program operation. These STW pauses are unbounded in length and are typically quite intrusive, ranging from hundreds of milliseconds to several seconds. The length of a pause depends on the heap size, the amount of live data in the heap, and how aggressively the collector tries to reclaim free memory. The garbage collector thus may interrupt a time-critical thread, which may have a significant

impact on the response time of the thread possibly causing it to miss a deadline [21].

To avoid the overhead and possible unpredictability of the garbage collector of standard Java, in addition to the garbage collected heap of standard Java, the RTSJ provides other memory areas including a region-based approach to memory management called scoped memory (SM) [22]. A scoped memory area is a region of memory that has a reference count associated with it which keeps track of how many real-time entities are currently using the area. When the reference count goes to zero, all of the memory associated with objects allocated in the memory region is reclaimed.

As a result of the advances in research on real-time garbage collection coupled with the fact that the scoped memory model is often criticised for being complex to use, many real-time JVMs provide real-time garbage collectors even though no such collectors are specified by the RTSJ.

The RTSJ also introduces two features which are particularly useful with regard to memory in embedded systems. Firstly, the RTSJ introduces Physical Memory Areas which allow objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access. Secondly, the RTSJ provides a mechanism for programmers to access raw memory locations that are being used to interface to the outside world; for example memory-mapped input and output device registers. This ensures that real-tine Java applications can for example interact with embedded systems written in C or C++ [23].

1.3.2 Time Values and Clocks

Time is the focal point of real-time systems. Unfortunately Java is very limited in its support for time values and clocks. Java has no support for absolute time, for example many systems require threads to sleep until an absolute time; however Java only allows a relative time to be specified. Also, Java supports the notion of

a wall clock (calendar time), with a class intended to reflect UTC (Coordinated Universal Time). However, the accuracy of the wall clock depends on the host system, and therefore standard Java may not provide fine enough time granularity for a real-time application.

The RTSJ overcomes these problems of standard Java by providing a means of expressing both absolute and relative time with sub-millisecond precision. This allows RTSJ implementations to provide time-based services, such as timers, using whatever precision it is capable of. There is also a real-time clock that advances monotonically. This monotonically advancing clock avoids the issues of leap ticks.

1.3.3 Scheduling

Scheduling of threads is a key aspect for all real-time systems. Although Java permits the priority of a thread to be specified, the JVM is not required to enforce such priorities and therefore offers no guarantees that the highest priority runnable thread will be the one executing at any point in time. The reason for this is that Java is platform independent and therefore it is not possible for the language to be able to make assumption about the concurrency model used by the underlying OS and JVM implementation. For example, threads may be invisible to the OS and instead scheduled by the JVM (as with Green Threads), or instead threads may map directly to native OS threads. In the case of native threads, the OS may not support pre-emptive priority-based scheduling. As a result of the poor support for scheduling threads, Java programs lack predictability and prevent the use of Java for the development of real-time systems

The RTSJ solves these scheduling issues by providing fixed-priority pre-emptive scheduling with at least 28 unique priority levels that must be respected by the scheduler. In addition, implementations of the RTSJ are able to provide different scheduling algorithms.

In addition to providing fixed priority scheduling, the RTSJ also abstracts away from threads to the notion of schedulable objects to include threads, asynchronous event handlers (discussed in Section 1.3.4) and threads which do not use the heap for memory allocation (no-heap threads[1]). In terms of the feature of schedulable objects, the following parameters can be set by the application developer

- Scheduling parameters, for example, the priority at which it should be scheduled.
- Release parameters, which define how a task is activated such as aperodic, sporadic, or periodic.
- Processing group parameters, allows a collection of schedulable objects to be grouped so as to ensure that collectively, the processor demand does not exceed a particular value.
- Memory parameters, for the purposes of pacing the garbage collector to satisfy all of the thread allocation rates

1.3.4 Asynchrony

In terms of event handling, standard Java classes can be programmed that multiplex events onto a single thread that handles them in a particular order. For example, the Abstract Windows Toolkit has an event-handling thread to respond to events caused by user interface components. However, from a real-time perspective, events may require their handlers to respond within deadlines. Hence, more control is needed over the order in which events are handled.

With regard to interrupting a thread, the interrupt mechanism of standard Java attempts to provide a limited form of asynchronous notification by setting a pollable flag in the target thread, and by throwing a synchronous exception when the target thread is blocked at an invocation of `wait()`, `sleep()`, or `join()`. However, given that there is no guarantee that the target thread will poll for notification in a timely manner, this facility is more of a synchronous notification

[1] By not allocating on the heap, such threads are able to in execute in preference to the garbage collector and can thus be typically provided with stronger timing guarantees than heap using threads.

method. It is not adequate for asynchronous notification in real-time systems. As a result, standard Java provides no predictable and safe method of transferring control of, or terminating a thread. Such mechanisms are almost certainly required in real-time systems designed to monitor and control environments.

To solve the above issues, the RTSJ provides a means of both asynchronous event handling and asynchronous transfer of control (ATC) [24], which includes thread termination.

Asynchronous event handling is performed with the use of the asynchronous event handling subclass of schedulable object (briefly mentioned in Section 1.3.3). As mentioned, the handlers scheduling and release characteristics can be specified and the handler can then be added to asynchronous events such that when events are fired, any attached handlers are released with whatever real-time parameters they were configured with. In terms of scheduling, each handler may be mapped to a separate thread if necessary, otherwise, many handlers will map to a single thread.

In ATC, the RTSJ extends the effect of the standard Java interrupt mechanism by offering a more comprehensive and non-polling asynchronous execution control facility. It is based on throwing and propagating exceptions that, though asynchronous, are deferred where necessary (such as synchronized methods) in order to avoid data structure corruption. As a note, ATC can be used to provide a safe means of terminating a thread.

1.3.5 Resource Sharing

Java provides a synchronization mechanism that is based on mutually exclusive access to shared data via a monitor-like construct. Unfortunately, all synchronization mechanisms that are based on mutual exclusion suffer from priority inversion i.e. they may cause a high priority thread to become blocked by a thread with a lower priority.

Priority inversion is a well known problem, for which there are many solutions such as the Stack Resource Policy [25], the Priority Ceiling Protocol [26], and the Priority Inheritance Protocol [26]. Standard Java does not support any of these mechanisms. Therefore, the RTSJ extends the semantics of Java synchronization to mandate priority inversion control, typically by using the priority inheritance protocol. This allows the priority inversion to be bounded.

However, there is still a problem if schedulable objects want to communicate with non-real-time threads. If the actions of the non-real-time thread result in garbage collection, the schedulable object will then pre-empt the garbage collector, but is unable to enter the mutual exclusion zone. It must then wait for the garbage collection to finish and the non-real-time thread to leave the zone.

To solve the above problem, the RTSJ provides wait-free non-blocking classes to help facilitate this communication: The wait-free queue classes facilitate communication and synchronization between instances of real-time and non-real-time threads.

1.4 Thesis Motivation and Thesis Hypothesis

Even if OSGi applications are written in the RTSJ as opposed to standard Java, as will become apparent throughout this thesis, the applications are still unable to be provided with real-time guarantees i.e. the integration of the RTSJ with the OSGi Framework is not enough to be able to develop dynamically reconfigurable real-time systems. As a result, it is not currently possible to undertake software maintenance of real-time systems without taking them offline and making them unavailable for use. As discussed in this chapter, while the real-time application is offline it has no utility and may incur penalties as a result.

Clearly, allowing real-time systems to remain available for use during maintenance/evolution will increase their utility and will avoid the possible implications such as financial losses associated with taking the system offline. This is the motivation for the work carried out in this thesis. The thesis

hypothesis is therefore as follows: The OSGi Framework has proven ideal in developing dynamically reconfigurable Java applications based on the principles of component-based software engineering and service-oriented architecture. With dynamic reconfiguration, software applications continue to remain available and have utility even while they are undergoing maintenance/evolution. One domain where OSGi has yet to make an impact is real-time systems. By integrating the OSGi Framework with the RTSJ, and by providing certain extensions to the OSGi Framework, OSGi can be used to develop real-time systems which are dynamically reconfigurable. This means that application maintenance/evolution can take place without taking the system offline and without affecting the application's real-time constraints. Such dynamic reconfiguration of real-time systems will allow them to remain available and have utility during software maintenance and evolution activities.

1.5 Thesis Contributions

As stated in the thesis hypothesis, in order for real-time systems to be dynamically reconfigurable with OSGi, the OSGi Framework requires extensions in order to enable it to support real-time system development and deployment. These extensions and the resulting real-time version of the OSGi Framework (or RT-OSGi) are the contributions of this thesis. More specifically, the contributions are the following:

1) Identification of the challenges of providing dynamic reconfiguration in the context of real-tie systems, and in particular, the issues preventing the standard OSGi Framework from being used to develop and deploy dynamically reconfigurable real-time systems (Chapter 3)
2) The design of a real-time version of the OSGi Framework (RT-OSGi) which solves the dynamic reconfiguration issues in the contest of real-time systems. The RT-OSGi design consists of the following extensions to the standard OSGi Framework:
    a. Application-level cost enforcement and temporal isolation (Chapter 4)

b. WCET contracts and automated stub service implementation generation for OSGi services and service factories (Chapter 4)
c. Bounded-time component activation and deactivation (Chapter 4)
d. RTSJ Asynchronous event handling model integration (Chapter 4)
e. CPU admission control (Chapter 5)
    i. Server parameter selection
    ii. Schedulability analysis – Response-Time Upper Bound and Boolean Response-Time Analysis
    iii. Hierarchical scheduling simulation – component priority range assignment
f. Asynchronous thread termination (ATT) (Chapter 5)
g. Time-based GC simulation (Chapter 6)
h. GC reconfiguration analysis (Chapter 6)
    i. GC work estimation
    ii. GC parameter calculation
    iii. GC cycle length estimation
    iv. Free memory requirement estimation
i. Memory admission control (Chapter 6)
j. Memory allocation enforcement (Chapter 6)
k. Mode change protocol (Chapter 7)

3) The design of a case study which has real-time, reconfiguration/ evolution/ maintenance, and high availability requirements. This case study both motivates the need for, and demonstrates the expressive power of, RT-OSGi (Chapter 7).
4) The implementation of a prototype of RT-OSGi and an evaluation of both the prototype and approach of RT-OSGi (Chapter 8)

1.6 Thesis Structure

The thesis is organised as follows: Chapter 2 discusses the underlying concepts of the OSGi Framework, namely Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA). The application of these principles to real-time systems is also discussed. In Chapter 3, the OSGi

Framework itself is discussed in detail. Among other things, the history of the OSGi Framework, its architecture, and its uses in various domains is discussed.

Chapter 4 is the first of three chapters discussing the extensions to the OSGi Framework which will enable its use in developing dynamically reconfigurable real-time Java systems. In Chapter 4, both temporal isolation and worst case execution-time (WCET) calculation are discussed. In Chapter 5, CPU admission control is discussed. This is required to control the load on the CPU and to only allow dynamic reconfiguration when it does not interfere with the timing requirements of deployed threads. The last chapter to discuss the real-time extensions to OSGi is Chapter 6, which discusses the reconfiguration of the garbage collector necessary to keep it in pace with the changing memory allocation associated with the dynamic reconfiguration of OSGi applications. Memory admission control is also discussed in this chapter. Chapter 7 introduces a case study which serves to give motivation for the use of RT-OSGi i.e. it motivates the need for dynamic reconfiguration in real-time systems, that is, the need for allowing the application to remain available for use despite undergoing maintenance/evolution. The case study also serves as a means of evaluating the expressive power of RT-OSGi along with the prototype implementation of RT-OSGi, which s discussed in Chapter 8. Finally, Chapter 9 concludes the thesis and discusses future work on RT-OSGi.

2

Underlying Concepts of the OSGi Framework

In order to meet the goals of this thesis (i.e. to develop dynamically reconfigurable real-time systems which have high availability requirements), a real-time version of the OSGi Framework has been proposed. Before discussing both the OSGi Framework and the extensions necessary to provide a real-time version of that Framework (RT-OSGi), it is necessary to discuss the underlying concepts of OSGi i.e. Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA). These concepts along with their application to real-time systems development are discussed in this chapter. In addition, other non-OSGi approaches to providing dynamic reconfiguration are discussed with the aim of showing the inadequacies of current research works in meeting the goals of this thesis.

2.1 Component-Based Software Engineering (CBSE)

Component-Based Software Engineering (CBSE) [27] is an approach to software development, developing applications as an integration of software components. The term software component is typically thought of in the ontological sense as being any software entity that can be composed into a composite. Such components are often identified as the result of problem decomposition, which is a common problem solving technique in software engineering. However, in CBSE, a number of criteria have been formulated to distinguish components in CBSE from what is commonly thought of as a software component. Although

this criterion is not universally agreed in the CBSE research community, it is widely accepted.

Generally, a software component in the context of CBSE is defined as: a unit of composition with contractually specified interfaces and explicit context dependencies only, can be deployed independently, and is subject to composition by 3rd parties [27].

The above criteria for what constitutes a component in CBSE is important for ensuring that components are reusable, the key motivation for CBSE. The importance of only using explicit dependencies is so that components in CBSE can easily be composed into many different applications thus maximising the reusability of components. Without such explicit dependencies, the user of third party components would require a sound knowledge of the internal structure of a component, which is unlikely. Using contractually specified interfaces is important so that the user of a third party component is given guarantees about the behaviour of the component and can be assured of a minimum quality of service before purchasing a third party component. Independent deployment of components is also important for reusability because the user of third party components should not be burdened with the task of identifying the components named in the component's explicit dependences. Rather, a run-time environment (a component framework, discussed later in this section) should allow components to be deployed in isolation and resolve component dependencies on behalf of the application developer.

In addition to the above, a further distinguishing feature of CBSE components from other definitions of components is that CBSE components should be conformant with a component model [28]. Component models [29] specify the design rules that must be obeyed by components. These design rules improve composability by removing a variety of sources of interface mismatch (i.e., mismatched assumptions such as communication or encoding techniques). More specifically, component models specify how components interact with each other (a standardized calling convention between components), and therefore express

global or architectural design constraints. This prevents an untidy composition of product-specific interaction schemes.

To support and enforce a component model, a component framework [29] (such as Component Object Model (COM) [30], CORBA Component Model (CCM) [31] and Enterprise JavaBeans (EJB) [32]) provides a variety of runtime services. It is essentially an infrastructure that manages resources for components and supports component interactions. One of the ways it supports component interactions is by introducing the concept of a "requires" interface in addition to the typical "provides" interface.

The "requires" interface specifies what services must be provided by other components in the system i.e. a specification of a components dependencies on other components [29]. As "requires" interfaces cannot be specified through current programming language, "requires" interfaces are specified declaratively through component framework dependent mechanisms. The component framework then matches up the "requires" interface of the newly deployed component with "provides" interfaces of other components. It is this run-time matching mechanism provided by the component framework which enables components in CBSE to be independently deployed.

2.1.1 Motivation for CBSE

Component models specify the necessary standards to ensure that independently developed components can be deployed into a common environment. The use of both component models and component frameworks enable the development of a CBSE component market, where it becomes possible to buy and sell components. This allows for software reuse. Such availability of components may drastically reduce the time and thus cost it takes to design, develop and deploy applications. The reason for this reduction in software development time is not only because of the reuse of third party components but also because the component framework implements key architectural elements of an application.

The use of reusable components that conform to a component model also gives structure to system design and development, which makes system verification and maintenance more tractable [33]. More specifically, such modularity eases both program comprehension and impact analysis of software maintenance and evolution. For example, in terms of impact analysis of changes to a component's "provides" interface in CBSE, all that is required is to trace affected "requires" interfaces. Since such interfaces are defined external to application code, it does not require any knowledge of the implementation of components

2.1.2 Limitations of CBSE

Whilst software reuse is very advantageous and therefore there is a strong motivation for using CBSE, in practice it is quite difficult to achieve software reusability. The reason for this is because currently the component models and component frameworks do not provide sufficient support for reasoning about the behaviour of component-based applications..

Crnkovic [34] makes a useful distinction between component integration and component composition. Crnkovic believes that what is commonly referred to in the literature as "composition", is "integration", and that composition is a stronger notion, which is currently not achievable in the current component frameworks.

Crnkovic defines component integration as the mechanical task of "wiring" components together by matching the needs and services of one component with the services and needs of others [35]. Integration can only detect problems preventing two components from being plugged together, these problems are known as architectural mismatches [36]. Component integration is based on syntactic interfaces, which are the current standard used in programming languages, component frameworks and middleware etc. Syntactic interfaces are limited to specifying only syntactic properties i.e. functionality in the form of method signatures. They do not capture key semantic information regarding substitutability of components and therefore defeat their intended purpose of

information hiding as an enabler to implementation substitutability [37]. In terms of component composition, syntactic interfaces have a lack of concern for the architecture of assembly and effects on behaviour of individual components. Furthermore, they do not provide enough information to predetermine the consequences of using two components together and are therefore unable to guarantee that integrated components will behave as expected. This is termed behavioural mismatch [34].

As an example of the behavioural mismatch that can occur in current component frameworks, consider a component-based software application for controlling an audio system. In the system an audio amplifier component can be integrated with a component for controlling the speakers via syntactically matching the "requires" interface of the speakers with the "provider" interface of the amplifier. However, such syntactic matching will not prevent the case of connecting a powerful amplifier with incompatible low wattage speakers. In such a case, the speakers will plug in with no problem and operating the amplifier at a low volume will almost certainly allow the speakers to function correctly, but if the volume is raised, the speakers will most likely be damaged [34].

In order to solve the behavioural mismatch issue in the audio system example, a more powerful means of component interface specification is required than the common syntactic level of interface specification. Behavioural contracts [38] such as assertions and pre and post conditions can be used to ease the behavioural mismatch problem by providing semantic information about a component's operations. Examples of technologies which support behavioural contracts include certain (dated) extensions to Java such as iContract [39] or JContractor [40], the Object Constraint Language (OCL) [41], and the Eiffel language [42].

Using behavioural contracts in the audio example would solve the behavioural mismatch problem by, for example, using pre conditions to guarantee correct behaviour only if the speakers are equal to or greater than a particular wattage. In addition, the amplifier component developer can include code to perform checks on the wattage of speakers and (say) constrain the volume range based on

the wattage of the attached speakers. Furthermore, rely/guarantee conditions [43] can be used to aid in reasoning about the correctness of such component compositionality.

Although having component frameworks use behavioural contracts (rather than purely syntactical contracts) for components solves some behavioural mismatch scenarios, it cannot solve others. One example of behavioural mismatch which cannot be solved entirely by using behavioural contracts is the issue of emergent properties [44]. Emergent properties are properties of the application that are not explicitly part of the application components themselves, but come into existence by the interactions among the components. Such properties may not be harmful, although it is still desirable to have the ability to determine emergent properties so that their effect on the application can be assessed. In order to solve such cases of behavioural mismatch, it is necessary for component frameworks to support reasoning about emergent system properties [45]. Unfortunately, current component frameworks do not provide such support. Currently, developers of component-based applications rely on some form of integration testing and/or fault injection [34] in order to have a high level of confidence about the correctness of the application integration. However, this is not an adequate solution.

Another issue which makes component composition difficult to achieve is related to the extra-functional characteristics of components. As syntactic interfaces and behavioural contracts cannot describe extra-functional properties then it also means that they cannot hide them therefore allowing such properties to bleed through the interface [29]. Extra-functional properties may therefore in turn become sources of implicit component dependency. As an example, consider a client wishing to use a sort algorithm. It is assumed that the sort algorithm has a sort interface with a `int[] sort(int[] nums)` method. In theory, any component which implements this interface and abides by the same behavioural contract should be substitutable. However, this is not necessarily the case. For example, assume an implementation of the sort interface has a time complexity of O(log n). A client wishing to sort a very large array of integers will become dependent on this performance extra functional property and another

implementation with a time complexity of $O(n^2)$ will simply not suffice. In order to solve this issue, in addition to syntactic and behavioural specifications of functionality, Beugnard [46] also discusses the need for Quality of Service (QoS) interface/contracts. QoS contracts are the highest level of component specification. They allow the definition of extra-functional attributes of software such as reusability, maintainability, reliability, and usability. NoFun [47] provides the ability to specify such QoS attributes of software components.

In summary, component frameworks require true component composition. Component composition extends component integration through the support for reasoning about properties of components assemblies by having the ability to infer application properties based on the properties of components and the relationship between them The purpose of this reasoning is to check the run-time compatibility of components and so prevent behavioural mismatch. The resulting component assembly may then also be used as part of a larger system and thus it must be possible to reason about how the assembly will affect the application [34].

2.2 Service-Orientation

A general definition of a service is [48] "an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production". This definition is not very useful in the realm of computing as components objects, and methods (functions) etc would be considered as services. Brown [49] attempts to refine this definition by discussing some of the properties that a service must exhibit: "A service is generally implemented as a course-grained, discoverable software entity that exists as a single instance and interacts with applications and other services through a loosely coupled (often asynchronous), message-based communication model". Essentially, the key principles of service-orientation include [50]:

- Dynamic availability – services may appear and disappear during run-time.

- Services are dynamically discoverable – services allow their descriptions to be discovered and allow the discovery of different implementations of the same service description.

- Coarse grain services – as services are often distributed; communication over networks incurs significant overheads. Therefore using coarse grain service descriptions to minimise the number of messages sent across the network will improve performance.

- Services are loosely coupled – services must be designed to interact without the need for tight, cross-service dependencies.

- Service contract –in order for services to interact, they need not share anything but a contract (service description) that describes each service and defines the terms of information exchange. In order to use a service, it is only necessary to be aware of the service, that is, to have the service description. The manner in which services use service descriptions results in a loosely coupled relationship, where service providers may be substituted with one another as long as they obey the same contract. The service provider may be substituted with another service provider as long as they obey the same contract.

- Services are composable – collections of services can be coordinated and assembled to form composite services. This allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers. As a note, service composition is considered the responsibility of service requesters, who combine services to address their changing business priorities.

- Services abstract underlying logic – the only part of a service that is visible to the outside world is what is exposed via the service contract. Underlying logic is invisible and irrelevant to service requesters.

- Asynchronous communication – it is beneficial to make services asynchronous in nature. As services are often distributed and will therefore experience network latency, it is important to reduce the time a requestor spends waiting for responses. By making a service call asynchronous, with a separate return message, it allows the requestor to continue execution while the provider has a chance to respond.

Service-Oriented Architecture (SOA) is a way of designing a software system to provide services to either end-user applications or other services through published and discoverable interfaces [49]. In SOA, application assembly is based on service descriptions; actual service providers are discovered and integrated into the application later (late binding), usually prior to or during application execution

As a result of the dynamic availability and dynamic discoverability of services, the architecture of a SOA-based application is dynamic. This differs from the architecture of traditional non-service based architectures, which tend to be static. This flexibility of SOA allows the system to easily evolve with the addition of new functionality through new services. This concept envisages a demand-led software market in which businesses dynamically compose services when needed, to address a particular requirement. SOA thus separates possession of software from use [48].

2.2.1 Motivation for Service-Orientation

Having explained service-orientation, it is clear that services are very much like components, both components and services promote the idea of constructing applications from the assembly of reusable building blocks. Like CBSE, the motivation for service-orientation is software reuse and therefore potentially

reduced software development costs and reduced time-to-market. SOA further enhances these advantages of CBSE [50]:

- Service-oriented software allows organisations to create new software applications dynamically to meet rapidly changing business needs. Such an increase in organisational agility will significantly reduce the cost and effort required to respond and adapt to business or technology related changes.

- Service-orientation solves software evolution issues. The process of discovering services, performing negotiation to acquire them, composing, binding, executing, and unbinding them, alleviates evolution problems. The reason for this is that there would be no system to maintain[2]—it would be created from a composition of services to meet particular requirements at a given time [51].

- Easy to make changes/extend software, for example, in order to produce an annual report in Russian, a service can be acquired, invoked and then discarded after use. This saves having to implement infrequently required functionality from scratch, or having the high cost of buying in components. [51].

- Cost and effort of cross application integration is lowered when the applications being integrated are SOA-compliant (due to loose coupling inherent in SOA).

2.2.2 Limitations of Service-Orientation

Some of the issues involved in service-orientation are:

[2] This assumes that all of the functionality required by the application is available as third party services, which is unlikely. It will almost certainly be necessary to maintain/evolve some non-service software.

1. Software understanding – many barriers to the successful understanding of service-oriented software arise from its distributed and dynamic nature. It becomes difficult to understand software when a SOA crosses organisational boundaries. Naturally, services are likely to depend on other services, which may in turn depend on other services. However, this chain must not get too long as this will make understanding the software very difficult, and will reduce performance.

2. Trust – obtaining trust in a particular service or supplier is difficult. As services are black boxes, service requesters do not know whether a service is malicious or not. Conversely, service requesters may be malicious and cause Denial-of-Service attacks on the service providers, thus trust is an issue for both parties.

   In addition to the issue of trust in terms of malicious service providers and requesters, there is also the issue of trust regarding confidentiality of data. Service requesters may need to pass on confidential data to services providers who are virtually unknown to the requesters. Many companies will be reluctant to do this. Of course, the issue of trusting code in the form of black boxes however is not unique to service-orientation, but rather is common where business functions are contracted out to other organisations

3. Performance – service-orientation introduces layers of data processing, each layer imposes a performance overhead. In addition, some service-oriented technologies such as Web services using SOAP (Simple Object Access Protocol) depend on XML for data representation. In contrast to binary data transport mechanism, XML has a lower performance and higher usage of network and internet traffic. Therefore SOAP performance is degraded due to the time involved in extracting the SOAP envelope from the SOAP packet, and from parsing the contained XML information in the SOAP envelope.

4. Dynamic availability and dynamic discovery – the dynamism of SOA impacts the predictability of a software application. As predictability is of utmost importance in the real-time systems domain, standard SOA is unsuitable for use in this domain. This is discussed further in Section 2.4.2.

2.3 Comparison of Service-Orientation and CBSE

Differences between SOA and CBSE are [52];

- Ownership – in CBSE, the user pays for a license to deploy a component, that is, either the binary file or source file is actually delivered to the user. With service-orientation, the user doesn't receive a copy of a service. Instead the user pays to consume a service, and the service is invoked via the Internet for example. In this way, SOA changes the focus from product delivery (as with CBSE) to service-based delivery.

- Assembly time – leading on from ownership, as components are always available to the user in CBSE, applications are almost always assembled from components at deployment time rather than at design or run-time.

  The reason for this is because no single component developer can predict which other components a third party wishes to integrate their component with. Component developers therefore use "requires" interfaces to declaratively state a component's dependencies. As discussed, the dependencies between components can then be resolved at deployment time by the component framework. Only once all of the components dependencies are resolved can a component be executed. Once the components dependencies are resolved and have thus been assembled, there is no possibility of discovering and integrating new components.

  In contrast, in SOA, as only service descriptions are available to a user during assembly, the actual service providers are dynamically discovered and integrated into the application prior to or during execution time. What

this means is that at compile-time, services are integrated into an application through their interfaces. At run-time, the service requester must query a service registry to find an actual service provider.

- Dynamic Vs Static Architecture – In SOA, as the assembly of services takes place at run-time, it is unknown a priori which services will be available. As a result SOA applications have a dynamic architecture. As assembly in CBSE takes place at deployment-time, CBSE is targeted more towards construction of static applications where the architecture cannot be modified during run-time.

- Execution environment – in CBSE, a lot of responsibility is given to the execution environment (component framework) for example for solving low level deployment issues. In contrast, service-orientation does not consider low-level activities such as deployment. In service-orientation, an execution environment provides two main mechanisms to service providers and requesters that support the service-oriented interaction pattern, service registry access, and a notification mechanism to signal service changes. The notification mechanism is used by service requesters to track the availability of the services they require.

- Communication protocols – components tend to use proprietary communications protocols such as CORBA's Internet Inter-Orb Protocol (IIOP). This means that components designed for different component frameworks cannot interoperate. With service-orientation on the other hand, service description is decoupled from the communication protocol, therefore it is possible for services developed using different technologies to communicate.

As the list above shows, there are many differences between service-orientation and CBSE. Despite these differences, the two concepts are complimentary and can be used together. One method of using CBSE and service-orientation together is to introduce service-oriented concepts into a component model. In

particular, the service-oriented interaction pattern could be used as a means to connect components, which act as service providers and service requesters. The benefit of this approach is that it introduces support for late binding and dynamic component availability to component models [52] thus adding dynamism to the otherwise static software architecture of CBSE applications. Such a combination of SOA and CBSE is used in the OSGi component framework, which is the subject of the next chapter.

2.4 Application to Real-Time Systems

Having discussed CBSE and SOA, it is necessary to briefly discuss their application to real-time system development in order to demonstrate that neither of these technologies alone can solve the issues described in this thesis. Research works relating to the OSGi Framework and real-time systems are discussed in Chapter 3.

2.4.1 Real-Time CBSE

As discussed, there are many reasons why CBSE is beneficial for use in the software development process. However, existing wide-spread component technologies are inherently heavyweight and complex, incurring significant overheads on the run-time platform. Such technologies also do not in general address timeliness, quality-of-service or similar non-functional properties important for embedded and real-time systems. There are a number of component models which have attempted to address the above issues, these component models can be categorised as those designed for use with the RTSJ, and those designed without the RTSJ in mind.

2.4.1.1 Non-RTSJ CBSE

One particularly interesting component model for static systems is UM-RTCOM [53]. UM-RTCOM is a component model implemented in RT-CORBA [54]. The most interesting aspect of this model is that it defines semantics for component interaction. This allows reasoning about interaction properties such as deadlock,

and also acts as input for offline schedulability analysis. However, UM-RTCOM offers no support for dynamic reconfiguration. If components were to change their dependencies at run-time (as would likely happen during dynamic reconfiguration), the system properties and schedulability results would no longer be valid.

Other real-time component models include: AutoComp [55], SAVEComp Component Model (SaveCCM) [56], Pin [57], PECOS (Pervasive Component Systems) [58], and The Robocop Component Model [59]. Unfortunately these component models are also only suitable for developing real-time systems with static configurations i.e. dynamic reconfiguration is not supported. Such a lack of support for dynamic reconfiguration in these component models is due in part to the fact that these component models typically provide real-time functionality by using compile-time mappings to a real-time operating system. Using this approach, an application is designed using a component model; this includes specifying the application real-time constraints. The compiler then incorporates a transition from the component based design to a real-time model. During this step the components are replaced by real-time tasks.

2.4.1.2 CBSE and the RTSJ

In [60], Etienne introduces a component framework which facilitates the development of component-based RTSJ applications. The authors' component model introduces the notion of components, connectors, composites and various contracts such as syntactic, behavioural and temporal contracts to the RTSJ. Similarly, Hu et al [61] introduce a component framework (Compadres) which the authors claim abstracts away the RTSJ memory management complexity by providing a Compadres compiler to automatically generate the scoped memory architecture for components. Thus this further simplifies the software development process and adds to the aforementioned advantages of using CBSE. Finally, in [62], Psek et al introduce a component framework which allows real-time and non real-time components to communicate, which is a limitation of the work presented by Hu et al.

Despite the fairly large number of research works relating to both RTSJ and non-RTSJ based real-time component frameworks, none of these component frameworks address the issue of dynamic reconfiguration. However, the work of Tatibana et al [63], which is in the context of a distributed component framework, addresses dynamism in the sense of a dynamic work load. In their work, the authors discuss allowing "server" components to adapt to dynamic loads in terms of the numbers of "client" components making remote method invocation requests. More specifically, CPU admission control is discussed as a means of only allowing new method invocation requests if the "server" component can still guarantee the real-time requirements of previously accepted requests for service. However, the dynamism discussed by Tatibana is not in the context of allowing an application to be dynamically reconfigured but rather simply a means of providing a general load management scheme for component-based real-time systems. Therefore, it is not possible to perform software maintenance/ evolution of real-time systems whilst maintaining a high level of software availability.

2.4.2 Real-Time SOA

In Real-Time SOA (RT-SOA), the service providers can offer real-time services i.e. they are able to guarantee that service invocations from service requesters will be executed within a given deadline. However, there are a number of critical issues which must be solved in order to support RT-SOA from both the service provider's and service requester's viewpoint and these were first discussed by Tsai in 2006 [64]. These issues include providing real-time scheduling for service providers and requesters, enabling resource reservation and temporal isolation in order to guarantee sufficient resources for service invocations from service requesters, providing real-time contracts for service discovery, and providing timing guarantees for the transmission of messages containing the service calls and results across the network.

Since Tsai first discussed the challenges which must be overcome in order to provide RT-SOA, there have been a number of research works related to RT-

SOA, with each research work attempting to solve one or more of these challenges. In any case, RT-SOA is still an emerging field of research and issues remain.

Perhaps the most significant research that has been carried out into RT-SOA is by Panahi et al [65]. The authors discuss the need to enhance SOA in order to be adopted in the real-time systems domain. Furthermore they introduce a framework for RT-SOA (RT-Llama), which, amongst other things, provides global resource management. This allows resource reservation to be made by service requesters in advance of service invocations such that service providers are able to provide real-time services i.e. service providers are able to guarantee that service invocations made by service requesters are completed within a given deadline regardless of the changing number of service requesters making invocations. In addition, the authors also provide CPU bandwidth management to ensure that service requests are temporally isolated. The combination of such CPU reservation and temporal isolation of service invocations provides predictability to service requesters and enables them to guarantee end-to-end deadlines.

Complementing the work by Panahi et al, Ayres et al [66, 67] discuss some of the issues involved in providing RT-SOA. In particular, the authors discuss the real-time issuers involved in the communication between the distributed service requesters and service providers. The authors provide a solution by introducing an architecture based on the Flexible Time Triggered communication paradigm (FTT).

However, the main focus of RT-SOA is on guaranteeing service requesters' timing constraints and not on providing applications with dynamic reconfigurability suitable for performing general real-time software maintenance/evolution online. In terms of the support for dynamic reconfiguration, RT- SOA enables service requesters to make use of different implementations of service interfaces that it compiled to and therefore enables a limited form of online corrective maintenance to take place. Furthermore, RT-SOA enables dynamic reconfiguration of service providers by allowing the

service provider to modify the set of services it deploys. However, this dynamic reconfigurability is quite primitive because the application as a whole cannot be dynamically reconfigured. For example, even if the set of services available to an application is dynamically changed, service requester cannot be dynamically reconfigured to make use of such changes. In addition, the application's set of threads cannot be modified and the non-service using parts of an application are unable to benefit from any dynamic reconfiguration. In order to perform such software maintenance/evolution in RT-SOA, it is necessary to take the application offline.

Figure 2.1 illustrates the dynamic reconfiguration issues of RT-SOA. In Figure 2.1, the service requester is able to exhibit a limited form of dynamic reconfiguration by binding with an alternative service implementation of Service A after the implementation it was using became unavailable. However, the service requester is unable to undergo dynamic reconfiguration to add additional threads and utilise newly deployed services such as Service B. The required service requester dynamic reconfiguration is depicted with dashed lines.

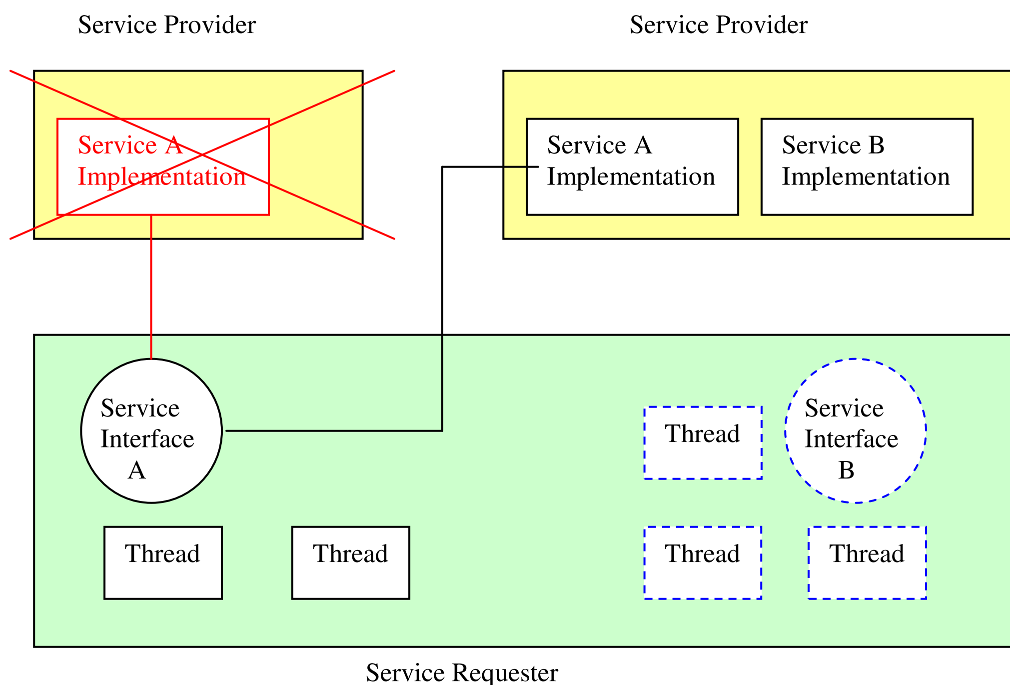

Figure 2.1 Limited Dynamic Reconfigurability of RT-SOA

Finally, in addition to the dynamic reconfigurability limitation inherent in the RT-SOA model, none of the aforementioned RT-SOA research works address all of the issues raised (and solved) in this thesis. Thus none of the RT-SOA research works are capable of meeting the goals of this thesis.

2.5 Other Related Work on Dynamic Reconfiguration

Although not related to CBSE and SOA, there have been a number of research works dedicated to providing a high level of dynamic reconfigurability to real-time systems. These research works are discussed here for completeness as these works more closely meet the aims of this thesis than the use of RT-CBSE and RT-SOA in isolation.

In [68], Pfefer discusses the dynamic reconfiguration of a real-time system by exchanging classes within a known, bounded amount of time, without interrupting the execution of the application threads i.e. without causing threads to miss deadlines. The idea is that the application polls to see if an update is available and if so it performs the reconfiguration. While this work closely matches the goals of this thesis in terms of developing dynamically reconfigurable real-time systems, it has a number of limitations. First of all, this work targets a specific JVM (the KOMODO JVM). Secondly, it is unclear how the KOMODO JVM addresses the limitations of using the Java language in the real-time systems domain since it does not implement the RTSJ. Thirdly, none of the features required for supporting dynamically reconfigurable real-time systems such as CPU and memory admission control and memory management are discussed. Finally, it is not clear how the real-time requirements of the application can be guaranteed if new threads are added to the application as part of a dynamic update.

While Pfefer's work focuses on providing the ability to perform unplanned dynamic reconfiguration (or more apparently dynamic updating of Java classes), there are a number of works which provide a mechanism for planned dynamic reconfiguration. In [69], Adler et al discuss the use of dynamic reconfiguration

in automotive systems in order to adapt the system in response to run-time errors caused both by internal system faults and adverse environmental situations. The planned dynamic reconfiguration essentially takes place by attempting to determine (through component composition analysis) at design time the set of possible system and environmental states, and by providing a mapping scheme between the various states and various configurations of the application. In this way the system can respond safely to any anticipated errors.

Rasche [70] discusses the design of a framework (Adaptive .NET) for the dynamic reconfiguration of Microsoft's .NET applications. The framework supports the selection of a particular configuration of objects/components based on measured environmental conditions, with changes in the environment driving dynamic application reconfiguration. Furthermore, in [71], Rasche provides extensions to Adaptive .NET in order to support for adaptive, heterogeneous applications based on not only .NET but also Java and CORBA. Bruneton et al [72] discuss a hierarchical, reflective component model named Fractal along with a Java implementation of that component model called Julia. Unlike the component models discussed in Section 2.4.1.2 but similar to Adaptive .NET, Fractal provides dynamic reconfigurability to applications. However, the work in both Adaptive .NET and the Fractal component model are not in the context of the dynamic reconfiguration of real-time systems and are therefore unable to meet the goals of this thesis.

To enhance the work in [70] and [71], Rasche [73] argues that dynamic reconfiguration is required in the real-time systems domain, specifically, in order to adapt microcontrollers as a result of unstable and ever changing environmental settings. To accommodate this, Rasche provides a dynamic reconfiguration algorithm that executes within a bounded amount of time and allows real-time applications to be adapted to changing environmental conditions whilst meeting all task deadlines during the reconfiguration process. However, like the work by Adler, this dynamic reconfiguration is constrained by the use of pre-defined adaptation policies, which are design-time defined mappings of environmental conditions to corresponding application configurations. As a result, this work suffers from the same problem as that proposed by Adler in that all possible

reconfigurations must be known before deploying the application, and so it is not possible to dynamically reconfigure the application in ways that were not pre-planed. Therefore, the application is not permitted to be evolved in ways unforeseen at design-time, and as a result, it is not generally possible to perform application maintenance without taking the application offline and making it unavailable for use. As can be seen, neither the work by Adler nor the work by Rasche is capable of meeting the goals of this thesis.

Similar to the work by Rasche [73], Brinkschulte [74] discusses an approach to meeting the timing constraints of a real-time application during the dynamic reconfiguration process i.e. providing a predictable and pre-defined blackout time (time for which part of the application is unavailable for use because it is being modified as part of the dynamic reconfiguration).

However, it is not clear whether this work addresses the issue of unplanned dynamic reconfiguration or whether the possible application configurations must be known at design-time in order to provide a predictable blackout time. In any case, although allowing a component/service of the application to continue to provide real-time guarantees even if it is being modified as part of dynamic reconfiguration is a desirable feature of dynamic reconfiguration of real-time systems, Brinkschulte does not discuss other equally important issues involved in guaranteeing that application threads' deadlines continue to be met after dynamic reconfiguration has occurred such as temporal isolation and admission control etc.

Finally, The FRESCOR project [75] was a consortium research project funded by the European Union. The aim of the project was to develop an infrastructure (FRSH) to support the deployment of a dynamic number of real-time applications with flexible scheduling requirements. Each application negotiates a contract with FRSH in order to attain a guarantee regarding a minimum quality of service level for various resources such as CPU-time and memory, regardless of the number of deployed applications. The main contribution of FRESCOR is that the FRSH infrastructure is able to redistribute resources to other deployed applications as the set of applications deployed changes thus, for example, making use of spare capacity. FRESCOR however does not provide an approach

to dynamically reconfigure an application because supporting application maintenance/evolution whilst maintaining a high level of application availability was not one of its design goals.

However, some of the features of the FRSH run-time environment are essential in RT-OSGi in order to support dynamic reconfigurability in the context of real-time systems. Such features are discussed in subsequent chapters of this thesis.

To conclude this chapter, the related works previously discussed fail to meet the goals if this thesis. The reasons for this include:

1. No dynamic reconfiguration (RT-CBSE)
2. Limited forms of dynamic reconfiguration (such as updating classes only, no new thread deployment, and new service deployment only (RT-SOA))
3. Dynamic reconfiguration of non-real-time systems only
4. Planned dynamic reconfiguration only (configurations must be known pre-deployment time)
5. General dynamic reconfiguration of real-time systems but without addressing all of the associated issues (e.g. temporal isolation, reconfigurable GC absent etc)

As a result of the limitations with the existing research works, the contributions and originality of the work carried out in this thesis is evident.

2.6 Summary

Real-Time CBSE (RT-CBSE) and Real-Time SOA (RT-SOA) typically reduce the complexity of real-time systems software development. There has been a number of research works related to RT-CBSE and RT-SOA in recent years, although none of these works provide adequate support for a high level of dynamic reconfiguration. In contrast, there have been a number of research works which are not component or service-based which provide various levels of dynamic reconfigurability to real-time systems. In the case of the research works that provide a high level of dynamic reconfigurability they either lack one or

more features necessary to guarantee timing requirements to the application or they are restricted to having pre-planned application configurations i.e. the application can only guarantee real-time constraints if the dynamic reconfiguration is changing the application to a configuration known before deployment-time. This means that it is not possible to adapt the application in ways unanticipated at design-time. As a result, no known existing research work is able to meet the goals of this thesis.

3

Overview of the OSGi Framework and its Real-Time Extensions (RT-OSGi)

As discussed in Chapter 1, the OSGi Framework provides Java applications with dynamic reconfigurability. However, in order to meet the goals of this thesis, the OSGi Framework requires various real-time extensions so as to provide a real-time version of the OSGi Framework (RT-OSGi) suitable for hosting dynamically reconfigurable real-time Java applications. The focus of this chapter therefore is on giving an in-depth discussion of the OSGi Framework, and giving an overview of RT-OSGi (the contribution of this thesis). Finally, an alternative approach to RT-OSGi presented in the literature is discussed in order to compare the two approaches so as to demonstrate the inadequacies of the alternative approach to RT-OSGi in attempting to meet the goals of this thesis.

3.1 OSGi Alliance and the OSGi Framework

The OSGi Alliance is an open standards organization consisting of a number of member organisations, adopter organisations and supporting organisations. These organisations include IBM, Hitachi, Mitsubishi, Oracle, NTT, Siemens and Red Hat.

The OSGi Alliance created the OSGi Service Platform Specification [17] which delivers an open, common architecture to develop, deploy and manage services

in a coordinated fashion [76]. At the centre point of this platform is the OSGi Framework.

The OSGi Framework is a run-time environment which encompasses the service-oriented concepts of dynamic discovery and dynamic availability along with the concepts of CBSE, namely modularity. More specifically, by combining the principles discussed in Chapter 2 (namely CBSE and SOA), the OSGi Framework essentially acts as a Java-based component framework with an intra-JVM service model. This allows Java applications to be developed as a number of service requesting and service providing components, with service requesters and providers communicating through the OSGi Framework's service registry/directory, as is typical in SOA technologies (see Chapter 8 for a further discussion and diagram of the OSGi framework and the associated environment).. Such Java applications have the property of being highly dynamically reconfigurable. Not only is it possible to utilise the dynamicity of SOA e.g. substituting service implementations, it is also possible to dynamically change the set of components deployed in ways unforeseen at design/deployment time. Such dynamism and substitutability is achievable because each component uses a separate class loader [77, 78]. For a more detailed discussion of class loaders, reflection, and (for example) how dynamic updates can be achieved, see [79].

In OSGi, components are functional units with life cycle operations and class loading capability. A component is a Java Archive (JAR) file, which packages classes (which may or may not be made available as services), meta-data, and resources such as HTML pages and graphics files. As a note, a component is functional in the sense that it can have dynamic reconfiguration operations performed on it. However, it is the code and resources stored within the component that are of direct use to other users of the framework.

The life-cycle operations are what enable the dynamic reconfiguration of application to take place in the OSGi Framework. They are method calls which enable components to be installed, started, updated, stopped, and uninstalled. Life-cycle operations can be performed either by another component in the system or by an OSGi Framework implementation-dependent manner such as via

the command line. Life-cycle operations cause a component to transition from one state to another. Figure 3.1 shows the possible state transitions of a component.

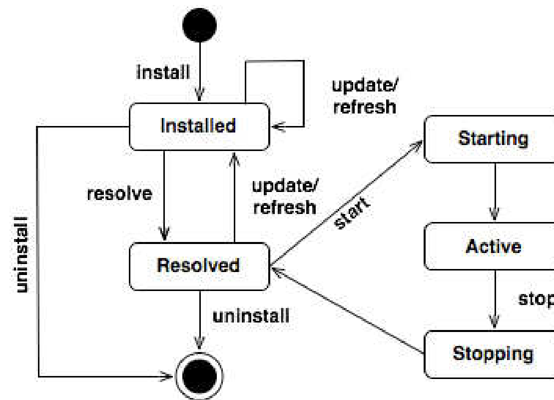

Figure 3.1 OSGi Component State Transitions

Each component state transition is explained below [17]:

- Installing a component – to install a component, the framework must be supplied with the URL of the component's JAR file, the JAR file may be on a Web server over a network, or on a local file system. Once the component is retrieved, the OSGi Framework examines the component's manifest headers, and extracts relevant data such as imports, exports, and the name of the activator class. This results in the component moving into the Installed state.
- Resolving a component – After installing a component, if it needs to import Java packages (i.e. if it has external dependencies), the Framework must resolve these dependencies by checking whether any components have exported those packages. This process typically takes place when the first request to start the component has been made after it has been installed. This resolution process is essentially the matching between the "requires" and "provides" interfaces of components in classic CBSE theory. If the resolution process is successful, the component transitions from the Installed to the Resolved state.
- Starting a component – The Framework creates a BundleContext object for the component. This object enables the component to interact with the

OSGi Framework by acting as an interface for the functionality offered by the Framework. If the component contains an activator, the Framework instantiate it and calls its start method, passing the BundleContext as a parameter. The start method is where the component is able to create threads, publish services, and find and bind with services provided by other components amongst other things. While the activator's start method is being executed, the component transitions to the Starting state, and upon completion, transitions to the Active state.

- Stopping a component – when a component is to be stopped, the Framework calls the component activator's `stop` method; while this method is being executed, the component is briefly in the Stopping state. The Framework automatically performs the following tasks:
  1. Unregisters services provided by the component, event broadcast to notify interested parties
  2. Releases any services in use by the component
  3. Moves component back to the resolved state
  4. Broadcasts an event to notify listeners of component state change to resolved

  The component then transitions to the Resolved state via the Stopping state.
- Uninstalling a component – This method causes the Framework to notify other components that the component is being uninstalled. To whatever extent possible, the Framework must remove any resources related to the component. If the component is in the Active state, it will transition, from the Stopping, Resolved, and finally to the Uninstalled state. If the component is either Installed or Resolved, it transitions directly to the Uninstalled state.
- Updating a component – The update process supports migration from one version of a component to a newer version of the same component. The new component should only make implementation changes, and not make changes to exported interfaces. This minimises disruption to components that depend on the component undergoing the update, as a framework restart is required for exported package modifications to take effect.

The update process is as follows [17]:

- If in the Active state, the component is stopped, moving ultimately to the Installed state. If the component is in the Resolved state, it moves directly to the Installed state. Any services are unregistered. Clients listen for the unregistration event and can therefore take some action before the component is stopped. For example, the synchronous event handling feature of OSGi enables service requester's to complete their invocation of services and then release their reference to the service before the service unregistration process completes.
- The new component is then fetched, and installed. If installation fails, the framework reverts to using the old component. Note that it is the component programmer's responsibility to provide a means of saving and loading component state to prevent application logic errors in such a failed-update scenario and in the update scenario in general. If installation succeeds, an event is broadcast to notify interested listeners that the component has been updated. The component then either remains as Installed, if the component was previously not in the Active state, otherwise, it returns to the Active state.
- As the component starts, no new packages are exported because the packages by the old version of the component remain exported and are still imported by clients
- Services are registered again. The service interface remains intact with the implementation successfully updated.
- The client component learns that the new service is registered and should reacquire the service. Having done this, the client component can continue as before.
- This essentially works by putting the service interface in a Java package to be exported, and hiding the service implementation in an unexported package. This allows implementation updates which aren't binary incompatible with callers, and also allows callers to take advantage of the update right away.

Package dependency is static, that is, packages exported by a stopped component continue to be available to other components. The reason for this is because garbage collection uses the concept of reachability for deciding when to free memory. Therefore, it is not possible to garbage collect objects of exported types because they are likely to still be referenced. This continued export implies that other components can execute code from a stopped component, and the designer of a component should assure that this is not harmful.

A concluding remark in this section on lifecycle operations is that events are broadcast at each state transition, and therefore components can register their interest in being notified about lifecycle changes in other components in the framework. When there are a large number of components in the OSGi Framework, there are scenarios where there is an explosion in the number of events. One example of this is when the Framework is first started and a large number of components need to be started. As each component is started, a starting event is fired. In order to control such an explosion of events, the OSGi Alliance has specified a Start Level Service. This service allows the Framework administrator to impose an ordering on the start-up of components, such as component D starts after C and B, which start after component A.

The OSGi life cycle operations enables components to be installed, removed and updated, and since components may contain services, this also allows services to be acquired and released at run-time without having to shut down the OSGi Framework and JVM. Figure 3.2 shows dynamic reconfiguration of an OSGi application through the invocation of different life cycle operations. The first figure (a) gives an example of an OSGi application which consists of two components. The second figure (b) shows the component install operation, with Component 3 being installed and deployed. It also shows the component update operation, with Component 2 being updated to include a new thread “T3”. The third figure (c) shows the removal of Component 1. Notice the service dynamism. The threads in Component 2 find and bind to the service in Component 3 after the service they were using in Component 1 was unregistered when Component 1 was removed.

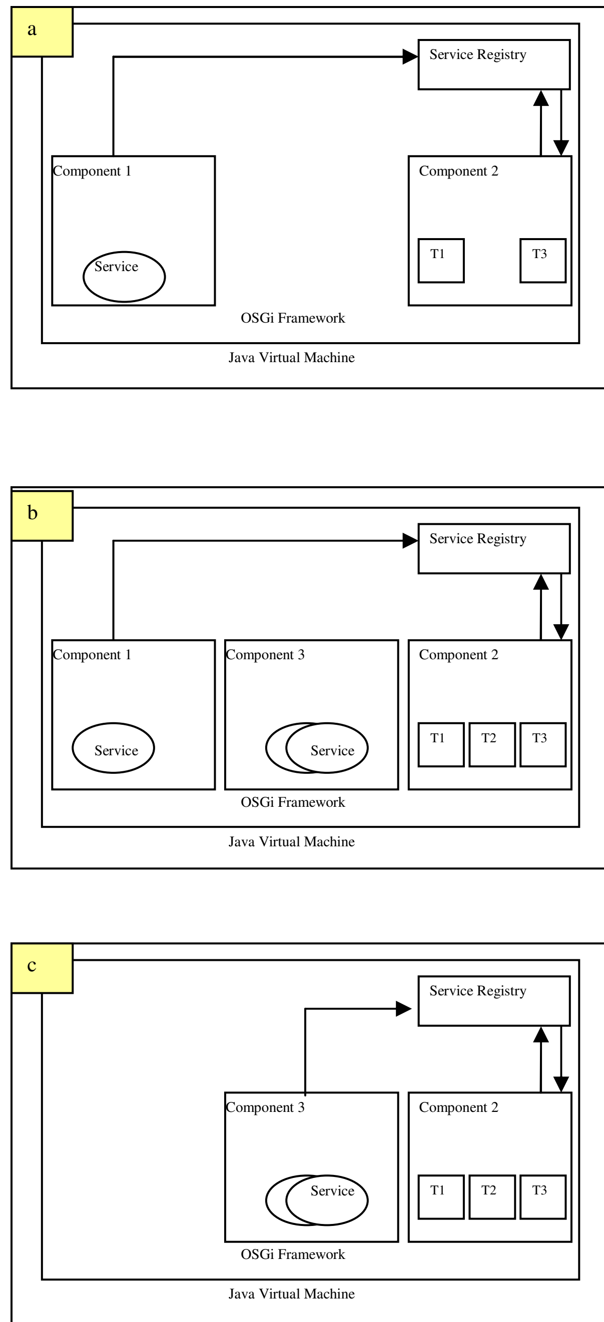

Figure **3**.2 Dynamic Reconfiguration in OSGi

While discussing such an example of application reconfiguration, it is important to note that while performing component updates, the component will be temporarily unavailable until the update procedure has competed. This is the case with the update of Component 2 in Figure 3.2 (b). Despite the fact that the rest of the application remains available for use and will offer at least some utility to users, such component unavailability is undesirable in the case of real-time systems. This is discussed further in Chapter 5.

3.2 Architecture of the OSGi Framework

In order to enable the deployment of dynamically reconfigurable Java applications, the OSGi Framework has a number of responsibilities.

Perhaps the four most significant are:

- Resolving interdependencies among components
- Managing the lifecycle of components
- Maintaining a registry of services
- Firing events and notifying listeners to state changes

These features along with all of the rest of the functionality of the OSGi Framework are divided into layers such that the OSGi Framework has a layered architecture. Lower layers in the architecture are responsible for the tasks associated with low level deployment of an application such as solving component interdependencies, and higher layers are responsible for handling more abstract concepts such as managing the life cycle of components, and maintaining a service registry. Furthermore, some functionality is split across layers such as firing events and notifying listeners when a component's state changes. This is the case because event handling can be related to services, components and the framework itself, the functionality of which are specified in different levels in the OSGi Framework layered architecture.

The OSGi Framework consists of the following layers: security layer, module layer, life cycle layer, and service layer. Figure 3.3 shows the OSGi architectural layers.

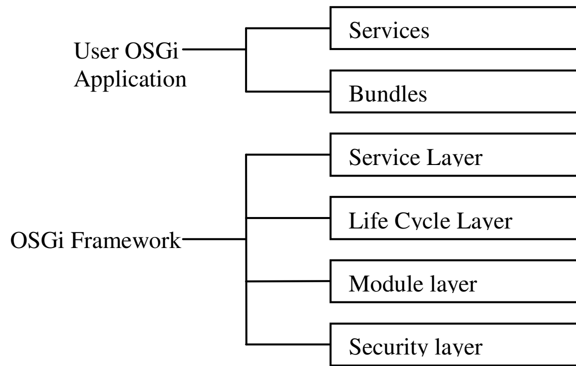

Figure 3.3 Layers of the OSGi Framework

3.2.1 Security Layer

The security layer enhances the Java security architecture with features to combat the risks of downloading and executing code from remote locations, as would typically be the case when installing new components into the OSGi Framework. For example, the security layer allows code to be authenticated based on the location of the component, or based on a digital signature.

However, the security layer is an optional OSGi Framework feature. This is due to the fact that the OSGi Alliance recognises the fact that some implementations of the OSGi Framework will be designed to run on embedded systems with resource constraints. In such systems it is clearly desirable to avoid the overheads of run-time policy checking and the overheads of verifying digital signatures. These overheads can be avoided by using code stubs which always allow classes to be loaded and executed.

3.2.2 Module Layer

The OSGi Framework is used to develop Java applications as a collection of components (bundles). Naturally, these components must be connected in some way in order to form an application; in component theory, matching a component's "requires" interface to the "provides" interface of other components. This is realised in the OSGi Framework by having component developers specify

in meta-data (in the component's manifest file), Java packages which the component is willing to make available (export) to other components, and also specify the Java packages which their component requires (imports). In addition to these imports and exports, there are a number of other ways of declaring dependencies on other components. The module layer specifies these different dependencies and in particular what they mean from a class-loading point of view. It essentially addresses some of the shortcomings of Java's deployment model by defining a modularization model for Java.

More specifically, the module layer is responsible for parsing a component's meta- data in order to determine its dependencies and then performing a process known as resolution. Resolution involves forming a delegation network of component class loaders. This means that when a component's class loader is requested to load a class, there are a number of sources for the class definition, and, therefore, a search for the class must be performed. When a component's class loader is requested to load a class, the search must be performed in the following (simplified) order [17]:

1. If the class is in a java.* package, the request is delegated to the system class loader; otherwise, the search continues with the next step.
2. If the class is from a package included in the boot delegation list (org.osgi.framework.bootdelegation), then the request is delegated to the system class loader. If the class or resource is found there, the search ends.
3. If the class is in a package that is imported using Import-Package, then the request is delegated to the exporting component's class loader; otherwise the search continues with the next step.
4. Search the component's embedded class-path (i.e. within the component itself). If it is found, the component's own class loader is used to load the class

3.2.3 Life Cycle Layer

As mentioned, the module layer is concerned with the low level aspects of components, such as parsing the meta-data, class loading and resolution. The life

cycle layer provides a higher level view of components, and focuses on how components are represented logically in programs. As the name implies, this layer supports the life cycle API for components, defining how components are installed, updated, and uninstalled etc.

A component can be installed by another component or by an OSGi Framework implementation specific means (such as via the command line). When one component installs another, a reference to that component is returned, which can then be used to invoke other life cycle operations on it. To control the life cycle of components for which a component did not install, there are methods which return a reference to a component given the components identifier.

3.2.4 Service Layer

In most applications, inter-component communication is typically required. The module layer provides components with the ability to share or hide Java packages with other components. However, shared (exported) Java packages are unable to take advantage of the dynamicity of components; for example, it isn't possible to update these exported packages at run-time whilst they are in use by other components. This is to ensure referential integrity. If there are no references to exported packages, then it may be possible to update exported packages, although the OSGi specification does not guarantee that this will be done dynamically, i.e., without restarting the OSGi Framework and JVM.

In order to improve on this situation, the service layer supports a publish, find and bind service model. This allows a component to register a service in the service registry, which other components can later retrieve and use. A service is simply a Java object registered under one or more interface names with the service registry. This means that only the service interface needs to be exported, which then allows the implementation of a service to be updated at run-time. This model allows component developers to bind to services only using their interface specifications. The selection of a specific implementation, optimized for a specific need or from a specific vendor, can thus be deferred to run-time.

In terms of the specificities of the publish find and bind model provided by the service layer, a number of functions must be provided. Firstly the service layer maintains a service registry. It also provides a means of filtering services returned from a service lookup from the service registry. Furthermore, when components wish to provide a service, they register the service under the names of the interfaces which the service implements. The service layer is then responsible for performing instance checks at run-time in order to verify that the service object is an instance of each service interface specified. Finally, the service layer is also responsible for dealing with the fact that multiple versions of a class can be loaded into the JVM. In order to prevent class cast exceptions, the service requesting components must not be presented with conflicting definitions of a class. The service layer must therefore ensure that only compatible services are visible to service requesters, that is, service requesters can only acquire services for which both service requester and service provider use the same class loader.

An overview of the OSGi Framework, application bundles, and services is shown in Figure 3.4 [80]. Figure 3.4 essentially shows how application bundles can register and obtain services in order to interact with other application bundles which are deployed on the OSGi Framework, which is itself a Java application executing on a Java Virtual Machine.

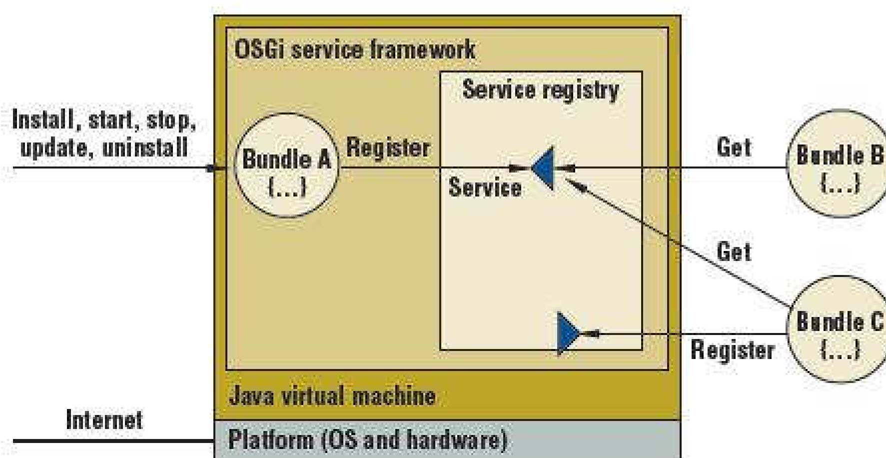

Figure 3.4 OSGi Overview [80]

3.3 Using the OSGi Framework

In this section, the OSGi Framework is discussed from a programmatic viewpoint. Java code is used to illustrate various features of the OSGi Framework. Examples include registering services, requesting services, executing component life cycle operation, and using event handling to tolerate the dynamic availability of services.

3.3.1 Developing a Component

In the component's JAR's manifest file, component developers add OSGi meta data (headers) in order for the OSGi Framework to be able to host it successfully.

Perhaps the most common and significant manifest headers are the following: `Bundle-SymbolicName`, `Bundle-Classpath`, `Import-Package`, `Export-Package`, and `Bundle-Activator`. The `Bundle-SymbolicName` header simply specifies a name for the component. The `Bundle-Classpath` header defines a comma-separated list of JAR file path names or directories (inside the component) containing classes and resources. This is used by the component's class loader for finding classes within the component. `Import-Package` informs the framework that a component requires classes in the explicitly named Java packages in order to execute. As discussed in Section 3.2.2, these imports do not include the standard "java.*" packages and packages provided by the JVM but rather packages that are exported by other components. For example, components need to import the Java packages that contain the interfaces to any services which they intend to bind to. The `Export-Package` header informs the Framework that the component would like to make one or more Java packages available to other components. Finally, the `Bundle-Activator`: manifest header gives the name of a class implementing the OSGi interface "`BundleActivator`". The `BundleActivator` interface has two methods:

- `start(BundleContext)` – This method is designed to provides an entry point to using the OSGi framework, much like the method `public static void main(String[] args)` in a standard Java application. In this method, developers typically allocate resources that a component needs, create and start threads, register services, and more.
- `stop(BundleContext)` – In this method, developers undo all of the actions of the `BundleActivator.start(BundleContext)` method so as to release any resources allocated since component activation. Furthermore, all threads associated with the stopping component should be stopped immediately so as to prevent "runaway" threads, which are discussed in Section 3.6.1

Figure 3.5 shows an implementation of the `BundleActivator` interface. The implementations of the Activator's start method creates and starts a thread in order to perform some application logic, before printing out a string notifying the user that the component has been successfully started. The implementation of the Activator's stop method terminates the thread created and started in the start method, before printing out a string notifying the user that the component has been successfully started

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class Foo implements BundleActivator
{
    public void start(BundleContext ctxt)
    {
        //lengthy computation performed in a dedicated //thread
        Thread t = new LongComputation();
        t.start();
        System.out.println("Bundle Started");
    }

    public void stop(BundleContext ctxt)
    {
        //if not terminated already, call terminate()
        //defined in LongComputation
        ((LongComputation)t).terminate();
        System.out.println("Bundle Stopped");
    }
}
```

Figure 3.5 Bundle Activator

The only parameter of the `BundleActivator` methods is an instance of `BundleContext`. This object is passed to the activator by the framework when the component is either started or stopped. It allows the component to interact with the underlying framework. For example with a `BundleContext` reference it is possible to [17]:

- Subscribe to events published by the Framework.
- Register service objects with the Framework service registry.
- Retrieve ServiceReferences from the Framework service registry.
- Get and release service objects for a referenced service.
- Install new components in the Framework.
- Get the list of components installed in the Framework.
- Get the Bundle object for a component.
- Create File objects for files in a persistent storage area provided for the component by the Framework.

Given the list of capabilities of a `BundleContext` object, it is recommended that a component's `BundleContext` object should not be shared with other components in the OSGi environment.

3.3.2 Controlling the Life Cycle of a Component

Implementations of the OSGi Framework typically provide either a graphical user interface or a command line interface in order to allow a user to, amongst other things, control the lifecycle of components. For example, the Apache Felix OSGi Framework implementation offers a command line interface to the Framework with the following syntax:

Install *location/componentName.jar*
Update *componentID location/NewComponent.jar*
Stop *componentID*
Start *componentID*
Uninstall *componentID*

In addition to users controlling component lifecycle operation manually via the command line, component lifecycle operations can also be performed programmatically, which is OSGi-Framework-implementation agnostic. An example of performing component life cycle operations programmatically is shown in Figure 3.6.

```
public void start(BundleContext ctxt)
{
	Bundle bun = null;
	try
	{
		//To install a bundle
		bun = ctxt.installBundle("http://webserver/bundles/X.jar");
	}
	catch(BundleException be)
	{
	}
	if(bun != null)
	{
		//perform other lifecycle operations
		try
		{
			bun.start();
			bun.stop();
			bun.update();
			bun.uninstall();
		}
		catch(BundleException bex)
		{
		}
	}
}
```

Figure 3.6 Example of Component Lifecycle Operations

3.3.3 Registering and Obtaining Services

If a component contains some services, it usually registers them in the `start` method of the activator class. Registering a service makes it available for use in other components. Figure 3.7 illustrates how the `MyServiceImpl` instance is registered under its interface name `MyService` in the service registry. The service is also registered with a property describing any optimisations that the implementer of `MyService` has used.

```
public class Activator implements BundleActivator
{
	private ServiceRegistration reg;

	public void start(BundleContext ctxt)
	{
		Propeties  props = new Properties();
		props.put("optimisation", new String("accuracy"));
		MyService mserv = new MyServiceImpl();
		reg = ctxt.registerService
		("bundle.service.MyService", mserv, props);
	}

	public void stop(BundleContext ctxt)
	{
		//service automatically unregistered but good programming etiquette
		if(reg != null)
		{
			reg.unregister();
		}
	}
}
```

Figure 3.7 Obtaining Services

As previously discussed, by exporting the service interface (`MyService`), the service provider is able to make multiple implementations of this service available. Figure 3.8 shows a service provider offering multiple implementations of `MyService`. The second implementation offers an optimization of computational speed rather than the quality of service optimization offered by the first implementation.

```
private ServiceRegistration reg,reg2;


public void start(BundleContext ctxt)
{
	Propeties  props = new Properties();
	props.put("optimisation", new String("accuracy"));
	MyService mserv = new MyServiceImpl();
	reg = ctxt.registerService
		("bundle.service.MyService", mserv, props);
	Propeties  props2 = new Properties();
	props2.put("optimisation", new String("speed"));
	MyService mserv2 = new MyServiceImpl2();
	reg2 = ctxt.registerService
		("bundle.service.MyService", mserv2, props2);
}
```

Figure 3.8 Providing Multiple Implementations of a Service

To use a service, service requesters must import the package containing the service interface. In terms of Java code, service requesters must call `BundleContext's getServiceReference` method specifying the name of the service interface in order to obtain a `ServiceReference` to the service. The `ServiceReference` is then passed as a parameter to `BundleContext's getService` method in order to get the service object itself. The reason for this indirection is so as to allow the client to examine service properties before committing to using a service, for example, Figure 3.9 illustrates obtaining a service based on the properties of the service.

```
public class Activator implements BundleActivator
{
	public void start(BundleContext ctxt) throws Exception
	{
		ServiceReference[] ref = ctxt.getServiceReferences
			("serviceName","");
		if(ref == null)
		{
			System.out.println("Service Not Registered");
			return;
		}

		for(i=0; i < ref.length; i++)
		{
			Object o = (Object); ref[i].getProperty("author");
			System.out.println("author = " + o.toString();
		}
	}
}
```

Figure 3.9 Examining Service Properties

In most situations however, service requesters do not need to examine the properties of available services before obtaining a service. Figure 3.10 shows how to obtain a service without examining a service's properties.

```
ServiceReference[] ref = ctxt.getServiceReferences("ServiceNmae");
ServiceName sn = (ServiceName) ctxt.getService(ref);
sn.methodA();
```

Figure 3.10 Obtaining a Service

3.3.4 Services with State

Service requesting components typically share an instance of a service object and so typically share service state. As this is not always desirable, a `ServiceFactory` can be used so as to provide each service requesting component with its own instance of a service rather than sharing a service with other components.

The service object returned by the `ServiceFactory` is cached by the OSGi Framework until the component releases its use of the service. The cached service object will then be returned on any future call to `BundleContext's getService` from the same component. Figure 3.11 illustrates how to use a service factory to return an instance of a service to each requesting component.

```
public class MyServiceFactory implements ServiceFactory
{
	public Object getService(Bundle bundle, ServiceRegistration reg)
	{
		return new MyServiceImpl(null);
	)
	public void ungetService (Bundle bundle, ServiceRegistration reg,
		Object  service)
	{
	}
}
```

Figure 3.11 Service Factories

Service providers register an instance of a service factory rather than directly registering a service implementation (Figure 3.12).

```
ctxt.registerService("MyService", new MyServiceFactory(), null)
```

Figure 3.12 Registering Service Factories

Service requesters obtain the service in the usual way and the service factory is transparent to the caller.

3.3.5 Dynamic Availability

Components can be started or stopped, and services can be registered or unregistered, which makes for a rather dynamic run-time environment. In order to aid programmers to write robust code in the presence of such dynamisms, the OSGi environment introduces three types of events defined in the `org.osgi.framework` package:

- Framework event – occurs if the OSGi Framework experiences errors
- Bundle event –occurs after a component life cycle operation takes place
- Service event – occurs in response to service registration and service unregistration, or when a service's properties are changed.

Figure 3.13 illustrates service event handling.

```
public class Activator implements BundleActivator, ServiceListener
{
      public void start(BundleContext ctxt)
      {
            ctxt.addServiceListener(this);
      }

      public void stop (BundleContext ctxt)
      {
            ctxt.removeServiceListener();
      }

      public void serviceChanged(ServiceEvent e)
      {
            ServiceReference ref = e.getServiceReference();
            switch(e.getType())
            {
                  case ServiceEvent.REGISTERED:
                  System.out.println("registered " + ref);
                  break;
                  case ServiceEvent.UNREGISTERING:
                  System.out.println("unregistered " + ref);
                  break;
            }
      }
}
```

Figure 3.13 Service Events

In order to help OSGi applications manage in the presence of dynamic reconfiguration, the OSGi Framework provides `public java.io.File getDataFile(java.lang.String filename)`, which creates a file in a persistent storage area reserved for each component by the Framework. This feature can be used along with the aforementioned event handling to, for example, save service state when a service providing component is stopped or updated. This is achieved by having the service provider encapsulate the logic to persistently save a service's state in the `ServiceFactory`'s `ungetService` method. The service provider also encapsulates the logic to load any service state in the `getService` method. In addition, the service requester must listen for events notifying it of any required services becoming unavailable or available and call the service factory's `ungetService` and `getSevice` methods respectively in its event handler.

3.4 Applications of the OSGi Framework

In this section, the original use-case for the OSGi Framework is discussed along with a number of other use-cases which have appeared as the OSGi Framework has evolved.

3.4.1 Service Gateways

OSGi was initially an acronym for "Open Service Gateway initiative" in order to reflect the intended use of the OSGi Framework in service gateways (JSR-8: Open Services Gateway Specification [81]). In service gateways [82], there is a gateway device hosting the OSGi Framework which interfaces an internal network of devices and appliances and the Internet. The gateway operator can then download and run service-containing components from the Internet to communicate with the devices attached to the internal network.

The service gateway approach to service provision is particularly useful for providing services to the home such as energy management. In order to avoid costly customer visits by technicians, the energy company could host an OSGi

component containing an energy management service on their company's HTTP server. The operator of the service gateway is then able to download and deploy the component on the customer's service gateway. From there, the energy management service is able to interact with the customer's gas metre over the home network so as to read the metre or perform other diagnostic functions before e-mailing the energy provider with the results. Of course, the protocols used by the energy management service in order to control/interact with the gas metre are not dictated by the OSGi Framework, but are instead left as an open issue so as not to restrict interoperability between devices and an OSGi application.

In addition to energy management, there are many other examples of services that can be deployed on a service gateway. These include home security, climate control such as ventilation, air conditioning and central heating, and home-based healthcare. Figure 3.14 shows an example of a service gateway with such services deployed.

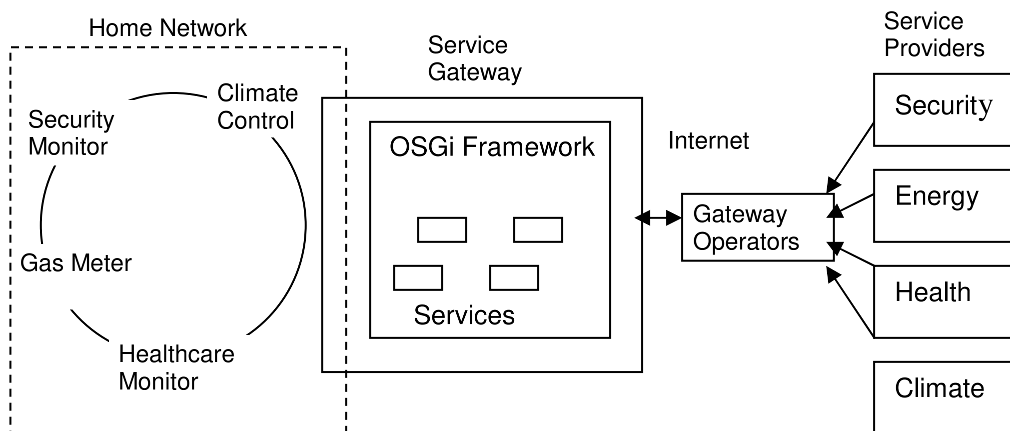

Figure 3.14 Service Gateway

Without using the approach of a service gateway hosting the OSGi Framework for service provision in the home, it would be necessary to have each service use a proprietary mechanism for service provision into the home. Such an approach however is not scalable, and would become unmanageable as the number of services a person subscribes to increases. Furthermore, without the use of a service gateway, it would make it difficult to have independently developed

services interact with one another because they have not been deployed with the principles discussed in Chapter 2 in mind. As an example, it is not possible to have the home theatre service to send a signal to the window blind controller device to automatically lower the automated window blind when a film starts.

3.4.2 Other Applications

The main motivation for using the OSGi Framework in service gateways was to act as a means for providing and managing remote services in a scalable manner. However, since its inception, it became apparent that the dynamic reconfigurability of OSGi applications is a very useful feature to many different types of software applications.

As a result, the OSG Framework is now used in many different application domains, and since it is no longer specifically used in service gateways, OSGi is no longer considered an acronym for "Open Service Gateway initiative" and its associated JSR (JSR-8) was withdrawn. To reflect the generality of the OSGi Framework, it is now also knows as JSR-291 Dynamic Component Support for Java SE [83].

The domains in which the OSGi Framework is used are quite diverse mostly because the member companies of the OSGi Framework are equally diverse, with each organisation looking to promote the use in their respective domain. The OSGi Framework is used in [76]: Integrated Development Environments (IDEs) [84], in home automation products [85], smart phones, in enterprise systems [86], and in the automotive industry [87]. More specifically, in the case of IDEs, the OSGi Framework is used as the underlying framework for the more recent versions of the Eclipse IDE. Version 4 of the eclipse IDE is developed as a collection of OSGi components hosted on the Equinox OSGi Framework implementation. It is the OSGi Framework's application dynamic reconfiguration capability which enables Eclipse users to install plug-ins without shutting the Eclipse IDE down. In the case of the automotive industry, the OSGi Framework has become a standard part of the BMW high-end telematics

platform and is finding its way into many models of Volkswagens. Finally, in the case of enterprise systems, the OSGi Framework is being used to support high end server software such as IBM WebSphere.

3.5 Motivation in Real-Time Systems

As mentioned, the OSGi Framework provides dynamism by allowing components to be installed, updated, and uninstalled from the OSGi Framework at run-time, and by allowing service implementations to be substituted (replaced) at run-time (i.e. without shutting the JVM and OSGi Framework down). This has led the OSGi Framework being used in many different application domains (as discussed in Section 3.4), one domain where the OSGi Framework has not yet been utilised, but where it would be particularly beneficial, is the real-time systems domain. In addition to timing requirements, such systems typically have very high availability requirements i.e. it is important that the system is available for use as long as possible for safety and/or financial reasons but at the same time require (as with all other useful software) evolution/maintenance

Since the OSGi Framework remains functional during such reconfiguration, applications hosted by the OSGi Framework can continue to operate during the reconfiguration period, i.e. the application remains available for use whilst components are being added/removed/updated and whilst services are being registered and unregistered.. This is unlike most other non-OSGi applications which typically must be taken offline for maintenance/evolution purposes that were not anticipated at the time that the application was designed and implemented. This means that the OSGi Framework would provide a suitable environment for performing real-time systems maintenance and evolution while maintaining high levels of application availability. Such dynamic reconfiguration is particularly useful when a real-time application requires frequent maintenance/evolution, as without being hosted by the OSGi Framework, the application would typically have to spend considerable amounts of time being shut down and thus unavailable for use offering no utility to end users.

Whilst the dynamic reconfiguration generally keeps an application available for use during reconfiguration, in standard OSGi (but not in RT-OSGi, as discussed in Chapter 5), the semantics of the component update life cycle operation are to cause the component being updated to transition to the Installed state from the Active state before the update process takes place. As a result, in standard OSGi, there is a blackout period when calling the component update life cycle operation. During this time, the component is unavailable for use until the new version of the component is deployed.

In any case, the important advantage of the standard OSGi Framework's dynamic reconfigurability is that in the worst case, only part of an application will be temporarily unavailable for use and that the application will continue to have some utility during the reconfiguration process. For example, components (new functionality) can be added to an application without making it unavailable for use, similarly removing components can be carried out without affecting the availability of the application, except from the component being removed of course! In order to mitigate some of the issues associated with temporary unavailability of components and services, OSGi events (as mentioned in Section 3.1) provide a means of making the reconfiguration a much smoother process. For example, a component may be updated to register new implementations of a service it has previously registered. In order for service requesters to be aware of this, they should listen for service events so as to prevent attempts to use unregistered service implementations.

In addition to the advantage of increasing application availability, dynamic reconfiguration is also useful for minimising resource usage such as CPU time and memory. This is particularly useful for managing resources when the real-time application operates in different modes of operation. For example when it is necessary to change to another mode of operation (i.e. the application must perform some different functionality), the component-set can be swapped on-demand, that is, required components can be installed, and unnecessary existing components can be removed. Such dynamic reconfiguration ensures that only necessary components are installed and active at any one time thus saving resources that would otherwise by consumed by deployed components which are

currently not required (i.e. components that are not required for the current mode of operation).

Also, in the OSGi Framework, application dynamic reconfiguration can be controlled from a remote location by issuing remote commands to install/uninstall/update components at run-time. This is a necessary feature for evolving real-time software that is deployed in harsh environments where there are many dangers involved in being physically present in the environment in which the system is deployed; for example in a Nuclear Power Plant radiation leak monitoring system, without remote control of the OSGi Framework, it would be necessary to enter areas of the plant which involve being exposed to large amounts of harmful ionizing radiation. The remote dynamic reconfigurability of OSGi applications is also useful for evolving mass produced embedded systems such as in consumer electronics. For example, in consumer electronics where millions of units are sold, it is not feasible to send a technician to each customer to update the software, nor is it acceptable to have customers send their units back to the supplier for update. Instead the software on these units can be evolved remotely.

Finally, OSGi offers more to applications than just the benefits of dynamic reconfiguration (high application availability, low resource requirements, remote reconfiguration control). It provides the benefits traditionally associated with CBSE and SOA. The most significant benefit of using CBSE is software reuse typically through third party component/service development. Using pre-built pre-tested Java components and services leads to reduced time to market and reduced development cost of real-time systems, the reason is that reusability means there is less software to develop. Another advantage of using the OSGi framework for developing and deploying real-time systems is the modularity offered by the underlying CBSE and SOA. This improves the maintenance issues of program comprehension and impact analysis and also, as a result of the high levels of decoupling, there is a great deal of fault containment within components which also impacts application availability. The OSGi Framework also enhances the modularity offered by the underlying CBSE and SOA concepts by creating separate class loaders for each component, and giving component developers the

choice of sharing or hiding their Java packages with other components in the system. This solves problems such as when two different versions of a class are made available to the components within a system.

3.6 Real-Time OSGi (RT-OSGi)

Despite the benefits of using OSGi to develop and deploy dynamically reconfigurable real-time systems, there are a number of problems. OSGi applications are comprised of components and services developed in standard Java. The OSGi Framework is itself developed in Java, and both the OSGi Framework and any applications that it hosts are deployed on a standard Java Virtual Machine. However, as discussed in Chapter 1, it is widely accepted that standard Java is not suitable for use in the development of real-time systems. Reasons for this include issues with memory management, clocks and time granularity, resource sharing, and poor scheduling semantics.

These issues can be solved by using the Real-Time Specification for Java (RTSJ) to facilitate real-time programming in Java, for example, by introducing fixed priority scheduling and real-time threads to Java, as discussed in Chapter 1. As a result, the OSGi Framework and OSGi applications can be hosted on a real-time JVM which supports the RTSJ, and OSGi applications developers can make use of the real-time classes specified by the RTSJ in order to develop the components and services comprising an OSGi application.

However, even when using the OSGi Framework and RTSJ together, problems remain which prevent such OSGi applications from having timing guarantees. These problems are generally as a result of the flexible/dynamic nature of the OSGi framework, with flexibility and predictability often contradictory in nature. More specifically, the issues that prevent timing constraints from being met include: unbounded dynamism, runaway threads, lack of temporal isolation, lack of memory allocation policing, no regard for the effect of application dynamic reconfiguration on the garbage collector, and issues with the worst case execution time (WCET) calculation of application threads. These issues are

briefly discussed shortly and discussed in further detail in subsequent chapters in the thesis.

3.6.1 Issues

In this section, a number of issues with the standard OSGi Framework in the context of real-time systems development and deployment are discussed.

3.6.1.1 Global and Local View – Priority Assignment

Since the OSGi Framework is designed for developing component-based Java applications, and is itself written in Java, one might think that using the OSGi Framework in the real-time domain is simply a case of running the OSGi Framework on a real-time JVM, and writing components using the RTSJ. Unfortunately, such an approach is flawed.

The reason why dynamically reconfigurable real-time systems cannot be built by simply using the OSGi Framework and the RTSJ together is because the OSGi Framework takes the component-based software engineering (CBSE) approach to software development. The central theme of CBSE is independent component development; that is, developers state their component requirements from the system, and that is all. No developer has a global knowledge of the system. In such a situation, it is difficult for a component developer to guarantee timeliness requirements of their component, without knowing the internals of every other component in the system.

To illustrate the above point, consider the problem of priority assignment. If component developers were to use the RTSJ and OSGi, each would (say) assign priorities to the threads within their component using Rate Monotonic priority assignment or Deadline Monotonic priority assignment [88]. However, because a component developer has no knowledge of the threads in other components, the priorities they assign will give a correct ordering within their component but not across components. Table 3.1 shows how the priory ordering within component

C1 is correct and the ordering in component C2 is correct. However the overall ordering is incorrect. The required ordering would be performed by the OSGi Framework once both components are admitted and a global view is available. This is discussed further in Chapter 5.

Component	Thread	Period	Developer Assignment	Required Assignment
C1	T1	10	3	4
C1	T2	15	2	2
C2	T1	13	4	3
C2	T2	19	1	1

Table 3.1 Priorities Assigned by Component Developers and by the System

3.6.1.2 Worst-Case Execution-Time (WCET) Analysis

There are a number of features of the OSGi Framework which make WCET calculation an even more difficult task than it already is. The general issues stem from the fact that threads typically synchronously interact with components and services written by third parties, for which the execution time of such code is unknown to the caller offline. The result is that the calling thread's WCET is affected in an unknown way. Examples of such synchronous calls include service method execution, service factories, component activation and deactivation, and synchronous event handling. As the WCET of application threads is an important parameter to using analysis for guaranteeing an application's real-time requirements, these synchronous calls in the OSGi Framework are particularly troublesome when attempting to use the Framework in developing dynamically reconfigurable real-time systems. These synchronous calls and their affect on the WCET of application threads are discussed in more detail in Chapter 4.

3.6.1.3 Scheduling – Dynamic Availability

The OSGi Framework has unbounded dynamism, where components can be installed, uninstalled, and updated at anytime. In a component-based real-time system, it is necessary to reserve resources for each component in the system. Such a dynamic environment poses problems for resource reservation, there must be bounds placed on the number of components in the system to ensure that new components can be installed only if their timing requirements can be met, whilst ensuring that the timing requirements of existing components are still met by the system. Without such a mechanism, overload situations would likely cause components' threads in the system to miss their deadlines. Dynamic availability also impact on WCET analysis; since service implementations can be updated (substituted) at run-time, this means a changing WCET for any threads using the service. Theses issues are discussed further in Chapter 5.

3.6.1.4 No Temporal Isolation – Denial of Service Attacks

As mentioned, threads can miss their deadlines through incorrect priority assignment, inaccurate WCET, and through system overload due to installing more components than is possible to guarantee resources for. Another way in which threads may miss deadlines is through the lack of temporal isolation [89] in the OSGi Framework. Temporal isolation prevents the timing misbehaviour in one thread from affecting the timing constraints of other independent threads. Without temporal isolation, it is entirely possible for an OSGi component to carry out a denial-of-service (DoS) attack on the OSGi Framework. The DoS attack could exhaust the systems resources such as CPU or memory and thus prevent other components from obtaining their resource guarantees, which are necessary to meet their deadlines. Temporal isolation is discussed further in Chapter 4.

3.6.1.5 GC Reconfiguration and Reconfiguration Analysis

The dynamic reconfiguration of OSGi applications makes it possible to change the application at run-time in ways unforeseen at design and deployment time of the application. As a result, the garbage creation rate and memory requirement of

the application may increase. Without increasing the pace of the garbage collector, garbage related memory exhaustion may occur. As a result, one or more threads may miss their deadlines when on-demand garbage collection occurs on the first request by a thread to allocate some memory on the heap after memory exhaustion has taken place.

To prevent garbage related memory exhaustion, it is necessary to ensure that application reconfiguration only takes place when the pace of the garbage collector can be increased to accommodate the increase in garbage creation. Of course, it is also essential that increasing the amount of time allocated to perform garbage collection does not affect the timing requirements of the application threads. This topic is discussed further in Chapter 6.

3.6.1.6 OSGi Framework is not Real-Time

As the OSGi Framework is written in standard Java and not the RTSJ, various issues need to be resolved. These issues stem from the fact that components will be written in the RTSJ and will need to interact with OSGi Framework classes which are developed in standard Java. The issues include:

1. Memory Assignment Errors – If a real-time thread in a component instantiates an OSGi Framework class whilst in heap memory and then enters scoped memory to execute methods of that object, there may be problems. An IllegalAssignmentError will be thrown if the method creates objects in scoped memory and then attempts to store references to these objects in an instance field. An IllegalAssignmentError prevents dangling references i.e. heap memory referencing objects in a scope which has been released. Also an IllegalAssignmentError will be thrown if a method executing in scoped memory creates objects and attempts to store them in static fields of a class. This is because static fields are stored in immortal memory, and like heap memory, immortal memory cannot reference scoped memory.

2. Memory Leak – In the OSGi Framework, every component has its own class loader. Since the set of components installed in the OSGi Framework will change over time, it is desirable that the memory used by class objects and class loaders can be reclaimed when they are no longer referenced. Generally, a class can be unloaded when its class loader is unreachable. A class loader becomes unreachable when the class loader object itself, the class object and instances of the class object are all unreachable. The reason why class instances must be unreachable is because they hold a reference to their class object which holds a reference to its class loader object. Unfortunately, in the RTSJ, class objects are stored in immortal memory, and developers may store objects in immortal memory. This complicates class unloading. Even if a real-time JVM implementation can detect and reclaim unused class objects in immortal memory, it will be impossible to unload classes and class loaders when an application developer stores an instance of a class in immortal memory. This leads to memory leaks in the OSGi Framework.

3. Poor OSGi Framework Performance -- The OSGi Framework is written in standard Java using ordinary threads. Components written in the RTSJ will be using real-time threads. These threads may lockout the Framework because the component's threads will have priorities higher than the system threads. For example, implementations of the OSGi Framework often provide a user interface for administering the Framework. Depending on the behaviour of real-time threads in components, the administrator may find it virtually impossible to issue commands to the Framework, this is particularly problematic when the administrator is trying to add/remove, or update components in the Framework.

4. Runaway Threads – Currently in the OSGi Framework, developers must program their threads to cooperate with the life cycle of their component. This means that, should a component developer not follow this approach, threads may continue to exist long after their component has been uninstalled from the OSGi Framework. On uninstalling a component, the

OSGi Framework therefore needs to safely terminate all of the threads associated with a component, should the component developer forget to.

These issues are discussed further in Chapters 5 and 6.

3.6.2 Other Approaches to Supporting Real-Time Applications using the OSGi Framework

In spite of the flexibility of the OSGi Framework, by integrating the OSGi Framework with the RTSJ and by extending the OSGi Framework so as to provide a real-time version (which solves the issues discussed in Section 3.6.1), it is possible to provide a predictable environment in which OSGi applications may be dynamically reconfigurable and still have timing requirements. Such dynamically reconfigurable real-time applications have high levels of availability during dynamic reconfiguration, e.g. during application maintenance/evolution. This is particularly beneficial as the application continues to have utility during circumstances when most other software applications would typically have no utility.

Before discussing the details of real-time OSGi (RT-OSGi) in subsequent chapters of this thesis, it is first necessary to conclude this chapter by discussing other approaches in the literature to using the OSGi Framework in developing real-time systems. This is so as to highlight the fact that related works fall short of meeting the goals of this thesis (as discussed above). This will help to emphasise the contribution of this thesis.

The most relevant piece of related work on the use of the OSGi Framework in real-time systems is by Gui et al [90, 91]. In their work, Gui et al state that many applications have real-time requirements such as media processing and control applications and that such applications would benefit from the dynamic reconfigurability offered by the OSGi Framework so as to evolve applications during system operation.

The authors state that the reason why dynamically reconfigurable real-time systems cannot be developed is because component developers do not have a global view of the system. This issue was discussed in Section 3.6.1 along with a number of other issues identified and addressed in this thesis.

In order to solve the global view issue, the authors propose a declarative real-time component model (Declarative Real-time Component Model (DRCom)) to be used over the OSGi Framework, allowing components to declaratively specify real-time contracts. More specifically, OSGi components contain non real-time standard OSGi Java code, native (non Java) real-time code, and an XML document containing details of the real-time constraints of the component such as threads' period, computation time and relative priority. In addition, the XML file also contains the component's dependencies on other components.

In order to support such a real-time component model, the authors propose that the OSGi Framework be extended with a centralised real-time manager and that each component implement a real-time management interface. The management interface which must be implemented by component developers is shown in Figure 3.15.

```
public interface IRealTimeManagement
{
    public boolean deploy();
    public void startRTtask();
    public int getStatus();
    public int setPriority(int priority);
    public int setProperty(String name, int value);
    public int getProperty(String name);
    public int suspendRTtask();
    public int resumeRTtask();
    public int stopRTtask();
}
```

Figure 3.15 the Real-time Management Interface

As dynamic reconfiguration takes place in the OSGi Framework i.e. as the life cycle operations are called, the methods in Figure 3.14 are called by the centralised manager so as to provide a mapping between the OSGi Framework life cycle operations and the externally developed (non Java) real-time application functionality of components. Such a mapping enables the non real-

time part of an application (i.e. the OSGi component) to control/reconfigure the natively developed real-time code of the component in accordance with the life cycle of the component. For example, by having the OSGi "stop" life cycle operation call `stopRTtask()`, the runaway thread issue discussed in Section 3.6.1 of this thesis can be solved by essentially synchronising the life time of native real-time threads with the life time of the OSGi component which created them.

Perhaps a more potential use of the centralised real-time manager of a component's `IRealTimeManagement` interface is the ability to dynamically adapt the real-time behaviour of the real-time thread to environmental conditions or resource availability which may change as a result of the dynamic reconfiguration of OSGi applications. The authors' framework enables adaptation of real-time threads to changing resource availability and changing environmental conditions. The authors name this type of adaptation parameter-based adaptation, the reason for this name is because parameters of the involved threads are adapted at runtime. Of course, in order to support such parameter-based adaptation, real-time threads must be designed with such adaptation in mind. For example, real-time threads should be designed to operate at various levels of quality of service, (QoS).

By designing real-time threads with adaptable QoS, it is possible to provide resource guarantees during OSGi dynamic reconfiguration by implementing QoS adaptation through the get/set property methods of Figure 3.4. These methods help the programmer to dynamically change the real-time tasks' behaviour. For example, when a new component is installed, the centralised real-time manager could then call the `setProperty(String name, int value)` method of each real-time component so as to decease their QoS in an attempt to free up sufficient resources for the real-time part of the newly installed component to meet its deadlines.

Whilst the real-time management model discussed by Gui et al is quite interesting and promising, the authors do not pursue any of the aforementioned

ideas i.e. they do not discuss the details of how runaway threads can be prevented through termination, or how the thread termination is safe in the case of threads holding resource locks. More importantly, they do not give details of how the QoS of the real-time part of components can be modified so as to guarantee real-time requirements. There is no discussion of providing temporal isolation, CPU and memory admission control, and WCET calculation. However, as discussed, these features are required in order to guarantee the timing requirements of real-time OSGi applications and so the work by Gui et al is not adequate for meeting the goals of this thesis. Finally, since Gui et al's work is not in the context of the RTSJ, such OSGi applications developers will not be able to reap the benefits involved in using both the RTSJ and the Java language.

In conclusion, the authors point out that currently their framework focuses on providing a general adaptation framework for real-time systems rather than on providing real-time guarantees in the presence of dynamic reconfiguration. Furthermore, given the fact that their framework does not address the aforementioned issues, which compromise the predictability of real-time applications, it is clear that their framework is unable to meet the goals of this thesis.

In addition to the work by Gui et al, there have been two other pieces of work which relates to the work in this thesis, although neither have the same motivation as this thesis and neither is quite as comprehensive as the work by Gui. In Section 5 of [92], Kung et al describe ideas for providing cost enforcement (to support temporal isolation) in the OSGi Framework so as to provide resource guarantees to each component. Their work is discussed in more detail in Chapter 4. However, suffice it to say that as the aims of their work are not the same as this thesis, they do not attempt to solve all of the issues discussed in Section 3.6.1, and as a result, their work is insufficient to meet the goals of this thesis. Finally, in Miettinen et al [93], the authors modified the OSGi Framework so as to enable the monitoring of a components resource consumption. Essentially, they add all of the threads in a component to a thread group, and provide a monitoring agent to collect resource usage information. Their work is similar to the work in this thesis in that the authors are providing

cost monitoring at the component level (discussed in Chapter 4). However, the motivation for the work by Miettinen et al is on improving the performance of standard Java applications. Their monitoring tools are intended to be used during testing so as to identify inefficient components before the application is finally deployed.

3.7 Summary

.

The primary aim of the OSGi Framework is to provide the ability to develop and deploy dynamically reconfigurable Java applications. The implication of this is that the software architecture can change at run-time. This is made possible by the component life cycle operations of OSGi, which enable application components to be installed, updated, and removed during run-time. Since components may contain services, services may also by registered/removed by service providing components and acquired/released by service requesting components at run-time. The dynamic reconfiguration offered by such services and components has many advantages. It improves system availability by not having to take the application offline for maintenance/evolution purposes. In addition, dynamic reconfiguration minimises application resource usage by only having the components and services comprising the current mode of operation installed. Furthermore, dynamic reconfiguration can be performed remotely which is useful when it is infeasible to enter the environment of the deployed application. However, there are various issues which must be addressed by providing real-time extensions to the OSGi Framework such as lack of temporal isolation, unbounded dynamism and its affect on real-time constraints, lack of reconfigurable garbage collection, and WCET calculation issues specific to the OSGi Framework. In the most relevant piece of related work by Gui et al, the authors do not address these issues and, therefore, their work can not be used to meet the goals of this thesis. Consequently, in subsequent chapters of this thesis, solutions to these issues are provided so as to meet the thesis goals.

4

Temporal Isolation and Worst Case Execution-Time (WCET) Analysis

4.1 Temporal Isolation

Temporal isolation [89] prevents the timing misbehaviour in one thread from affecting the timing constraints of other independent threads. Temporal isolation can be broken if a thread uses excessive amounts of a resource such as the CPU. An example of this is when the threads within one components use more CPU time than was specified in schedulability analysis; the result compromises the schedulability of the system and other threads in the system may be starved of the CPU and may miss their deadlines. Another example of breaking temporal isolation can be seen through memory management. Threads may use excessive amounts of memory, starving other threads of memory which may then cause those threads to block on memory exhaustion. Threads may also break temporal isolation indirectly through their impact on garbage collection. The issue of temporal isolation and CPU time overruns is discussed in this chapter, and the effect of memory management on temporal isolation is discussed in Chapter 6.

Currently, the OSGi Framework does not provide temporal isolation amongst threads or bundles. This is undesirable for two reasons. Firstly, OSGi bundles may be developed by third parties and so it is difficult to be confident that such components are trustworthy and are not faulty or malicious. Without temporal isolation, third party faulty/malicious components' threads would be able to

starve lower priority threads of the CPU. This may cause threads to miss deadlines. The second reason is because the OSGi Framework allows components comprising multiple independent applications to be deployed together. Clearly, it is unfair to allow an application's components' threads to monopolise the use of the CPU causing other applications' components' threads to miss their deadlines. Therefore temporal isolation is imperative in the OSGi Framework.

4.1.1 Providing Temporal Isolation

There are two main approaches to providing temporal isolation to applications, time slicing and execution-time servers. These two techniques are further discussed in this section.

4.1.1.1 Time-Slicing

In time-slicing [94], applications are scheduled using cyclic scheduling. Cyclic scheduling [3] requires that the temporal axis be divided into time slices of equal length, with one or more application tasks allocated to the time slices. During each time slice, only the tasks allocated to that time slice are eligible for execution. After completion of all of the time slices, the schedule repeats itself. The duration of a time slice is called a minor cycle, and the minimum period after which the schedule repeats itself is called the major cycle.

For temporal isolation purposes, time slices should not be shared between applications/components. Furthermore, before the threads belonging to the next scheduled time slice are granted the right to execution, the OS should ensure that the tasks belonging to the current time slice terminate or suspend at the minor cycle boundary. In this way, the OS ensures that tasks access the CPU only at pre-defined intervals of time.

As an example of using time slicing to provide temporal isolation, consider the ARINC-653 specification. The ARINC-653 software specification describes the

standard Application Executive (APEX) partitioning kernel and associated services that should be supported by safety-critical real-time operating systems (RTOS) used in avionics [95]. To support avionics applications, the APEX kernel provides temporal and spatial isolation. These features help to provide fault containment, which is essential in avionics and other safety critical applications.

4.1.1.2 Execution-Time Servers

Execution-time servers provide a means of guaranteeing a collection of application threads with a computation-time per period. There are a number of different types of servers for both fixed priority and dynamic priority based scheduling. However, as an RTSJ implementation is only required to provide a fixed priority pre-emptive scheduler, only fixed priority servers are discussed.

Assuming fixed priority scheduling, there are three well known types of execution-time server: Periodic (Polling) Server [96] , Sporadic Server [97], and Deferrable Server [98]. With Periodic Servers, when a server becomes active at the beginning of its period, if there are application threads ready to use the server's capacity then they execute until either completion or until the server's capacity is exhausted. If there are no threads ready to use the server then the server suspends itself until the beginning of its next period. In this case, the time allocated to the server is not preserved but is used by other periodic threads executing in the system. If a thread arrives just after the server has suspended, it must wait until the beginning of the next server period, when the server capacity is replenished to its full capacity [3].

Deferrable Server differs from Periodic Server in that if no tasks are ready to use the server, the server may delay (defer) its execution thus preserving its capacity throughout its period. This means that, unlike Periodic Server, Deferrable Server preserves its server capacity throughout its server period and threads arriving late into the server's period may execute if the server has capacity remaining.

Sporadic Server is different from both Periodic and Deferrable Server in its server capacity replenishment policy. In Sporadic Server, its capacity is only

replenished once it has been used. For example, if a thread uses x amount of server capacity at time t_1, the server capacity is replenished by an amount equal to x at time t_1 plus the server period.

To support such execution-time servers, an OS/JVM needs to provide at least a two-level hierarchical scheduler and cost enforcement.

Hierarchical scheduling [99] provides an approach to scheduling execution-time servers for a number of separate applications/application components using multiple levels of schedulers. Each server is assigned an execution capacity and a replenishment period. These parameters essentially enable each application/component to be assigned a fraction of the overall CPU capacity.

In order to determine which components' server should be allocated the processor at any given time, a global scheduler is used. Each server then uses its own local scheduler to determine which of its tasks should actually execute. For example, assuming that fixed priority pre-emptive scheduling is used by both the local and global schedulers, each server is assigned a priority that is used by the global scheduler in order to determine which of the servers with remaining capacity and threads ready to execute should be allocated the processor. When a server is made active by the global scheduler, the highest priority thread within the server is executed by the server's local scheduler. Of course, although the above example assumed fixed priority pre-emptive scheduling is used by both local schedulers and the global scheduler, in hierarchical scheduling, it is possible for application/component developers to utilise their own local scheduling scheme such as cyclic scheduling, dynamic priority based scheduling, and fixed priority based scheduling. Figure 4.1 shows the general scheme of hierarchical scheduling, with different application/component servers utilising different local scheduling algorithms.

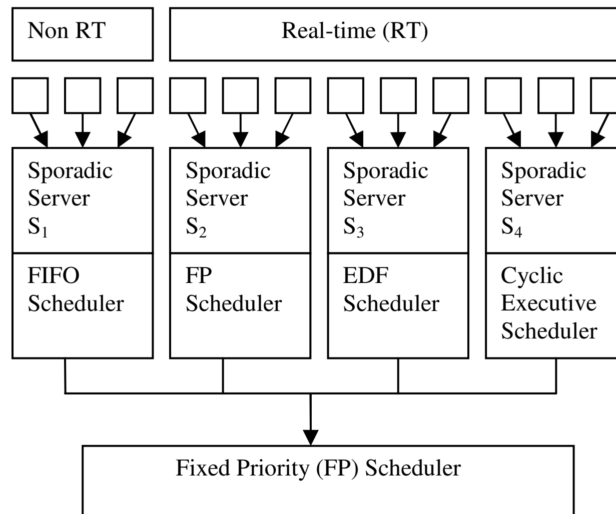

Figure 4.1 Hierarchical Scheduling

In order to support execution-time servers, cost enforcement is required to essentially guarantee each application/component server its computational capacity per replenishment period.

Cost enforcement [100] is a means of monitoring the amount of CPU time a server (or more generally a thread) consumes and taking some action when it consumes all of its predefined CPU allocation/budget. The typical action is to deschedule the server until its CPU allocation/budget is replenished e.g. at the beginning of the period of a server.

In the context of the RTSJ, cost enforcement can be provided at three levels: the Operating System (OS) level, the Java Virtual Machine Level (JVM), or at the application level. Cost enforcement at the OS level involves the scheduler taking the remaining CPU budget into consideration during scheduling decisions. In addition, the OS needs to periodically decrement the currently executing server's CPU budget and re-evaluate whether it is the most eligible server for execution based on whether the currently executing server's budget has been consumed.

Providing cost enforcement at the JVM and application level is necessary when the OS scheduler does not consider servers to have CPU budgets and therefore

makes no attempt to monitor their remaining budget or use such information to base scheduling decisions on. At the JVM level, a high priority JVM thread can monitor the cost consumed by application servers and notify the OS to deschedule any servers which overrun their budget. Similarly, application can themselves take such an approach if the JVM does not support cost enforcement.

4.1.1.3 Comparison of Time Slicing and Execution-Time Servers

From the discussion of time slicing (as provided by the APEX kernel), it is clear that such an approach to temporal isolation has many benefits over using servers. Perhaps the most important advantage of time slicing is that it provides a much stronger notion of task isolation because of its use of cyclic scheduling as opposed to the dynamic or fixed priority-based scheduling used with servers. This makes it easier to certify the safety of many hard real-time systems.

Despite the advantage of time slicing, there are two issues. Firstly, tine slicing has the disadvantage that it is not bandwidth preserving. This means that the CPU-time allocated to a time slice can be lost to the system if there are no threads to execute. This is unlike execution-time servers whereby background threads would be able to utilise any server capacity that is unused by a server's threads. The second and more significant issue with time slicing is that the number and size of time slices (minor cycles) that comprise a major cycle must be configured pre-deployment time and is fixed i.e. the schedule cannot be modified during run-time. Clearly such a static scheduling scheme is unsuitable for dynamically reconfigurable real-time systems such as RT-OSGi applications. Using time slicing would require an upper bound on the number of RT-OSGi components to be deployed to be made. In addition, the timing requirements of the entire component's' threads would need to be known and fixed at deployment time in order to find the number and length of time slices. As a result, time slicing is not considered further in this thesis for RT-OSGi and execution-time servers are selected as the approach to providing temporal isolation in RT-OSGi.

4.1.2 RT-OSGi Temporal Isolation Extensions

In order to prevent CPU related breakage of temporal isolation, it is necessary to reserve each component enough CPU time for its threads to meet their deadlines. Such a reservation means that regardless of the behaviour of other independent threads, each thread is guaranteed to have its CPU requirements met. To provide a CPU reservation for components, two features are required: admission control (discussed in Chapter 5), and CPU cost enforcement (discussed in this chapter). The admission control will bound the number of components (and thus threads) deployed on the OSGi Framework, and as a result, will control the CPU load. For example, if adding a new component would cause threads to miss deadlines; the request for deployment is rejected. The cost enforcement will ensure that those components and threads that pass admission control, and are thus deployed, do not use more than the CPU time specified in admission control (specifically, in schedulability analysis).

Simply controlling the CPU load in terms of the number of components deployed may not be effective since it relies on the CPU-time requirements specified by components being accurate. Without cost enforcement actually bounding a component's CPU usage to that specified, the component has unrestricted use of the CPU and may accidentally (through errors in Worst Case Execution-Time (WCET) calculation), or deliberately (through a CPU denial-of-service attack) use the CPU more than specified after it has passed admission control and been deployed. As a result, without cost enforcement, deployed component may starve other components of the CPU.

The combination of enforcing a bound on the CPU usage of currently deployed components/threads (through cost enforcement), and, preventing new components/threads from being deployed when this would lead to insufficient CPU time for currently deployed components/threads (through admission control), gives components/threads resource reservation guarantees. However, the reservations are not hard in the sense that they are not guaranteed from the point of view of the Operating System (OS). This means that entities outside of

the Java Virtual Machine (JVM) may be able to take CPU time from RT-OSGi threads. Therefore, it is assumed that no other application threads are running on the same machine as RT-OSGi. To remove this assumption, some kind of contract between the JVM and OS is required. Such contracts are discussed in [101].

Although the time slicing of the APEX kernel is inappropriate for use in RT-OSGi as previously discussed, various features of the APEX kernel would also be beneficial in RT-OSGi such as spatial isolation and the fine grain management of various resources. One way of achieving this is to integrate RT-OSGi with JSR 121 and JSR 284.

JSR 121 [102] defines an "Isolate" API which allows for multiple isolated computations (Isolates) to execute within a single JVM. Each Isolate has its own logical heap space. Such isolation is much more powerful than the isolation offered by the OSGi Framework. The OSGi Framework creates a separate class loader for each component, this provides separate namespaces. However, the Bootstrap class loader loads the core Java classes (such as `java.lang`, `java.io` etc), these classes are therefore shared across components. This means that static members of core classes are shared across components. One potential problem of this is that synchronized static methods may cause blocking of threads across components.

JSR 284 [103] defines a resource management API, the purpose of this API is to allow the availability of resources to be queried, and if available, reserved and consumed. There is work in progress [104] concerning the integration of such application isolation and resource management within a JVM. Their work also looks at running the OSGi Framework on such a partitioning JVM. For example installing components in separate partitions, but only when there are enough resources available, and allowing a components resource usage to be monitored.

However, insufficient research has been carried out into integrating these APIs with the OSGi Framework so as to provide spatial isolation and fine grain resource management to OSGi applications. These features are not discussed

further in this thesis but are considered as interesting directions for future work on RT-OSGi.

4.1.2.1 Servers in RT-OSGi

As discussed in Section 4.1.1.2, to provide temporal isolation through execution-time servers, cost enforcement and hierarchical scheduling are required.

Cost Enforcement in RT-OSGi

As Section 4.1.1.2 discussed, there are three approaches to cost enforcement provision, the OS, JVM and application level. Although the preference is to have cost enforcement provided at the lowest layer possible i.e. at the OS level because it incurs fewer overheads, no widely used OS provides cost enforcement. Of course some research OS may provide such cost enforcement but they are not in widespread use.

Similarly, it is preferable to provide cost enforcement at the JVM level rather than the application level as it incurs fewer overheads. However, cost enforcement is an optional feature of the RTSJ i.e. RT-JVMs supporting the RTSJ may or may not provide cost enforcement. The reason why cost enforcement is not a mandatory part of the RTSJ is because it would constrain the use of the RTSJ to hardware architectures which support cost enforcement e.g. only processors that provide time stamp counter functionality. Unfortunately, none of the widely used RTSJ implementations provide cost enforcement.

Clearly, not having a widely used OS or RTSJ implementation support cost enforcement is problematic since one of the design goals of RT-OSGi is to make it as accessible to as many users as possible. Having RT-OSGi depend on an OS/JVM which is not in widespread use will inhibit the utility of the work in this thesis. For this same reason, it was decided not to modify a widely used JVM or implement a new JVM in order to provide the necessary temporal isolation for RT-OSGi.

Although no widely used JVM provides cost enforcement, at least one widely used JVM does support cost monitoring, another optional feature of the RTSJ. Cost monitoring [105] is similar to cost enforcement in that it keeps track of the CPU consumption of a thread[3]. The only difference is that upon detecting a cost overrun it doesn't deschedule the thread. Instead, it fires an event to notify the application and allows the application to recover from the overrun by releasing a user-defined cost overrun asynchronous event handler. Thus, cost enforcement-like functionality and temporal isolation can be provided at a higher level than the OS/JVM level by using cost monitoring and a cost overrun handler (the code to be executed upon a cost overrun). Within the cost overrun handler, cost enforcement-like functionality can be provided by using one of the following approaches[107]:

1. The cost overrun handler can fire an AsynchronouslyInterruptedException into the method which is causing the thread to overrun. The method will then asynchronously transfer control to a recovery block. This requires the offending method to be asynchronously interruptible.
2. The cost overrun handler can set a flag to indicate that the thread has overrun, the thread can then poll the flag for notification of an overrun and try and recover.
3. The cost overrun handler can simply reduce the priority of the overrunning threads to a value low enough to enable other threads to make progress.

Any of the aforementioned approaches could support cost enforcement, although the first two options involve the application being designed in a very cooperative manner i.e. to either be asynchronously interruptible, or to poll an application-defined flag for overrun notification. Therefore the third choice is the preferred choice in RT-OSGi. As a note, since this approach to cost enforcement does not deschedule threads but rather lowers and raises their priorities, it only gives cost

[3] Cost monitoring is likely to be a required feature in RTSJ version 1.1 [106. JCP. *JSR 282: RTSJ version 1.1.* 2009 [cited 13th January 2011]; Available from: http://jcp.org/en/jsr/detail?id=282

enforcement at the thread's priority level. By this it is meant that the thread may use more than its cost/budget but at a background/non-real-time priority. Clearly this is not problematic because RT-OSGi only requires that the cost enforcement support temporal isolation between real-time threads. Lowering the priority of cost-overrunning threads to a non-real-time priority achieves such temporal isolation between real-time threads. Of course, the effect of blocking and priority inheritance should be taken into account. This is discussed in Chapter 5.

Regardless of the approach used to provide cost enforcement-like functionality from cost monitoring (i.e. regardless of whether the cost overrun handler uses AIE, polling or directly lowers thread priorities) there is a major issue: Using cost monitoring requires cooperation on the component developer's part. As mentioned, cost monitoring simply informs the thread that it has overrun, the component developer may chose to ignore this. This may happen for two reasons: firstly, because providing cost enforcement-like functionality requires extra component design and development effort. The developer must develop a cost overrun handler, and also design threads to be cost-enforcement cooperative, for example when polling for overrun notification or using asynchronous transfer of control. Secondly, the component developer may have no incentive to make the extra effort because they will not directly benefit from the extra coding effort. Even if component developers are fully cooperative, there is still a reliance on them. It is preferable that the RT-OSGi Framework take the responsibility of providing cost enforcement. This is achieved by extending the OSGi life cycle operations (install and uninstall etc) so as to automatically provide cost enforcement.

To provide temporal isolation with the previously mentioned approach to cost enforcement, when a component is installed in RT-OSGi, an instance of the RTSJ's ProcessingGroupParameters (PGP) class [108] is created for the component, and all of the component's threads are added to the PGP. The PGP class allows multiple threads (or more generally Schedulables) to be grouped together, and assigned a group budget (server capacity) per period (replenishment period), thus the PGP acts as an accounting mechanism for the threads in the group much like an execution-time server. As a note, the PGP

parameters are assigned so as to ensure that all of the threads executing under the PGP are schedulable. This PGP (server) parameter selection process is discussed further in Chapter 5.

Using PGPs with the cost enforcement previously described, the JVM monitors the CPU time used by the threads in the processing group, upon detecting that the threads have consumed their PGP's budget, an asynchronous cost overrun event is fired by the JVM. The component's PGP's asynchronous cost overrun event handler is released. This handler will lower the priorities of the component's PGP's threads to a background priority such that the threads in other components in the system can make progress.

Unlike servers however, the execution eligibility of threads in a PGP are not automatically returned to their original value upon PGP budget replenishment; i.e. in the case of RT-OSGi, the threads' priorities are not automatically raised to their original values on PGP budget replenishment. Budget replenishment can however be easily implemented through the PeriodicTimer and AsyncEventHandler RTSJ facilities and thread priority manipulation. The general approach is to create a periodic timer for each PGP with a period equal to the PGP budget replenishment period. Rather than having the timer fire the associated budget replenishment event periodically, the event firing should be disabled until a budget overrun occurs. At this point, the "budget replenishment" event firing should be enabled so that on the next replenishment period after an overrun, the replenishment event is fired and the associated replenishment event handler is subsequently released. The budget replenishment event handler should raise the PGP's threads' priorities back to the value they had before they were lowered on budget overrun, in addition, it should also disable the periodic event firing associated with the budget replenishment periodic timer until the next cost overrun. Figure 4.2 Shows the process of lowering a PGP's threads' priorities on PGP cost overrun, and the process of raising the PGP's threads' priorities on the PGP's next budget replenishment period.

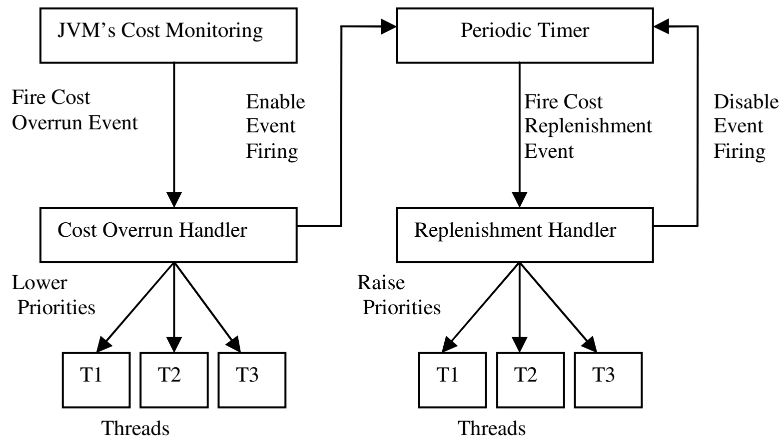

Figure **4**.2 Model of Cost-Overrun and Budget Replenishment

As a note, temporal isolation only requires manipulation of thread priorities upon server budget overrun and does not need to periodically raise and lower the thread priorities. This is because RT-OSGi is mostly concerned with periodic and sporadic threads (as these can be given timing guarantees pre-component deployment-time), which, provided they abide by their estimated Worst-Case Execution Time (WCET) and Minimum Interarrival Time (MIT) constraints, should not overrun their CPU budget e.g. a periodic thread will block for its next period before using the entire group budget. If aperiodic threads are executing under a server, they may always consume the entire server budget and in such a case, the periodic timer for firing replenishment event to release the replenishment handler could well be permanently enabled. However, there appears to be little benefit in keeping the replenishment event firing enabled as opposed to enabling it after an overrun and disabling it after the following budget replenishment.

By creating a PGP for a component and having RT-OSGi provide cost enforcement by creating and adding an asynchronous cost overrun handler for each components' PGP, a component's threads are unable to collectively use more than an amount of computation time equal to the server capacity per server replenishment period at a real-time priority level. In this way, the temporal isolation acts as a form of run-time policing by ensuring a component's threads

are never able to use more than the CPU time they were guaranteed on admission control. Admission control is discussed in Chapter 5, and includes the server parameter (budget an replenishment period) selection amongst other forms of analysis.

In order to provide the aforementioned RT-OSGi temporal isolation, some of the RTSJ classes require extending, namely, the PGP class needs subclassing and so do any classes implementing the RTSJ's Schedulable interface such as the RealtimeThread and AsyncEventHandling classes. These extensions are discussed in the upcoming sections.

Subclassing classes implementing Schedulable

As discussed, it is undesirable to rely on the cooperation of component developers to provide temporal isolation. A more suitable approach is to have threads perform any work required to support temporal isolation in their constructor. Since changing the semantics of RTSJ classes is also undesirable as it would restrict the usability of RT-OSGi by making it incompatible with the standard RTSJ, the RTSJ classes implementing the RTSJ's Schedulable interface are sub classed.

The Schedulable interface is the RTSJ's means of abstracting beyond threads to any entity which can be scheduled, for example, asynchronous event handlers. This is a useful abstraction when programming since it makes it clear that, for example, AsyncEventHandlers are also under the control of the scheduler.

Regarding the functionality required by the subclass versions of the RTSJ classes implementing Schedulable to support temporal isolation, the subclasses must:

- Set the schedulable object's PGP to the one belonging to their component so that all of a component's Schedulables belong to the same PGP (so as to account for CPU usage of a component).

- Pass a self reference to the component so that upon cost overrun or cost replenishment, the PGP can manipulate the priorities of all of the threads associated with a component.

Figure 4.3 shows the constructor of a class (OSGiRTT), which RT-OSGi application developers use to create real-time threads with temporal isolation. Other extensions to this class relating to WCET analysis, admission control and memory management are not shown here.

```
public OSGiRTT(BundleContext bc, SchedulingParameters sp)
{
    super();
    b = (RTBundle) bc.getBundle();
    //allowing "this"  to escape the constructor is unsafe
    //for concurrency reasons. Therefore in reality,
    //this is called after construction
    b.addSchedulable(this);
    setSchedulingParameters(sp);
    setProcessingGroupParameters(b.getPGP());
}
```

Figure **4**.3 Constructor of the OSGiRTT class

The OSGiRTT constructor is simple to explain. When a component is installed and started in the OSGi Framework it is passed a BundleContext object. The BundleContext object allows the component to interact with the Framework, for example, registering services, installing new components, and subscribing to events. OSGiRTT uses the BundleContext object in order to obtain the object representing its encompassing component. This object is extended so as to maintain a list of all of the Schedulable objects which belong to the component, with the OSGiRTT constructor adding a self reference to this list. This allows the cost overrun handler of the component's PGP to iterate through the list lowering the priorities of all of its Schedulables. Finally, the constructor sets its PGP to be the one belonging to its component. This is so that its resource usage is accounted for by its PGP (server). The other RTSJ classes implementing Schedulable, namely, AsyncEventHandler, and NoHeapRealtimeThread, are extended in the same way in order to support temporal isolation.

Subclassing ProcessingGroupParameters

The ProcessingGroupParameters class also requires extensions in order to support temporal isolation by providing PGPs with execution-time server semantics. This involves:

- Providing a cost overrun handler to lower the priorities of all of the Schedulables of the component associated with this PGP.
- Creating and starting a timed event to correspond to replenishment time, and providing an AEH to actually raise the associated component's threads priorities back to their original values

Figure 4.4 shows the sub-classed version of PGP used in RT-OSGi.

```
package uk.ac.york.rtosgi;
import javax.realtime.*;
import org.osgi.framework.*;

public class OSGiPGP extends ProcessingGroupParameters implements
Comparable
{
	private HighResolutionTime start;
	private Bundle b;
	private PeriodicTimer pt;
	private AsyncEventHandler costOverrun = new AsyncEventHandler()
	{
		public void handleAsyncEvent()
		{
			Bundle b =  OSGiPGP.this.getBundle();
			RTBundle rt = (RTBundle) b;
			rt.lowerPriority();

		}
	};

	private AsyncEventHandler rep = new AsyncEventHandler()
	{
		public void handleAsyncEvent()
		{
			Bundle b =  OSGiPGP.this.getBundle();
			RTBundle rt = (RTBundle) b;
			rt.raisePriority();
		}
	};
```

```
 public OSGiPGP(HighResolutionTime start, RelativeTime period,
            RelativeTime cost, RelativeTime deadline, AsyncEventHandler coh,
                AsyncEventHandler missHandler)
    {
            super(start, period, cost, deadline,null, missHandler);
            setCostOverrunHandler(costOverrun);
            pt = new PeriodicTimer(new
                RelativeTime(period.getMilliseconds(),0),
                        this.getPeriod(),this.rep);
            //timer is counting but not firing event, only need to fire when we have overrun
            // and only for one firing, then disable firing until next overrun
            // this avoids overhead of unnecessarily firing event and releasing handler
           //periodically
            pt.start(true);
    }

    //so AEH can manipulate threads in a Bundle
    public Bundle getBundle()
    {
            return this.b;
    }

    public void setBundle(Bundle b)
    {
            this.b = b;
    }

    //methods to support firing of replenishment handler, should only enable
    //the firing of one replenishment event after an overrun
    // after the replenishment handler has been released, the timer should be disabled
    public void disableFiring()
    {
            pt.disable();
    }

    public void enableFiring()
    {
            pt.enable();
    }

    public void destroy()
    {
            pt.destroy();
    }
}
```

Figure 4.4 OSGiPGP Skeleton Class

The OSGiPGP class requires some explanation. It defines two asynchronous event handlers, costOverrun and replen. The constructor of this class sets the cost overrun handler and other parameters of the super class. Once the group budget has been exceeded, the costOverrun handler executes and lowers the priorities of

all of the schedulable objects in the component associated with the PGP. The constructor also sets a timer to fire an event when the associated components threads priorities are to be raised to their original values as a result of group budget replenishment. Finally, the get and set methods provide the link between a PGP and a component. The OSGi Framework will create a PGP for a component, and set the link between a component and its PGP by calling `setBundle(…)`. The replen and costOverrun asynchronous event handlers then use `getBundle(…)` to manipulate the associated components threads.

To summarise this section, temporal isolation is provided at the component level by having the OSGi Framework create a PGP object for each component in the system. Any threads created within a component have their cost overrun handler set to the one associated with their defining component. The cost overrun handler for the PGP lowers threads' priorities on overruns. This approach assumes that it is always safe to take immediate action on overrunning threads. However, in some applications, the highest priority thread must continue to execute at the highest priority even after it has overrun. This may be necessary to keep the system stable or to bring the system into a safe state. Although the aforementioned current approach does not allow for this, it is relatively simple to add such a facility. For example, a delay is added after cost overrun to allow for threads to put the system into a safe state before the cost enforcement functionality is executed. This delay can also be used to ensure that the threads executing under a server release any locks they are holding after an initial (soft) cost overrun, in order to prevent the risk of the cost enforcement mechanism of RT-OSGi from breaking temporal isolation between real-time threads. This scenario may occur because, if a thread holds locks and has its priority lowered to a background priority level on cost overrun, it will likely result in priority inversion and the subsequent activation of the RTSJ's priority inheritance mechanism. The priority inheritance mechanism of the RTSJ will then result in a break in temporal isolation and potential deadline misses for the thread requiring the lock.

As a note, ensuring that locks are released before threads' priorities are lowered as part of server capacity exhaustion is not adequate to prevent priority inversion and a break in temporal isolation. For example, priority inversion may occur when a thread holds a lock and is pre-empted while the thread's server still has capacity remaining, or when a thread acquires a lock after its server capacity is exhausted, while it is running at a background priority. In order to solve these issues, the blocking time that each thread could potentially experience should be calculated and lock acquisition should not be permitted after threads have had their priorities lowered. This is discussed further in Chapter 5.7.

Enforcing Use of RT-OSGi Extension Classes

Although temporal isolation can be supported by having component developers use the subclasses of the RTSJ classes specifically designed for RT-OSGi such as OSGiRTT, developers may break temporal isolation by either accidentally or deliberately directly using classes which implement the Schedulable interface such as RealtimeThread.

This problem can be solved by substituting references to the RTSJ classes such as RealtimeThread with the corresponding RT-OSGi subclasses which support temporal isolation such as OSGiRTT. To do this, instances of classloaders used by the OSGi Framework to load classes for bundles are replaced with RT-OSGi classloaders. In the `loadClass(…)` method of the RT-OSGi class loader, the bytecode is read from the .class files contained in components and the bytecode is manipulated/rewritten [109] so as to replace references of (say) RealtimeThread with OSGiRTT and AsyncEventHandler with OSGiAEH etc. The overhead of such bytecode transformations are insignificant since this process takes place in the non real-time start-up phase of application components.

4.1.2.2 Simulating Hierarchical Scheduling in RT-OSGi

To attain the semantics of execution-time servers in RT-OSGi; i.e. to ensure that the threads under control of a server execute only when the server is eligible for execution, hierarchical scheduling is required, as discussed in Section 4.1.1.2. However, the RTSJ does not support hierarchical scheduling. In order to solve this issue, the semantics of hierarchical scheduling must be simulated using the RTSJ's single level fixed priority pre-emptive scheduler. As a note, in addition to not wishing to make changes to the RTSJ's scheduling structure, it is also not desirable to propose semantic changes to the RTSJ in order to introduce the notion of execution-time server, as was carried out in [110].

Simulating hierarchical scheduling can be achieved by assigning the PGPs of components a logical priority. This priority is then used to influence the priority of the PGP's Schedulables such that the Schedulables belonging to a component with a high priority PGP execute in preference to the threads belonging to a component with a low priority PGP. This follows the semantics of servers in the sense that it appears as though a global scheduler scheduled the highest priority PGP and a local scheduler scheduled the highest priority Schedulable. After its threads have been scheduled or the capacity is exhausted, the next highest priority PGP is scheduled. Of course, in actual fact, there is only one scheduler with the Schedulables being scheduled directly and which is not capable of scheduling PGPs. It is the priority mapping being responsible for this simulated PGP scheduling. The priority mapping is discussed in further detail in Chapter 5 as it is closely related with admission control.

The disadvantage of simulated hierarchical scheduling is that there can be no mixing of Schedulable priorities between PGPs i.e. the priorities of Schedulables in one component's PGP must all be higher than or lower than the priorities of Schedulables in another component's PGP in order to reflect the logical priority of their respective PGPs (servers). Allowing otherwise would violate the behaviour of hierarchical scheduling as it would means that the Schedulables executing under a lower priority server may execute in preference to Schedulables executing under a higher priority server. As a result, any

schedulability analysis would be invalidated because the priority ordering (e.g. Rate Monotonic) at the server level would not be adhered to.

In summary then, PGP with cost monitoring and an overrun handler, along with a replenishment handler to raise the PGP's threads' priorities to their original values at the start of the next replenishment period after an overrun, will give temporal isolation through execution-time server semantics in RT-OSGi. This will prevent the threads in one component (executing under one PGP) from overrunning their CPU budget and affecting the timing constraints of threads executing in other components (running under other PGPs). The temporal isolation model for RT-OSGi is shown in Figure 4.5.

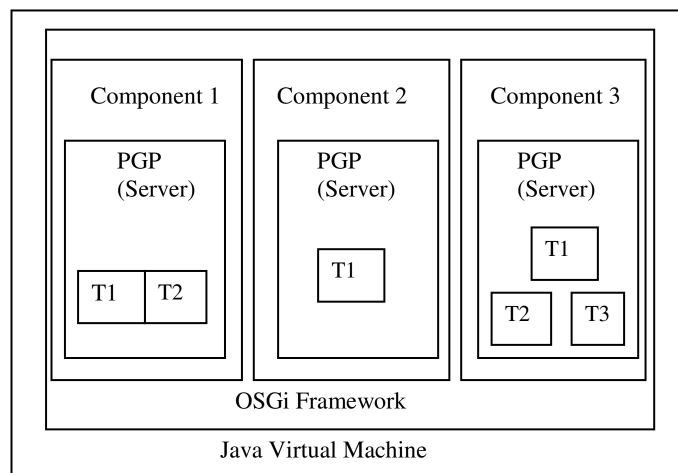

Figure **4**.5 Basic Temporal Isolation Scheme of RT-OSGi.

4.2 Worst-Case Execution-Time (WCET) Analysis

In this section, Worst Case Execution Time (WCET) analysis is introduced. The problems of performing this analysis in the context of OSGi applications are discussed, and the section is concluded by providing solutions to these problems.

4.2.1 Introduction

WCET analysis is the process of computing (upper) bounds for the execution-time of the tasks in the system [111]. When calculating the WCET, it is assumed

that any resources required are available and that the task is not interrupted i.e. blocking and pre-emption is not taken into account during WCET calculation.

WCET analysis is a fundamental concept in real-time systems, without the knowledge of the WCET of each task, it is impossible to build a predictable system which is not completely inefficient with resources. More specifically, the WCET of threads is required as input to the process of schedulability analysis, which is used in order to verify whether the deadline of threads will be met. Schedulability analysis is discussed in Chapter 5. Moreover, WCET analysis is particularly important in RT-OSGi since it provides temporal isolation. If threads' WCET are underestimated, the threads may consume their server capacity before completing their computation and thus they will miss their deadline[4].

Although the importance of WCET calculation is clear, unfortunately, it is not possible, in general, to obtain the WCET of a task. Otherwise, one could solve the halting problem. Since it is difficult to calculate the WCET exactly, real-time systems use a restricted form of programming, which guarantees that the task always terminates. For example, recursion is bounded and iterations are bounded. By restricting real-time system development in this way, it is possible to obtain an upper bound on the WCET.

There are two principle ways of obtaining the WCET of a task, execution-time measurement [112] (often called dynamic timing analysis) and static analysis of the task. In industry, the most common method of WCET estimation is using execution-time measurements, which involves measuring the end-to-end time of each task for a subset of all the possible executions—test cases by executing each task on the target hardware or simulator. This determines the maximal observed execution time. However, since only a subset of all possible executions are observed and measured, this approach will generally underestimate the WCET as it may be possible that the execution path which generates the WCET of a thread

[4] The reason why this is not guaranteed to happen is because often the server parameter selection algorithm (discussed in Chapter 5) over allocates the CPU to a server such that the slack may be sufficient to compensate for the underestimation of threads' WCET.

has not been tested. As a result, a measurement-based approach is not safe for hard real-time systems.

In contrast, static analysis [111] emphasize safety by producing bounds on the execution time, guaranteeing that the execution time will not exceed these bounds. The bounds allow safe schedulability analysis of hard real-time systems.

In static analysis, rather than relying on execution-time measurements, two types of analysis are performed: high level analysis and low level analysis. The major aim of the high-level analysis is to analyse possible control-flows from the source program, without regard to the time for each atomic unit, which is also known as a basic block of flow. This level is only concerned with the programming language issues rather than low-level issues, such as hardware architectures and operating systems. With low level analysis, the focus is on determining the execution time of the basic blocks on a model of the target hardware architecture. This low level analysis is therefore mainly concerned with the processor architectural issues, such as instruction cache, data cache, multilevel cache, pipelining and branch predictions etc. The end result of static analysis is an upper bound on the WCET [113]. As a note, using static analysis means that the upper bound may be very pessimistic because the abstractions used in static analysis lose timing information, thus the CPU may be underutilised.

In addition to the WCET of a task, for schedulability analysis, it is also necessary to find an upper bound for the delays caused by the administrative services of the operating system, the worst case administrative overhead (WCAO). The WCAO includes all administrative delays that affect an application task but are outside the direct control of the task (e.g. those caused by context switches, scheduling, cache reloading because of task pre-emption by interrupts or blocking, and direct memory access) [4].

Whilst it is clear just how important the WCET of a task is in RTS, unfortunately, there are a number of factors which make calculating an accurate WCET very difficult. One such factor is that the architecture of computer hardware is

becoming more complex with features such as caches, pipelines, branch prediction, and out-of-order execution. The reason is that these features increase the speed of execution on average, but also make timing behaviour much harder to predict. As an example, dynamic branch prediction mechanisms try to predict which way a particular branch instruction will go long before the branch has actually been resolved in the processor pipeline. The processor will then speculatively fetch instructions along the predicted path. The problem arises when the hardware makes an incorrect branch prediction and thus time spent fetching instructions was in vain. Solutions to these general WCET issues are out of the scope of this thesis.

4.2.2 WCET Issues

Various features of OSGi make it difficult to calculate the WCET of a thread. In this section, each of these features is introduced and its effect on the calculation of a thread's WCET is explained. In all of the features discussed below, the common problem is that they are all called synchronously, yet their WCET are unknown. This means that the WCET of the calling thread is unknown. As discussed, without knowing the WCET of all of a component's threads, it is impossible during admission control to guarantee a component's threads sufficient resources to meet their deadlines.

4.2.2.1 Service Execution

As discussed in Chapter 2 and Chapter 3, the OSGi Framework draws from both CBSE and SOA, therefore communication amongst components can take place using both the traditional CBSE approach and the service-oriented approach. In CBSE, components typically communicate by specifying the functionality it requires from other components, and also the functionality that it is able to provide to other components. In the OSGi Framework this is achieved by specifying the importing of Java packages from other components, and by specifying the exporting of Java packages to other components. This is a useful way of sharing large numbers of Java classes between components. Since

importing Java packages from other components requires that those packages and thus their encapsulating components to be available to the importing component at compile-time, it is simple to calculate the WCET of the threads which use the imported packages because the code is available for execution measurements or perhaps (if the source code is contained within the component) static analysis.

While having the code to be executed pre-runtime is beneficial for analysis purposes (as in the above CBSE case), it also leads to very static systems in the sense that the component is tied to a particular implementation of the operations it requires. The service-oriented approach to communication is much more dynamic and flexible.

To support service-orientation, the OSGi Framework provides an intra-JVM (within a single JVM) service model. This means that service requesters and service providers are both deployed within the same JVM. This differs from the inter-JVM service models that are more typical of SOA such as ServiceDDS [114]. In these other service models, service providers and requesters are typically deployed across a network. Figure 4.6 and Figure 4.7 show the relationship between service providers and JVMs.

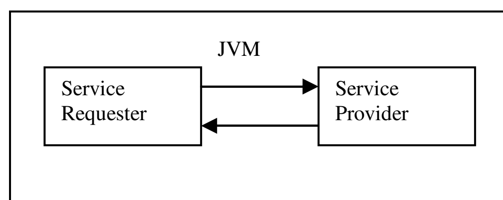

Figure 4.6 Intra-JVM Service Model:

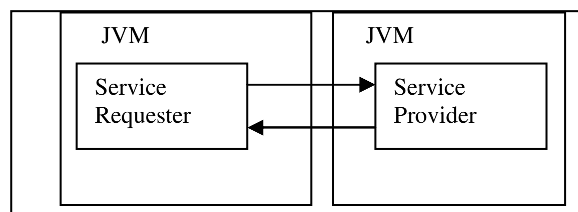

Figure 4.7 Inter-JVM Service Model:

The intra-JVM service model in OSGi allows components to be connected together by a publish-find-bind model for Plain Old Java Objects (POJOs). In the intra-JVM service model, if a thread calls a service, the calling thread executes the service as a synchronous local method call; there is no separate service providing thread. As a result of this, a thread's WCET is not only the time it spends executing its own code, but also the time it spends executing code of services.

In this model, a component may provide a service to other components by registering an object (implementing an interface) in the service registry. Any service-requesting components need only be aware of (compile with) the interface. In this way components are loosely coupled, service-requesting threads can lookup implementations of a service interface in the service registry and invoke their methods, but are not tightly coupled with a particular service implementation.

The issue of WCET analysis in the OSGi Framework's intra-JVM service model is complicated by the loose coupling between service-requesting and service-providing components, which gives two forms of dynamism. Firstly, service requesters dynamically discover service implementations. Secondly, the service implementations are dynamically available meaning that at different points in time, different implementations of a service interface will be available in the service registry. Dynamic discovery and availability of services allows for service implementation substitution, that is, a service requester may use one implementation of a service, then discover it is no longer available, and finally obtain a different implementation of a service, possibly provided by a different component.

As a result of the dynamic discovery aspect of service-orientation, the service implementation that a service requester binds with is unknown offline. All that a service requester compiles to offline is a service interface, which is not executable and cannot be used in a measurement-based approach to WCET calculation. This means that, until service bind-time, the WCET of a thread is

unknown. Furthermore, although the dynamic availability of services i.e. the ability of service requesters to bind with different service implementations is quite flexible and useful for fault tolerance, it is problematic for calculating a thread's WCET in real-time systems. Different service implementations are likely to have different WCETs, the service requesting thread's WCET will therefore be affected by the WCET of the service implementation it is invoking. An example of the effects of dynamic availability on the WCET of a service calling thread can be seen with a "sort" service, which sorts a collection of numbers into ascending order. A service requester performs WCET analysis and includes the cost of calling a Mergesort service implementation. However during run-time, the Mergesort service implementation goes offline and the service requester then binds with a Bubblesort service implementation. Clearly, the WCET of the service calling thread will increase because Mergesort typically outperforms Bubblesort.

As a note, services which create their own threads are not problematic since these threads have resources guaranteed for them by the admission control of their component since threads, unlike services, are active entities and not passive.

Closely related with the issue of WCET calculation in the OSGi's service model is the issue of malicious services and their impact on other threads deployed in the OSGi Framework. This thesis does not address the issue of malicious services directly, as it is assumed that service requesters only bind with service providers with whom they have a service level agreement and are therefore deemed trustworthy. In any case, the temporal isolation and memory management (discussed in Chapter 6) of RT-OSG will provide a means of damage limitation against malicious services. The temporal isolation ensures that if a service requesting thread is hijacked by a malicious service by having the thread execute an infinite loop, the temporal isolation will ensure that in the worst case i.e. when the service requesting thread is the highest priority within its PGP, only the threads within the calling thread's component's PGP will be starved of the CPU.

4.2.2.2 Synchronous Event Handling

The OSGi Framework is a very dynamic environment with services being registered, unregistered, and components being installed, uninstalled, and updated. For example service providers may, amongst other things, unregister services. However as the service object is still valid in the JVM, it is still potentially usable. Despite this, the service may behave unexpectedly. For example, a print service may be unregistered when the physical printing device attached to the OSGi host is out of paper. Therefore, it is imperative that service requesters are notified that the service is no longer suitable for invocation to prevent erroneous results from service invocation.

In order for the threads within a component to keep track of the current state of services and components, and to react quickly to the changing environment, the OSGi Framework provides synchronous event handling through a publish subscribe [115] mechanism. In this model, when a thread is interested in subscribing to events, it passes a listener object to the Framework. When any thread registers, modifies the properties of, or unregisters a service, this event-publishing thread must synchronously execute the service listener objects of any subscribers to the event. Similarly when life cycle operations are invoked on components, the invoking thread must synchronously call any registered synchronous component event handlers. This synchronous event handling model may drastically affect the WCET of any event-publishing thread. The effect on the publishing thread's WCET will depend both on the total number of subscribers and on the subscribers' listener's event handling method. Clearly, even if the WCET of each subscriber's event handling listener method is small, if the number of subscribers is large, the event- publishing thread's WCET will still be greatly affected.

4.2.2.3 Service Factories

When registering a service, the service provider registers an implementation of a service interface in the service registry. However, all service requesters obtain a reference to the same service object and such sharing of service objects may be

undesirable in some applications. Instead, it is possible for service providers to register a service factory object in the service registry rather than registering a service object directly. When service requesters attempt to bind to a service which has a service factory registered, the service requesting thread must synchronously call a method in the service factory provided by the service provider. The service factory will typically create a service object specifically for the service requester and may perform some other initialisation tasks before returning the newly created service object to the service requester. Similarly when a service requester wishes to release the service, a service factory method is synchronously called by the service requester, allowing the service provider to customise the release process.

An example of the use of service factories is providing a mechanism for saving and loading service state. As service availability is dynamic, this means that a service may become unregistered while a service requesting thread is executing a service method. Upon being notified of this, the service requester can release their use of the service. This will call the service factory which can then save service state on behalf of the service requester. When the service becomes available again, the service factory can be called to load the service state thus allowing the service requester to resume using the service.

Clearly, the use of service factories affects the WCET of the service requester

4.2.2.4 Component Activation and Deactivation

Components in the OSGi Framework may be broadly classified as passive, and active. Passive components do not contain any threads, but typically provide service objects, for other components to invoke, and other resources such as Java packages, and HTML files etc. Active components create and start threads of their own and may well also provide resources and services to other components. In order to publish services and/or create threads, active and passive components require an initialisation phase to perform activities such as creating threads, registering services and creating network sockets etc. This initialisation phase is

known as component activation. Similarly, when a component is to be stopped, both active and passive components often require a cleanup phase e.g. removing services, and stopping threads etc. This is called component deactivation.

The thread which calls the life-cycle operations of start and stop must synchronously execute the components activation and deactivation code respectively.

4.2.3 Solving the WCET Problems of OSGi

Before discussing solutions to the WCET calculation problems discussed in Section 4.2.2, the general approach used by RT-OSGi application developers to calculate the WCET of threads is discussed.

To calculate the WCET of threads within a component, it is proposed that a measurement based approach to WCET calculation be used during the last phase of system testing. The WCET calculation is performed by profiling code. The general idea is to use execution-time profiling during execution of each test case, where each test case should execute a path through each thread's run method/ asynchronous event handler's handleAsyncEvent method. A simple approach to measuring execution-time is as follows. When a thread starts execution (either for the first time, or on subsequent releases), it writes its ID and the current time to a list. Similarly, when the thread blocks for its next period, it writes its ID and the current time to a list. In the case of asynchronous event handlers, the time and handler's ID can be written at the beginning and end of the call to the handleAsyncEvent method.

In between the two entries in the list for a thread/handler will be the entries from each release of higher priority threads which have pre-empted the initial thread. The execution time of a thread in the list can then be calculated by taking the difference between the time recorded at the end of the thread's execution, minus the time recorded at the start of its execution minus the execution-time used by

all the releases of higher priority threads that have occurred during this time i.e. all of the execution-times recorded in the list between the start and end time stamp elements in the list of the thread in question. In this way, execution time calculation takes thread pre-emption into account by removing the execution time incurred by the (possibly multiple) releases of higher priority threads.

The thread's execution time along with a unique ID for the thread is written to an output file. For each test case (path through the thread) the execution time should only be written if it is greater than any previous value stored for that thread. In this way after the thread has had all paths executed, the WCET during a path will be the only value recorded for a thread.

As a note, this simple approach to measuring the execution-time of a thread may be very pessimistic. Any blocking on a thread (such as the execution of code that is synchronized) or any self-suspension (such as the execution of the sleep and wait methods) will be accounted to the thread's execution-time. This makes the measurement process potentially very pessimistic, which is clearly undesirable. The issue of self-suspending code can be solved by having the classes used in RT-OSGi application development (such as OSGiRTT and OSGiAEH) provide wrapper methods for the (final) wait and notify methods, and the (static) sleep method. The wrapper methods can be used to record the time that the calling thread called and returned from the self-blocking methods. This is adequate to take self-blocking into account during measurement-based WCET analysis. Unfortunately, the more general case of blocking through the use of synchronized blocks and methods cannot easily be taken into account during such a simple approach to measurement-based WCET calculation.

Many more accurate approaches to measuring the execution-time of Java threads exist (such as the Java Virtual Machine Profiler Interface (JVMPI) [116] and the Eclipse Integrated Development Environment[84]), since the technique of measuring the WCET of application threads is well known in the literature. RT-OSGi developers are therefore able to utilise such tools as alternatives to using the aforementioned simple approach to WCET measurement, and should

therefore be able to avoid the impact of blocking on the measured execution-times

Regardless of the approach used to measure the WCET of threads, a number of features are proposed which can be integrated with a measurement approach to WCET in order to solve the WCET issues discussed in Section 4.2.2. These proposed features are discussed in the following subsections.

4.2.3.1 WCET Contracts

Although the aforementioned measurement-based approach to calculating the WCET of threads in RT-OSGi is adequate when a thread doesn't use services, as discussed in Section 4.2.2.1, the dynamism of services means that services are not available pre-deployment time (and their implementation may change at run-time in any case). Therefore, it is impossible to simply measure the execution-time of service implementations.

In order to calculate the WCET of a service requester's use of services, it is proposed that services have execution-time contracts, similar to the quality of service contracts specified in [46], that is, the service requester annotates their copy of any service interfaces with an acceptable bound on the amount of execution-time that each service method of an implementation of a service may cause the calling thread to incur. This essentially gives each service method an execution-time budget which service implementations must abide by in order to make them compatible for use with the service requester. The sum of the execution-time budgets of service calls and the calling thread's own execution-time is enforced at the thread's priority level. However, as mentioned, the thread may use more than this combined execution-time budget at a background/non-real-time priority level.

The result of such execution-time contracts is that that service requesters will know pre-runtime the WCET of any service methods they invoke, but will still have the flexibility with binding with different service implementations provided

that the service providers abide by the service contracts. This allows service requesters to still benefit from the dynamic discovery of service implementations, a key theme in SOA. Note that the service contracts are only applicable to intra-JVM service models. In the case of remote services, the services' execution-time would not be accountable to the calling thread and thus such contracts would not be required.

The service WCET annotations are in the style of Java annotations. This enables either the Annotation Processing Tool (APT) or the more recent Java compilers to read the service interface WCET annotations and perform some user-defined actions. When a service interface annotation is read by the APT or compiler, a stub (or dummy) service implementation class should be created. For each service method annotation of the service interface, implementation methods should be generated in the corresponding stub implementation class. The stub service implementation methods should obtain a reference to a calling thread and modify a field in the thread which is used to keep track of the WCET it would have incurred had it actually executed a service implementation with the WCET specified in the contract. In this way, the stub service implementations generated as part of annotation processing simulates a service implementation with the worst case execution-time specified in the service contract and it can be used as part of measurement-based WCET analysis to include the WCET of calls to any required services. The final WCET of the service requesting thread can then be set to equal the cost measured from executing the thread's own code plus the value stored in the service cost field which mimics the execution of service implementations with the worst case execution time specified in the associated service contract. As a note, the WCET of a service method call is set by calling a method in the calling thread, passing the WCET as a parameter. This is shown in Figure 4.9. The WCET is not set by using the return statement of methods and so it does not interfere with the application logic.

An example of a service execution-time contract and the corresponding stub service implementation class are shown in Figure 4.8 and Figure 4.9 respectively. As a side note, such stub service implementations are required by service

requesters in service-oriented systems regardless of whether they are used in conjunction with execution-time contracts and WCET calculation. The reason for this is because service requesters will need to test their code before it is deployed and before it is able to utilise third party service implementations.

```
public interface PrinterService
{
      @WorstCaseExecutionTime(100)
      public void print();
}
```

Figure 4.8 Service Execution-Time Contract

```
public class Printer implements PrinterService
{
      public void print()
      {
            ((OSGiRTT)RealtimeThread.currentRealtimeThread())
                  .addTime(100);
      }
}
```

Figure 4.9 A Stub Service Implementation

As a note, if a service does not cause the calling thread any execution-time overhead e.g. because it starts it own threads or because it fires events, the contract should have a zero execution-time cost and the dummy service implementation should convey this information during execution-time profiling The execution-time of the threads that are started by the service will be accounted for during execution-time profiling of those threads. The same applies for any event handlers that are released as a result of an event firing associated with service execution.

Although the use of service contracts and stub service implementations during a thread's WCET analysis still allows service requesting threads flexibility with regard to binding to service implementations at run-time, it is imperative that the service implementations that a service requester binds with at run-time abide by the service contract used as part of WCET analysis. The service requester can check this by using the service contract as part of service discovery at run-time. Service providers register WCET information along with their service implementation, and before a service requester obtains a reference to a service

implementation, contract matching takes place in order to ensure that the WCET of the service implementation is less than or equal to that used as part of a service requester's WCET analysis.

Finally, in addition to the use of contracts for accounting for the WCET of service methods, the same mechanism can be used for solving the issues of service factories and component activation/deactivation.

The OSGi Alliance recommends that service factories and component activation/deactivation should incur very little execution-time overhead on the calling thread. If a service factory or component activator needs to perform extensive computation e.g. if a service factory must save a large amount of service state or if a component activator must perform computationally intensive component initialisation, such computations should be performed by spawning new threads dedicated to these tasks such that the calling thread's WCET is not severely affected. Therefore, execution-time contracts can be used to place a bound on the execution-time of component activation/deactivation and service factories such that the execution-time overhead on the synchronously calling thread is known pre-runtime. The calling thread's WCET can therefore also be measured pre-runtime.

4.2.3.2 Adaptive Resource Reservation

A possible alternative to using WCET contracts in OSGi applications with very soft timing requirements is to use adaptive resource reservation. This subsection discusses adaptive resource reservation but only as a theoretical alternative to WCET contracts. It is important to stress that such an adaptive approach has not been integrated with the OSGi Framework in this thesis.

The general idea with the adaptive approach is that, unlike execution-time contracts which include the WCET of service invocations while still permitting service implementation substitutability, no attempt is made pre-deployment time

to include the cost of executing service methods. Instead, the resource reservation for a component is based on the WCET of its threads excluding their use of services. Rather than directly taking the execution-time overheads of service method and service factory method invocations into account, it is assumed that the resource over-allocation from the various pessimistic assumption made in the design of RT-OSGi will compensate.

These pessimistic assumptions are:

1. WCET – threads will often execute much faster than their WCET
2. Event Firing – events are typically not fired as often as the worst case
3. Offline Server Parameter Generation Algorithms – typically significantly over allocate the CPU to each server

Relying on the aforementioned pessimism of hard real-time system assumptions alone is not enough. If a component's threads use services extensively, then the over allocation associated with the pessimistic real-time assumptions may be insufficient and threads may begin to miss deadlines. To minimise the effects of this problem, each thread's performance (in terms of deadline misses) is monitored at run-time using the RTSJ's deadline miss handlers. Each thread has associated with it a value that indicates what it considered to be a reasonable number of missed deadlines within a specified period of time. The system keeps a count of how many deadlines are missed within this time frame. If the execution time associated with service invocations is less than the over allocation of execution-time through real-time pessimistic assumptions, then deadlines will still be met and the server parameters are adequate. If on the other hand there are insufficient resources available from the pessimism of real-time assumptions to compensate for the execution-execution-time incurred as the result of service method invocations, the service requesting thread will miss deadlines. If the thread misses a greater number of deadlines than it deems reasonable per time period, then adaptive resource reservation will occur i.e. the server parameters will be adjusted in order to try and reduce the number of deadlines missed. Figure 4.10 shows the general mechanism of resource adaptation.

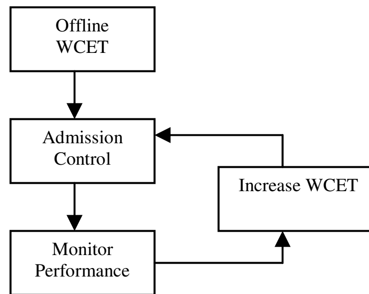

Figure **4**.10 Resource Adaptation

In Figure 4.10 it can be seen that WCET analysis occurs offline without taking service execution into consideration. As part of admission control, server parameters are generated which essentially give a component's threads a resource reservation (this is discussed further in Chapter 5). This resource reservation does not include resources for service execution.

As discussed, at run-time, the performance of the thread in terms of the number of deadline misses it experiences in a period of time is recorded. When the thread deems to be performing inadequately, its resource reservation is increased The adaptive resource reservation is essentially server parameter selection followed by a schedulability test in order to check whether it is possible to adapt the amount of resources allocated to the component.

As discussed at the beginning of this subsection, this adaptive resource reservation is currently not a part of RT-OSGi. It serves more as a possibility for future work.

4.2.3.3 Asynchronous Event Handling

To solve the issues associated with synchronous event handling in the OSGi Framework, the RTSJ's asynchronous event handlers (AEH) model is applied to RT-OSGi. As discussed, currently, threads interested in service and component event handling call the following two methods respectively: "add Service Listener (ServiceListener listener)", and "add Bundle Listener ((Synchronous) Bundle Listener listener)". These methods essentially add the listener object

parameters passed to a list. When a service is registered, unregistered or modified, a service event is fired and the thread which caused the event to fire iterates through the list of listeners and executes their handling code. Likewise, when a component has a life cycle operation performed, the life cycle operation invoking thread must iterate through the list of component (bundle) listeners are execute their handling code.

For RT-OSGi, it is proposed that every component and every service interface have an asynchronous event associated with it. The following two methods should then be used by component developers in place of the standard OSGi Framework synchronous event handling methods in order to subscribe to service and bundle events: addAsyncServiceHandler(String serviceInterfaceName, AsynchronousEventHandler aeh), and addAsyncBundleHandler(Bundle bundle). When a thread calls addAsyncServiceHandler(…), the method will check to see if an asynchronous event for the service interface named as a parameter has already been created, if so, the method adds the asynchronous event handler passed as a parameter to the existing service interface event. If not it creates the event and adds the handler to the event. In this way, it is possible to add a handler for a service before an implementation of the service of interest has been registered and so for example it is possible to be notified of the very first service implementation to be registered. Calls to register, unregister and modify services will fire the asynchronous event corresponding to the service, passing a parameter such as a string indicating what is happening with the service e.g. the service is being unregistered, and perhaps also passing a properties file containing service attributes. Any asynchronous event handlers associated with the asynchronous service event will then be released with whatever real-time parameters they were configured with. Note, events are associated with the service interface and not a particular service implementation. Therefore registration of multiple service implementations of the same service interface will not cause multiple events to be created but instead will fire the event associated with the service interface multiple times. Likewise unregistration of a service implementation will not cause the removal of an event but will simply fire the associated event. For bundle event handling, when a thread calls `addAsyncBundleHandler(Bundle bundle)` an event will be created for the

component passed as a parameter. Calls to any life cycle operations on this component will then fire the asynchronous event, passing a parameter informing the handler what life cycle operation was invoked on the component of interest.

This asynchronous event model is advantageous as the event firing is decoupled from the event handling and so the WCET of the thread firing the event is unaffected by the event handling. In terms of the asynchronous event handlers, these need to be sporadic with a minimum inter-arrival time (MIT), and whatever queue violation policy is necessary. This ensures that the time spent processing aperiodic events is bounded and does not consume all of the resources allocated to a component. The MIT value will depend on the application, if the event handling response time is to be minimised, the MIT must be small. A longer MIT will lead to events being queued and poorer event handling response times but more resources will be available to other threads within the component.

Finally, in order to support asynchronous event handling in RT-OSGi, it must be possible to pass parameters with asynchronous events so as to identify the service/bundle of interest. Therefore since the current version of the RTSJ does not support this, such parameter must be implemented until the next version of the RTSJ (RTSJ 1.1) is released which does support parameterised asynchronous events.

4.3 Summary

In the standard OSGi Framework, threads have unrestricted use of the CPU which means that it is impossible to guarantee the timing requirements of real-time threads. Execution-time servers solve this issue by providing temporal isolation between threads in separate components in the application without imposing a strict cyclic scheduling regime and without affecting the ability to develop dynamically reconfigurable real-time applications. To support execution-time servers, cost enforcement and simulated hierarchical scheduling extensions are proposed to the OSGi Framework.

The standard OSGi Framework also hinders the development of real-time applications by increasing the difficulty of Worst Case Execution Time (WCET) analysis. This analysis is essential for guaranteeing the timing requirements of real-time threads. This issue is addressed by integrating WCET contracts with a measurement-based approach to WCET analysis in order to address the specific issue of application component developers not having third party code (such as service and service factory implementations) available at the time of performing WCET analysis. Furthermore, the OSGi Framework itself is extended with asynchronous event handling in order to solve the specific issue of the current synchronous event handling mechanism. Finally, the OSGi Framework is also extended by bounding the amount of time for the component activation and deactivation processes.

5

Admission Control

5.1 Introduction

As discussed in Chapter 4, in order to provide components with a CPU resource reservation it is necessary to enforce computation-time budgets for threads in currently deployed components, and it is also necessary to control the load on the CPU in terms of the number of components to be installed/updated. The latter topic is discussed in this chapter.

The OSGi Framework offers life-cycle operations (discussed in Chapter 3) to component developers enabling components to be installed, updated, uninstalled, started, and stopped during run-time.

The install and update operations may increase the load on the CPU by essentially permitting more threads to be deployed, furthermore, life cycle operations can occur at any time. This means that the CPU can be overloaded by the repeated use of install/update operations and thus threads may miss their deadlines. In order to prevent this situation, RT-OSGi enables components to have resource reservations. To support such reservations, in addition to the temporal isolation discussed in Chapter 4, a means of controlling the CPU load is also required. The contribution of this chapter is to discuss the necessary extensions to the OSGi life cycle operations to support real-time OSGi applications.

One means of controlling the system load is to use adaptable Quality of Service (QoS) components. This refers to the ability of components/services to adapt in run-time the quality exhibited [117]. Such adaptability can be used to accommodate dynamic reconfiguration in the OSGi Framework. For example, Lima et al [118] propose that a real-time application should be structured as a set of multiversion tasks, with each task version offering a different level of quality of service. During the installation of a new component, rather than preventing overload situations by rejecting the dynamic reconfiguration, the goal is instead is to select the version of tasks associated with a lower quality of service thus reducing the CPU requirements of each task and the application as a whole. The idea being that application utility is greater than it would be by either rejecting the dynamic reconfiguration or by accepting it and scheduling tasks with higher levels of quality of service which always fail to meet their deadlines.

Similar to multiversion tasks, adaptive QoS to support dynamic reconfiguration can also be achieved by designing tasks to use imprecise computation algorithms, this approach was taken in [118]. Imprecise computation algorithms [119] are algorithms whereby the quality of the result produced by the algorithm varies depending on the amount of resources (such as CPU time) allocated to executing the algorithm. The more CPU-time allocated, the more time the algorithm can spend on iteratively improving the quality of the computed result. One example of an iterative algorithm is finding successively better approximations to the roots (or zeroes) of a real-valued function. Such an algorithm can be used by tasks to reduce the execution-time requirements of the application after dynamic reconfiguration, much like the aforementioned multi-version tasks.

However, using adaptive QoS has a number of issues, especially when used in conjunction with the OSGi Framework. Despite the fact that adaptive QoS potentially allows more components to be deployed, the utility of each component will be reduced i.e. the application essentially undergoes graceful degradation. For many applications, this is not tolerable. Moreover, multiversion tasks and iterative algorithms have limited applicability, and even when they are applicable, these approaches also complicate the software development process even with the support of tools such as [120] . Finally, since the RT-OSGi

Framework may be hosting multiple independent applications, it is unfair to have unrelated applications affecting each others utility by forcing currently deployed components to switch to a lower QoS version in order to allow for the deployment of a new component.

As a result of the issues with adaptive QoS, RT-OSGi instead makes use of admission control in order to help ensure that components' threads meet their timing requirements. Admission control is a mechanism which attempts to control the load on the processor by using acceptance tests to filter requests for deployment; only entities passing the acceptance tests may be deployed. The result of applying admission control to the install and update OSGi life cycle operations is that these operations fail when they would result in the system becoming unschedulable (i.e. the worst-case response time of at least one thread is greater than its deadline). Note that there is no reason why components undergoing admission control may not be multi-versioned. The important point is that the component itself should have control over its version/QoS and not the deployment environment, as is the case in the aforementioned related work.

In this chapter, the admission control extensions for the install and update life cycle operations are discussed further. Furthermore, in addition to the admission control required for component installation and update, the process of starting and removing components is also discussed in this chapter. Although the processes of starting and removing components do not require admission control, they nevertheless need extending for RT-OSGi and hence the extensions to these life cycle operations are discussed in this chapter.

5.2 Component-Derived and User-Derived Life Cycle Operations

The life cycle operations can have two sources, life cycle operations can be invoked from the code (i.e. programmatically) of components, or, a user can invoke life cycle operations through the use of an interactive user interface to the OSGi Framework (so as to perform corrective, perfective, or adaptive

maintenance). In order to distinguish between these two types of life cycle operations in this thesis, the former is named component-derived life cycle operations, and the latter, user-derived life cycle operations.

The dynamic reconfiguration of the application that occurs as a consequence of component-derived life cycle operations is known before the component is deployed. This is a key point, and is particularly important for the install and update life cycle operations, which make use of the fact that the dynamic reconfiguration of an application through component-derived life cycle operations is pre-planned. A more flexible approach to dynamic reconfiguration is the user-derived life cycle operations which are discussed below.

In standard OSGi, user-derived life cycle operations are issued via an implementation dependent means; for example one implementation of the OSGi specification may use a Web Browser and a Web Server hosted in an OSGi component in order to receive user-derived life cycle operations, whilst another implementation may provide a user interface through the command-line. In both cases, the life cycle operation invocation will take place in a standard i.e. non-real-time Java thread. In RT-OSGi, such a model would provide poor performance in terms of the worst-case response time (i.e. the longest time from the task becoming ready to execute to it completing execution) of user-derived life cycle operation requests, and as a result, the time taken to perform the associated dynamic reconfiguration may be quite long. Therefore, in RT-OSGi, the user-derived life cycle operations are processed by a real-time thread executing under an execution-time server. The life cycle server has a server capacity, replenishment period and deadline and has cost enforcement just like any other application server. The server parameters are application dependent and can be configured by the user; of course, whatever the server parameters are configured to be, they will be included in application schedulability analysis. Finally, as will become apparent, the install life cycle operation cannot be performed with real-time constraints, and therefore the life cycle processing server does not offer real-time constraints on the time it takes to complete a user-derived life cycle operation. This should not be an issue since user-derive life cycle operations are invoked by the user not by the application.

5.3 Installing Components

For admission control, it is necessary to know: what the resource requirements of a component are, and, whether the system has sufficient resources to meet the components' resource demands. The CPU resource requirements of a component can be gathered by using a Server Parameter Selection algorithm [121]. Checking whether there are sufficient CPU resources available can be achieved by using schedulability analysis [122]. Finally, priority range assignment must take place to ensure that the priority ordering assumed in the schedulability analysis is still valid after application reconfiguration. Each of these three features is discussed in detail below. After performing these CPU admission control related procedures, it is necessary to perform GC reconfiguration analysis, memory admission control, and finally reconfiguration of the GC. These processes are discussed in Chapter 6.

Note that because a component may need to install other components as part of its threads' execution, admission control is carried out for the entire group of components rather than in isolation i.e. during the installation of the first component of a group of components, the admission control guarantees resources for all other components in the group, i.e. if a thread in Component A needs to install Component B, and one of its threads needs to install Component C, then the installation of Component A will involve performing admission control for all three components. In this way, Component A will only be installed and deployed if resources can be guaranteed for Component B and Component C. Essentially, RT-OSGi applications have a non-real-time initialisation phase during which, admission control for all of the components of an application that are installed takes place.

Having a non-real-time initialisation phase for carrying out the admission control for the install operation (i.e. reserving resources in advance of performing the install operation) is necessary because there is little benefit in reserving resources for a component which must function as part of a group, When a component is installed, either, resources should be guaranteed for the whole group, or, installation should fail. An additional benefit of such upfront admission control

during an application initialisation phase is that, after performing the admission control, component installation can take place in the application real-time phase without admission control and thus the execution-time of the install operation will have a much shorter and deterministic execution-time. The result of this is that it is possible for component installation to be carried out with real-time constraints e.g. it is possible for a component to install another component within a deadline since the admission control for the operation would have been performed during the initialisation phase of the application.

5.3.1 Execution-Time Server Parameter Selection

As discussed in Chapter 4, temporal isolation (partitioning) amongst components is provided by RT-OSGi by using execution-time servers. Execution-time servers, as discussed, are implemented using ProcessingGroupParameters). This provides a way of managing the processing time assigned to each component. Each component has a CPU budget per period in which to execute the component's threads.

When a component undergoes installation in RT-OSGi, an execution-time server (ProcessingGroupParameters) is created for it. This server must then be assigned parameters (computation-time/capacity (C), replenishment period (T), and deadline (D)) such that all of the component's threads executing under the server have sufficient CPU time to meet their deadlines. However, at the same time, it is important that the parameters assigned to the server are not too pessimistic. The issue with over allocating the CPU to each server is that it may result in components failing admission control on schedulability grounds, which will reduce the total number of components that can be deployed. The reason for this is that it appears during admission control that the system is heavily loaded, when in fact, most of the CPU time assigned to components' servers is unnecessary for making their threads schedulable.

In terms of the server parameter selection algorithms, they can be classified as either offline [121, 123, 124] or online [125] algorithms. There is a trade-off

between the degree of pessimism of server parameters generated, and the execution-time of the algorithm. The online algorithms have smaller execution-times than the offline algorithms, but as a result, the server parameters generated are much more pessimistic, over-allocating the CPU to the server. Both approaches are applicable to OSGi, the component developer can generate server parameters offline and include the generated parameters in their component's manifest file, alternatively, an online server parameter algorithm can be called from the component install/update life cycle operations as part of admission control.

5.3.2 Schedulability Analysis

After server parameter selection, it is essential to check that the server parameters (CPU resource requirement) of a component can be assigned without causing threads in other components to miss their deadlines, i.e. without making the system unschedulable. Since servers are like periodic threads in the sense that no more than "C" units of computation time can be consumed within a period "T" with cost enforcement, the schedulability analysis used for sporadic task systems is also applicable to systems with fixed priority servers. The exception to this is the Deferrable Server (which requires a different schedulability analysis) as it violates the implicit assumption that a periodic task must execute whenever it is the highest priority task ready to run. The Sporadic Server algorithm also violates this assumption but its replenishment policy compensates for this violation.

Schedulability analysis is used to predict temporal behaviour via tests which determine whether the temporal constraints of tasks will be met at run-time [126]. A schedulability test is said to be sufficient if all task sets passing the test are guaranteed to be schedulable i.e. are guaranteed to meet their deadlines. Task sets failing sufficient schedulability analysis may still be schedulable but this is not guaranteed, that is, failing a sufficient test does not necessarily mean that the task set is not schedulable. An example of such a sufficient schedulability test is the utilisation bound test presented by Liu and Layland [127].

An exact schedulability test is both sufficient and necessary i.e. task sets passing the test are guaranteed to be schedulable and those failing the test are guaranteed not to be schedulable. An example of such an exact test is Response Time Analysis (RTA) [128], which firstly calculates the worst-case response time, that is, the longest time between the arrival of a task and its subsequent completion, and then determines the schedulability of the task by comparing the worst-case response time of the task with its deadline. Clearly, if a task in the task set has a worst-case response time greater than its deadline, the task set is not schedulable.

While an exact schedulability test is desirable, it is computationally expensive and therefore not well suited for use in online systems such as OSGi. However, it is observed that the execution-time of RTA can be significantly reduced by using different initial values for the algorithm such that the RTA algorithm terminates quicker. This approach is taken by Davis et al [129] and is known as Boolean Schedulability Analysis. This variant of RTA is known as Boolean because it can only be used to determine the schedulability of the system, i.e. schedulable or non-schedulable. The response times of tasks cannot be used for anything else as they are pessimistic estimates and not the exact response times. However, since RT-OSGi only requires such a Boolean answer to the question of application schedulability, such a Boolean test is adequate.

To further reduce the time it takes to determine whether the system is schedulable, a sufficient schedulability test known as Response Time Upper Bound (RUB) [130] is used in combination with the Boolean test previously mentioned. The general approach is to use the RUB test on a task by task basis. Tasks failing the RUB test undergo the Boolean test. The combination of these tests acts as an acceptance test for CPU admission control. If a task fails the RUB test it is not a problem since the system may still be schedulable. However, since the Boolean RTA is an exact schedulability test, any task failing this indicates that the system is not schedulable and the component undergoing admission control should be rejected. In this case, it may be desirable to remove one or more currently deployed tasks in order to free up enough resources to allow the component to pass admission control and be successfully installed.

Whether or not this technique is used depends on whether components are considered to have importance levels. If an RT-OSGi application developer assigns importance levels to their components, the RT-OSGi admission controller will then use these levels in order to try and admit new application components at the expense of removing one or more components of a lower importance level.

As a note, although the aforementioned efficient Boolean RTA improves the performance of the standard RTA algorithm, the execution-time of the Boolean RTA still depends on the number of tasks and the temporal parameters of the tasks to be analysed. Therefore, Boolean RTA can not be used in a real-time context because it is not possible to efficiently determine the WCET of the thread calling the algorithm. The result of this is that the admission control process does not have timing constraints.

As a result, it could be argued that the reduction in execution-time of the efficient Boolean RTA offers little advantage over the standard RTA algorithm. However, since RTA is performed online and there is no requirement in RT-OSGi to determine the exact response times of tasks, it is a waste of CPU time to run the standard RTA algorithm when it offers no advantages to RT-OSGi.

5.3.3 Priority Assignment

As discussed throughout this thesis, the OSGi Framework is a very dynamic environment, with application undergoing dynamic reconfiguration through the invocation of life cycle operations. In terms of scheduling application threads, dynamic scheduling is ideally suited. However, the RTSJ only provides a fixed priority pre-emptive scheduler. Perhaps the reason why the RTSJ only mandates that an implementation provide fixed priority pre-emptive scheduling is because fixed priority pre-emptive scheduling offers advantages of flexibility over cyclic approaches whilst being sufficiently simple to implement in comparison to dynamic priority based schedulers. For further discussion about fixed priority vs. dynamic priority scheduling, see [88].

As discussed in Chapter 4, it is not desirable to modify a JVM or OS to provide a dynamic scheduler such as EDF (Earliest Deadline First) [131] because this would constrain the use of RT-OSGi. Instead, RT-OSGi uses the RTSJ's default fixed priority pre-emptive scheduler. As a result, the aforementioned stages of admission control, namely server parameter selection and schedulability analysis therefore assume a priority assignment policy which supports such fixed priority systems. In RT-OSGi, Rate Monotonic (RM) priority assignment [127] is used. In RM, tasks with a higher rate/frequency of execution are assigned higher priorities than tasks with a lower rate/frequency of execution i.e. tasks with smaller periods are assigned higher priorities than tasks with longer periods.

In order to bridge the gap between the dynamic environment of the OSGi Framework and the RTSJ's fixed priority pre-emptive scheduler, RT-OSGi provides a priority mapping scheme. When lifecycle operations are invoked e.g. to install new components or to update currently deployed components so as to change the set of threads deployed or their timing requirements, the priorities of threads may need to be manipulated by RT-OSGi so as to reflect the priority assignment algorithm (such as Rate Monotonic priority ordering [127]) assumed during schedulability analysis. Not doing so would violate the schedulability analysis and render the results of the analysis useless; thus threads may indeed miss deadlines. Furthermore, as discussed in Chapter 4, so as to provide temporal isolation with execution-time servers, hierarchical scheduling is simulated by assigning servers logical priorities and using these logical priorities with RM priority assignment in order to assign the component a range of priorities which can then be assigned to its threads. This is discussed in more detail below.

As part of admission control, the range of priorities that can be used by a component's threads can be calculated by having RT-OSGi maintain a priority ordered list of servers (according to RMA), and when a component is undergoing install admission control, the server is inserted in the correct place in the server list. The priority range assigned to the server's component can then be assigned such that the range is higher than the next lowest priority server in the list but lower than the next highest priority server in the list. This ensures that the

semantics of hierarchical scheduling are respected i.e. that the threads in a higher priority server execute in preference to the threads in a lower priority server.

The priority range assignment works as follows. Each component states in its temporal specification file (RealTimeDefs) the number of required unique priorities, rather than simply the number of threads. The reason for this is that threads within a component may share a priority. If the number of unique priorities required by a component is greater than the number of free priorities, the component must not be installed. This check is carried out before the aforementioned RUB and Boolean RTA schedulability tests as part of the acceptance test.

The first component that passes the admission test will have its priority range assigned based on a set of rules which map the range based on the size of the component's server's period, the component will then be added to the list of currently installed components along with the priority range that it occupies. This list is sorted based on increasing server period/deadline. Table 5.1 shows an example of the priority mapping rules.

Period (ms)	Priority Range (x = num of supported priorities / 4)	Example (Min = 0, Max = 27, X = 7)
<= 1500	min + 3x, max	21-27
> 1500 & <= 3000	min + 2x , min + (3x – 1)	14-20
> 3000 & <= 6000	min + x, min + (2x – 1)	7-13
> 6000	min, min + (x – 1)	0-6

Table 5.1 Period to Priority Range Mapping Rules

The priority range mapping rules used are very primitive. However these rules can be redefined over time once real-time OSGi applications are deployed on the Framework and the typical range of deadlines used in such applications emerge. Furthermore, the mapping rules would likely be changed to assign a larger range of priorities to the shorter periods rather than an equal divide across all periods.

Subsequent components passing the admission test are added to the list of currently installed components, their position is determined based on the size of their server period. The priority range to be assigned should be smaller than the component with the next smallest period (left neighbour in list) and higher than the component with next longest period (right neighbour in list). If there are sufficient free priorities between the priority ranges used by the component's left and right neighbours, the component will have its priority range assigned from these free priorities.

During priority range calculation, it is entirely possible that priorities of threads of currently deployed components may need to be remapped so that the priorities assigned to the new component and existing components' threads continue to reflect the RM priority assignment policy. This happens in the case where a component is added to the installed list and there are insufficient free priorities within the range required i.e. in the component list, there are insufficient free priorities between the priority ranges used by the next and previous components in the component list. For example, if a newly installed component's server has a logical priority higher than all other servers in the system and the highest priorities are already assigned to the currently deployed server with the highest priority, then priority reassignment will be necessary. Clearly this behaviour is undesirable, but unavoidable as previously mentioned. An example of priority reassignment is shown in Table 5.2. S_{New} is the server of a component undergoing admission control. It has a smaller period than S_2 but a larger period than S_1 According to Rate Monotonic priority assignment S_{New} therefore requires a priority range higher than S_2, but lower than S_1. However, there are insufficient free priorities available, assuming that S_{New} requires three priority levels. As a result, priority reassignment must take place and in this example, S_2 must have its priority range reassigned. This reassignment is performed by the admission control-invoking thread. The invoking-thread will obtain a reference to all necessary threads (through the thread references maintained by RT-OSGi) and will call setSchedulingParameters n each thread wit the new priority.

Server	Period/ Deadline	Before S_{New} Start Priority	Before S_{New} End Priority	After S_{New} Start Priority	After S_{New} End Priority
S_1	1200	25	26	25	26
S_{New}	1300	NA	NA	21	23
S_2	3100	19	23	15	19
S_3	5000	02	06	02	06

Table 5.2 Priority Range Reassignment

As a note, in terms of the priority reassignments in RT-OSGi, RT-OSGi does not reassign thread priorities directly. This would interfere with the semantics of execution-time servers. For example if the threads in an execution-time server exhaust their server's capacity, their priorities are lowered to a background level. If before their priorities are raised in accordance with their server's next capacity replenishment period, the install life cycle is invoked either by a thread in another component which has server capacity remaining[5] (component-derived) or from the user interface to RT-OSGi (user-derived), and this invocation of install results in priority reassignments, then such direct thread reassignments would mean that the threads in any execution-time server which has no capacity would begin to execute at their new priority level before their server capacity is next replenished. Therefore, the priority range assignment algorithm writes the new thread priority to a variable. This variable is then used by the replenishment handle to determine the new priority at which the thread should execute once the server has its capacity replenished. In this way, the semantics of execution-time servers are preserved.

Before discussing the next life cycle operation (component updates), it is important to point out here that, because the install operation can be called concurrently, there is a requirement to make the install admission control atomic. Without making this operation atomic, an issue may arise. The set of execution-time servers used in schedulability analysis is manipulated by the install operation, i.e. installing a component will add a new server to the list of servers used in schedulability analysis. If the install operation was not atomic, it would

[5] The implications for priority inheritance are discussed in Section 5.7.

be possible that a component falsely pass admission control. A thread may be executing the install operation and be part way through performing response-time analysis when it is pre-empted. The pre-empting thread may then itself call the install operation. Once this operation completes and the pre-empted thread regains control of the CPU, the result of the RTA it carried out will be invalid. The server created as a result of the pre-empting thread installing a component will not have existed at the time that the pre-empted thread was executing response-time analysis. As a result, the interference caused by this server will not have been included in the response-time calculations that completed before the operation was pre-empted. Therefore, the response-times of those threads calculated may have been larger than their deadlines (and therefore response-time analysis would have deemed the system unschedulable) had the server been included in the calculations.

Fortunately, the atomicity of the install operation should not be problematic to RT-OSGi applications in terms of blocking because the admission control from the install operation invocation, as previously discussed, is performed in an application non-real-time initialisation phase. Similarly, the admission control, and indeed the entire user-derived install operation itself is a non-real-time procedure. As a result, the blocking time is not of concern.

As a note, the uninstall operation (discussed in Section 5.6) also modifies the execution-time server set (used for schedulability analysis) by removing a component's execution-time servers from the list when the component is being uninstalled. Therefore, ideally, the admission control of this operation should also be made atomic with respect to the install operation so as to prevent the situation of components falsely failing admission control because the execution-time server belonging to the component was not removed from consideration in schedulability analysis before the operation was pre-empted by a thread performing the install life cycle operation, and as a result, response-time analysis would include the interference of a server that will actually not exist by the time that the component undergoing installation is deployed.

Since the uninstall operation (unlike the install operation) can be performed in a real-time context, blocking-time may be an issue. In order to prevent this issue, the uninstall operation should not be made atomic. Instead, although not desirable, it is safe to not have this operation be atomic with respect to the install operation and to simply tolerate the potential for false failure of admission control of components in this context.

5.4 Updating Components

A component update allows a component's contents to be changed and then deployed again, essentially acting as a short-hand for uninstalling the current version of a component and installing the new version of the component. As discussed in Chapter 3, if a component is in the Active state when the update operation is called, the component is stopped, moving ultimately to the Installed state. In terms of real-time systems, this means that the real-time threads in the component would need to be terminated as part of the semantics of the OSGi update operation. Clearly, when an active component is updated, this will result in deadline misses for the threads of the component being updated, and this is undesirable. Instead, in order to maintain a component's threads' real-time requirements during an update, it is necessary to have the new version of the component installed first. Only when the new threads have taken over the role of the old threads without breaking the timing constraints of the old threads can the old version of the component be removed. Thus, in RT-OSGi, when an active component needs updating, the OSGi update life cycle is not used and instead, a call should be made to the install operation with the new version of the component as a parameter followed by a call to the uninstall operation with the old version of the component when it is a safe to do so without breaking the threads' temporal requirements. This "mode change" protocol for RT-OSGi is discussed in Chapters 7 and 8.

Clearly, there are some real-time applications which are capable of initiating a non-real-time phase at some point during their execution in order to allow for component updates to take place. In these scenarios, there is no requirement to

maintain real-time constraints of a component's threads as it undergoes an update, and therefore, the update life cycle operation can be used. This section discusses the extensions to the update operation which enable the update operation to be used by such real-time applications.

As a component's thread set may change as part of an update e.g. a component may change the number of threads it creates, or the temporal specification (period, deadline, computation-time and priority) of threads may be changed, one would assume that the component must undergo admission control. However, this is not necessarily true.

In component-derived updates, the updates of a component are known prior to deploying a component, such updates can be treated as different versions of a component which the application code knows about and wishes to switch between during run-time. As the different versions of a component (update versions of a component) are known pre-deployment time, admission control is unnecessary for the component-derived update life cycle operation. Instead, as part of the component-derived install operation's admission control; resources should be reserved for the worst case version of a component. Switching between different versions (updating) is then not problematic because no analysis is required at the time that the update needs to take place. The reason for this is that since resources were reserved for the worst-case version of a component at install-time, it is safe to switch to using any version of the component without performing any analysis. Although this approach is necessary, it has the drawback that when the worst case version of a component is not deployed, other components may fail admission control because, despite sufficient resources being available for immediate use, they are reserved for later use by the worst-case resource using versions of the currently deployed components.

As discussed, in component-derived updates, the life cycle invoking code within a component is written pre-deployment time, and as a result, the updated version of a component must also be known pre-deployment time. In the case of user-derived updates, the updated version of a component is unknown pre-deployment

time as such updated versions of components are typically based on the user's observations of the currently running component-set. For example, the user may notice software errors, or may identify software components which would benefit from optimisations etc. Since the updated version of components is unknown pre-deployment time, it means that the resource requirements of a component will change in ways unknown at the time of component install, and therefore, unlike with component-derived updates, it is not possible to reserve resources for the worst-case version of a component. As a result, user-derived updates typically require the same admission control as the install life cycle operation (i.e. server parameter selection, schedulability analysis, and possibly priority range selection).

However, because a user-derived update may not necessarily change the thread set (e.g. by adding some configuration or HTML files to a component), update analysis can be carried out to determine whether it is necessary to perform the more computationally expensive install admission control. As a component is being updated, it must have been previously installed, and therefore must have server parameters generated for it as part of install admission control. Update Analysis [132] checks whether the server's parameters generated on component install are sufficient to make the threads of the updated version of the component schedulable i.e. this analysis tests whether the existing guaranteed resources for the component are adequate for the updated component. Only if the updated version of the component isn't schedulable with the previously generated server parameters is it necessary to perform component install admission control. As a note, only when the existing threads' temporal specification changes, or when the set of threads itself changes is it necessary to perform any update analysis or admission control associated with component installation. Therefore such changes can be tested for before performing any of the update and install analysis.

5.5 Starting Components

Once a component has passed the admission test and has been installed, it is likely that the component will be started. However, before a component can be started, and before threads can start running, the RT-OSGi Framework must perform some additional tasks. These tasks are divided into two phases: component initialisation and thread initialisation.

1) Component Initialisation: When a component is being started it must have an execution-time server created for it for temporal isolation purposes (as discussed in Chapter 4). The server is assigned the budget and period that were calculated during the server parameter calculation which was carried out as part of admission control when the component was installed. Furthermore, the execution-time server is then passed a reference to its component; this is so that the execution-time server can manipulate the component's threads' priority in accordance with server capacity exhaustion and server capacity replenishment. Finally, the component is passed a reference to the newly created execution-time server. The reason for this is so that, during construction, threads and other schedulable objects can add themselves to be managed by their component's execution-time server. These steps are shown in Figure 5.1.

2) *Thread Initialisation*: As discussed in Section 5.3, when a component is installed it is assigned a range of priorities from which its threads' priorities can be assigned. Since the actual priority range will be unknown until run-time, it is impossible for a component developer to assign absolute priorities to their threads in their RTSJ application code. It is therefore proposed that the component developer assign relative priorities to threads starting from zero for the thread with the longest period upwards to the number of required unique priorities minus one for the thread with the shortest period. For example if there are four threads the thread with the shortest period will be assigned three and the thread with the longest period will be assigned zero. This ensures that the relative ordering of priorities between threads is correct. It is then proposed that for each component, the RT-OSGi Framework stores an array of the priorities in the

component's range, with the lowest priority stored in `priorityArray[0]` and the highest priority stored in `priorityArray[numInRange - 1]`. The priorities used in the component's RTSJ application code can then be used as a lookup to the actual absolute priority to be used. For example the subclassed versions of classes implementing Schedulable (discussed in Chapter 4) can be extended. The subclasses can override the `setSchedulingParameters` method such that they extract the priority parameter, and reassign the priority to be `priorityArray[numberExtracted]`, calling the super class method with the priority obtained from the array lookup.

```
protected OSGiPGP generateServerParameters(ArrayList tasks,
      Bundle bundle)
{
      int[] serverParamResult = new ServerParamCalculator()
            .generateServerParams(tasks);
      OSGiPGP pgp = new OSGiPGP(new RelativeTime(0,0),
            new RelativeTime(serverParamResult[0],0),
            new RelativeTime(serverParamResult[1],0),
            new RelativeTime(serverParamResult[0],0),
            null,null);
      pgp.setBundle(bundle);
      ((RTBundle)bundle).setPGP(pgp);
  return pgp;
}
```

Figure 5.1 Execution-Time Server Creation and Initialisation

5.6 Stopping and Uninstalling Components

Although removing components (as with starting components) does not require admission control, the OSGi life cycle operation responsible for removing components nevertheless needs extending for use with RT-OSGi. The two ways in which component removal needs extending are discussed below.

5.6.1 Controlling the Life-Time of Threads

As the OSGi Framework is Java based, and the standard Java language provides no safe way of terminating threads, OSGi does not attempt to coordinate the life

cycle of threads with the life cycle of the component from which they belong i.e. the OSGi Framework has no control over the threads started by application components. An implication of this is that, unless an application is designed with synchronising component and thread life cycles in mind, threads may continue to execute after their component has been uninstalled from the OSGi Framework. Such "runaway" threads are a resource leak using up CPU time and memory. Furthermore, threads may cause errors if they continue execution beyond the point when their component is uninstalled. This is because they may attempt to use code and data resources of their component, which is no longer available. Such problems are not tolerable when the OSGi Framework is to be used in the development of real-time systems.

The RTSJ introduces Asynchronous Transfer of Control (ATC) [24] which allows a thread to cause another thread to abort its normal processing code and transition to some exception processing code. Consequently, a thread may be executing in one method and then suddenly, through no action of its own, find itself executing in another method. While supporting ATC has a number of potential drawbacks such as complicating programming language semantics, increasing the complexity of reasoning about application code correctness because of transfer of control, slowing down the execution of code which does not use ATC, and increasing the complexity of the real-time JVM, ATC does have a number of advantages. Amongst the advantages of ATC such as supporting coordinated error recovery between real-time threads, supporting mode changes in real-time systems, and supporting imprecise computations, perhaps the most significant advantage is asynchronous thread termination (ATT). ATT through ATC allows threads to terminate more safely than simply using facilities provided by the OS to terminate a thread, furthermore, ATT through ATC allows for the termination much more quickly than having the thread poll for notification of termination.

ATC is achieved in the RTSJ by combining the exception handling model of Java with extensions of the standard Java thread interruption mechanism. Rather than real-time threads having to poll for interruption (as threads do in standard Java), the RTSJ instead throws an asynchronous exception at the thread when

interrupt() (in the RealtimeThread class) is called. All methods which are prepared to allow the delivery of an asynchronous exception (AsynchronouslyInterruptedException (AIE)) place the exception in their throw list. The RTSJ calls such methods AI-methods (Asynchronously Interruptible). An object which wishes to provide an interruptible method does so by implementing the Interruptible interface The interface's run(…) method is the method that is interruptible; the interruptedAction(..) method is called by the system if the run(…) method is interrupted. Once an application has implemented this interface, the implementation can be passed as a parameter to the doInterruptible(…) method in the AsynchronouslyInterruptedException class. The run(…) method can then be interrupted by calling the fire(…) method in the AsynchronouslyInterruptedException class

Such an asynchronous transfer of control is safe because if a method is not declared an AI-method, then the asynchronous exception is not delivered but instead held pending until the thread is in a method which has the appropriate throw clause. For example, synchronized methods defer the ATC in this way. This means that before ATC takes place any locks being held are released. ATC can therefore be used for the asynchronous termination of threads (ATT) when a component is uninstalled from the OSGi Framework. Similarly, if it is imperative that some application logic complete before the thread terminates, this code can also be placed in an ATC deferred method. This ATC is enforced by integrating it into the RT-OSGi class hierarchy, as discussed in Chapter 8.

It is proposed that ATC be used for the asynchronous termination of threads (ATT) when a component is stopped in the OSGi Framework. In order to ensure that threads are cooperative with such a scheme, the classes presented in Chapter 4 are extended. For example the class OSGiRTT is extended to implement the Interruptible interface. Figure 5.2 shows the RTSJ's Interruptible interface. When a thread is constructed it creates an AsynchronouslyInterruptedException and passes a reference to its component, which is managed by RT-OSGi. When a component is stopped, RT-

OSGi can iterate through the list of AIE references associated with the threads of the component being stopped, calling `fire()` on each AIE reference. This causes the thread to terminate its `run()` method and execute its `interruptAction(…)` method as previously discussed.

OSGiRTT is also made abstract such that subclasses must provide an implementation of the `Interruptible` method `run(…)`.The `Interruptible` method `interruptAction(…)` is implemented in the abstract OOSGiRTT class so as to prevent application developers from using the method to potentially find a way to enable their threads to continue execution when they should be terminated. In addition, OSGiRTT's `run()` method (from `RealtimeThread`) calls `doInterruptible(this)`, combined with the fact that `run()` is also made final, this means that when subclasses call `start()`, OSGiRTT's `run()` method is called which will result in the subclasses' `Interruptible run` method being executed and this thus forces threads deployed in RT-OSGi to support asynchronous termination via ATC. Figure 5.3 shows the extensions to OSGiRTT which enforce asynchronous thread termination in RT-OSGi through the RTSJ's asynchronous transfer of control. Figure 5.4 shows an example of an application thread which is forced to support asynchronous thread termination.

As a note, since ATC can be deferred, when a component is stopped, its threads may not terminate immediately. It is therefore recommended that component developers avoid using long ATC deferred methods. In the case where this is not possible, the impact of long running ATC deferred methods executing after the component has been stopped can be minimised by lowering the priority of any remaining executing threads to a background priority. They will remain at this priority until they terminate on return from an ATC deferred execution context.

```
package javax.realtime;
import javax.realtime.*;

public interface Interruptible
{
	public void interruptAction
		(AsynchronouslyInterruptedException exception);
	public void run
		(AsynchronouslyInterruptedException exception);
}
```

Figure 5.2 the RTSJ's Interruptible Interface

```
import javax.realtime.*;

public abstract class OSGiRTT extends RealtimeThread
	implements Interruptible
{
  //will allow thread to safely terminate
  public final void interruptAction
	(AsynchronouslyInterruptedException exception)
  {

  }

  //method of RealtimeThread made final to enforce ATT
  public final void run()
  {
	AsynchronouslyInterruptedException a=
	  new AsynchronouslyInterruptedException();
	a.doInterruptible(this);
  }

  //method of Interruptible interface
  public abstract void run
	(AsynchronouslyInterruptedException exception);
}
```

Figure 5.3 OSGiRTT Asynchronous Thread Termination Extensions

```
import javax.realtime.*;

public class MyThread extends OSGiRTT
{
	public void run(AsynchronouslyInterruptedException exception)
	{
		while(true)
		{
			System.out.println("Thread Executing");
		}
	}
}
```

Figure 5.4 Example of an Application Thread in RT-OSGi

5.6.2 Resource Reclamation

After a component is uninstalled, the resources used by the component (such as CPU reservation and priority range) must be made available for future components to use. As discussed, the resource reservation is simply a specification (computation-time (C), period (T), deadline (D)) used in schedulability analysis along with cost enforcement. Therefore, removing the reservation is easily achieved by removing the component's server from the list of servers considered as part of schedulability analysis, destroying the component's server (PGP) and replenishment timer, and by making the priority range used by the component available to other components.

In addition to the above, had RT-OSGi targeted applications with adaptable levels of QoS, RT-OSGi would also provide some form of dynamic reclamation of resources. When a component is uninstalled RT-OSGi would have the resources used by the component being uninstalled distributed to the other active components in the Framework. However, since RT-OSGi does not currently target such adaptable applications, dynamic reclamation is not discussed further in this thesis.

5.7 Blocking and Life Cycle Operations

Having discussed the admission control related to life cycle operations, it is necessary to discuss the issue of blocking.

In the OSGi Framework, it is typical that deployed applications will consist of a number of thread-containing components. However, since the OSGi specification does not address concurrency, it is the responsibility of implementers of the specification to ensure thread safety when calling OSGi code such as life cycle operations. The synchronisation used in the Apache Felix OSGi Framework implementation (on which RT-OSGi is based) is as follows. When calling `install(…)`, the lock of the object associated with the bundle being installed

is acquired. Blocking during component install will only occur if two threads attempt to install the same component and thus attempt to acquire the lock associated with an object representing that component. Two threads calling install with different component identifiers will not cause blocking because they will acquire different locks. Clearly, no other life cycle operations can be invoked on a component that has not yet been successfully installed.

Calling the life cycle operations `stop()`, `start()`, `uninstall()` and `update()` acquires the lock for the associated component, thus blocking will occur if multiple threads attempt to concurrently call any of these operations on the same component. Furthermore, calling start() and update() may start the process of resolution, which as one may recall from Chapter 3, is in an attempt to match any Java package import requirements of the component with package exporting components currently deployed. Before performing resolution as part of start() or update(), the locks for all components are acquired until resolution completes, because resolution requires that the component set does not changed during its execution. Blocking will occur if a thread calls either start() or update() on a component which requires resolution while one or more other threads call any life cycle operation (with the exception of `install()`).

Since one application's components' threads should not be permitted to invoke life cycle operations on components belonging to another independent application, the issue of blocking due to acquiring the same lock should not happen. Furthermore, since it makes no sense to have an application's components trying to call life cycle operations on the same component simultaneously, blocking as a result of calling these life cycle operation should not occur within an application. However, as mentioned above, it is entirely possible that one applications components' threads may block the threads of another application by calling the `start()` or `update()` operations causing resolution to occur and thus acquiring the locks to such independent applications' components. Thus, resolution is the only form of blocking considered in this thesis.

In terms of the effect of blocking on RT-OSGi applications, consider the following example. A thread in a component calls `start()` which requires the locks to all other components as part of resolution. For one reason or another, the WCET of calling resolution was not included in the execution-time server parameter selection when the calling component was first installed into RT-OSGi thus the server capacity is insufficient to allow the call to resolution to complete. As a result, when the server capacity is exhausted, the priorities of all of the component's server's threads are lowered, including the thread holding the locks to all components. Priority inversion will then occur when a thread in a component executing under a different server calls a life cycle operation. Priority inversion may also occur even if the server has capacity remaining, for example, if a thread holding the resolution lock is pre-empted by a higher priority thread requiring the same resolution lock or a lock to a component. In either of these cases, the RTSJ's priority inheritance protocol [26] will be activated so as to bound the priority inversion (i.e. the priority of the lock holding thread will be raised to that of the lock requesting thread to prevent threads with priorities between these two threads from executing and thus blocking the lock requesting thread even further). Clearly the priority inheritance may cause threads to miss their deadlines if threads executing when their server has no capacity was not taken into consideration during schedulability analysis.

In order to prevent deadline misses from occurring as a result of the aforementioned priority inversion, RT-OSGi can do the following. Priority inversion from the pre-emption of a thread with remaining server capacity can be solved by having RT-OSGi itself calculate the blocking time [26] of any threads which execute the resolution operation and therefore acquire the resolution lock. This blocking time can easily be calculated since the WCET of resolution is known to RT-OSGi as are the total number and priorities of application threads. Furthermore, component developers can convey the MIT of resolution calls in their threads to RT-OSGi through the component's manifest file. Once calculated, the blocking time can then be included in schedulability analysis as part of a component's admission control. Priority inversion from a thread holding the resolution lock after its server capacity has been exhausted can be solved by firstly having the WCET of calls to the start and update operations (which both

call resolution and acquire the resolution lock) included in the server parameter selection process so as to try and prevent the priority inversion situation from occurring. Secondly, as discussed briefly in Chapter 4, execution-time servers in RT-OSGi can be given two capacities, a soft capacity and a hard capacity. When the soft server capacity is exhausted, a check is made by RT-OSGi. If RT-OSGi finds that the resolution lock is held by a thread that executes under the server whose soft capacity is exhausted, the server is allowed to continue execution until its hard capacity is reached. The difference in computation-time between the soft and hard capacity should be just enough to allow resolution to complete and the resolution lock to be released before the server's threads have their priorities lowered. Thirdly, because threads continue to execute at a non-real-time priority after server capacity exhaustion, it is entirely possible that these threads will attempt to acquire the resolution lock. However, since overrunning the server capacity is essentially an error condition likely resulting in deadline misses for the server's threads, the issue of acquiring the resolution lock can easily be solved by having the resolution method check the priority of the calling thread. If the calling thread's priority is a background priority indicative of server capacity exhaustion, the calling thread is denied the lock and is instead blocked for its next release.

Finally, in addition to the aforementioned resolution-induced priority inversion occurring between application threads, it may also occur as a result of the thread responsible for processing the user-derived life cycle operations. Therefore, this thread's server should also have a soft and hard server capacity like the application servers. Moreover, the life cycle processing server should also be included when calculating the blocking time for application threads. As a note, since the life cycle processing server does not have real-time requirements, it is not necessary to calculate the blocking time for the server.

5.8 Summary

Two significant issues in the standard OSGi Framework are unbounded dynamism and "runaway" threads which live beyond the lifetime of the

component which created and started them. Unbounded dynamism means that components can be installed and updated without regulation, and as a result, the CPU may become overloaded such that it becomes impossible to guarantee the timing requirements of components' threads. "Runaway" threads are a resource leak using CPU-time and memory beyond the life-time of the containing component. Furthermore, such threads may cause errors by attempting to use resources allocated to their containing component which no longer exist.

In this chapter these issues are solved by extending the OSGi life cycle operations with an admission control protocol and asynchronous thread termination (ATT) in RT-OSGi. A summary of these extensions is shown in Figure 5.5. The admission control, which consists of execution-time server parameter selection, schedulability analysis and priority range assignment, determines the resource requirements of a component and determines whether it is possible to allocate those resources to the component without causing threads in currently deployed components to miss their deadlines. Only if deadlines misses can be avoided is the component permitted to be deployed. ATT is provided through integrating the RTSJ's asynchronous transfer of control (ATC) mechanism into the class hierarchy that application developers are required to use in order to deploy applications on the RT-OSGi Framework.

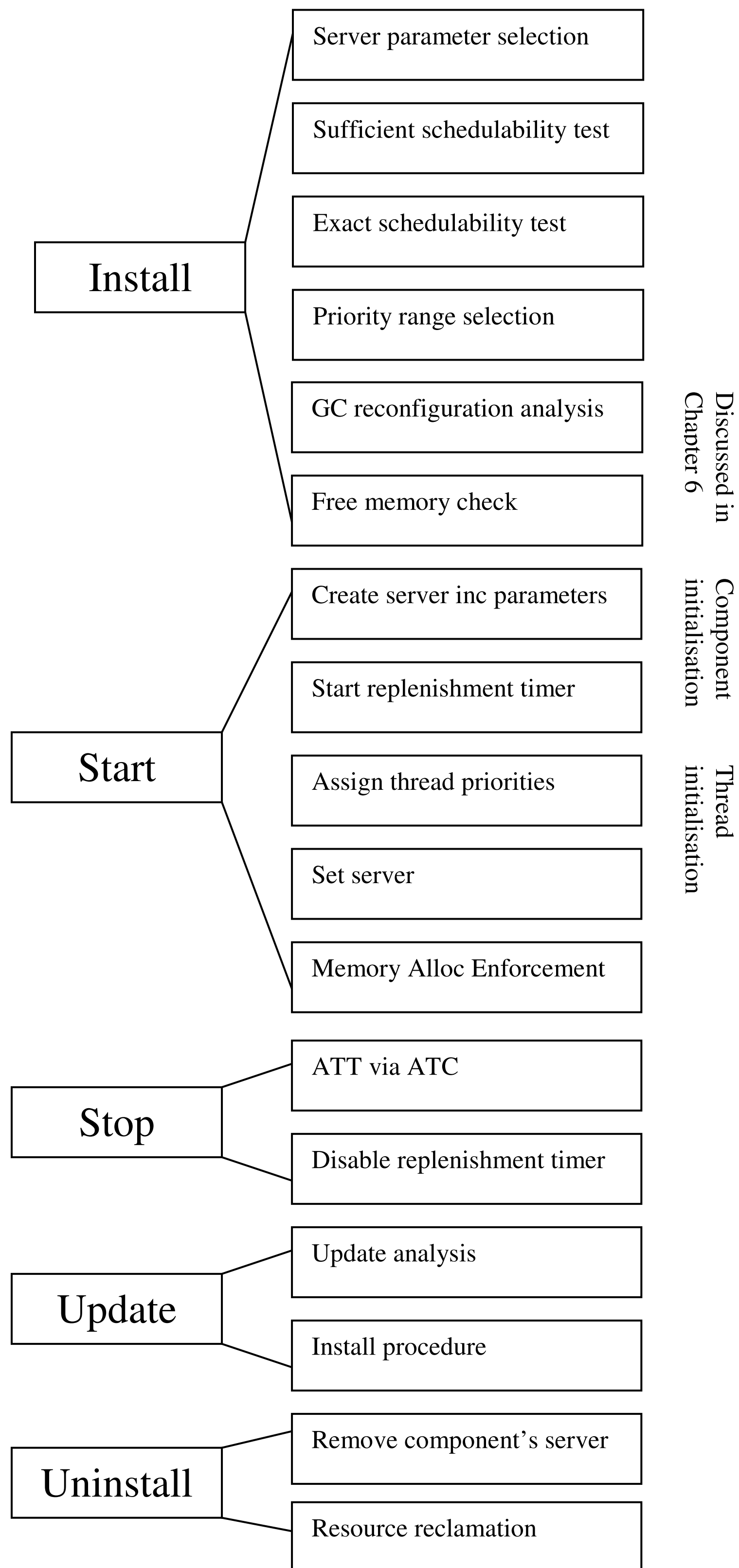

Figure 5.5 Summary of Life Cycle Operation Extensions in RT-OSGi

6

Memory Management

6.1 Introduction

In addition to the garbage collected heap of standard Java, the RTSJ provides other memory areas including a region-based approach to memory management called scoped memory (SM) [22]. A scoped memory area is a region of memory that has a reference count associated with it which keeps track of how many real-time entities are currently using the area. When the reference count goes to zero, all of the memory associated with objects allocated in the memory region is reclaimed. One of the benefits of scoped memory is that it avoids the overheads and possible unpredictability of garbage collection (GC). However, the third party software development nature of SOA means that SM is not a general solution to memory management in RT-OSGi, as discussed below.

There are two possible approaches to using SM with RT-OSGi services. Threads can either enter SM before calling services, or the service can take responsibility for creating SM and have calling threads enter it from within the service method. In the former case, IllegalAssignmentErrors will be thrown if a service method breaks the RTSJ memory assignment rules. In the latter case, ScopedCycleExceptions may be thrown depending on the scope stack of calling threads. Also, since multiple threads are able to call a service concurrently, it is necessary to synchronize access to the service's SM. Not doing so would mean that there is the potential for the SM's reference count to constantly remain above zero eventually causing memory exhaustion of the SM area. This issue can be solved by using synchronization. However, this introduces a further issue,

calling threads may experience blocking, and this must be taken into account when doing schedulability analysis.

Due to the issues with SM, along with the fact that RT-OSGi applications do not require the stronger timing guarantees which SM can provide over Real-Time Garbage Collection (RT-GC), RT-GC rather than SM is considered for use with RT-OSGi.

The general idea of the approach to memory management in RT-OSGi is to calculate the memory requirements of each thread by executing the local code and using memory contracts (similar to the WCET contracts discussed in Chapter 4) for services during testing, and from that, along with other factors (discussed later in this chapter), deriving the amount of GC work necessary to complete a GC cycle. This information can then be used to determine the GC parameters online (as new components are being deployed), and from the GC parameters, the amount of time and memory required to complete a GC cycle. The time and memory requirements of a GC cycle can then be used as acceptance tests in order to provide memory-related extensions to the CPU admission control discussed in Chapter 5, thus preventing overload situations. In this way, it is guaranteed that each GC cycle will complete and reclaim the memory that was occupied by garbage before the currently deployed threads executing in parallel with the GC thread exhaust memory.

In the remainder of this chapter, the issues with current real-time garbage collectors in the context of dynamically reconfigurable real-time applications such as RT-OSGi applications are firstly discussed. Then, the memory management approach used in RT-OSGi is discussed. This includes: the implementation of a dynamically reconfigurable GC based on one of the GCs provided by a major RTSJ implementation, the reconfiguration analysis needed to support the dynamically reconfigurable GC, and finally, the memory allocation enforcement used to support the dynamically reconfigurable GC.

6.2 Current Real-Time Garbage Collectors

In real-time systems, the RT-GC must recycle memory often enough to prevent memory exhaustion without disrupting the temporal predictability of real-time threads. As the RTSJ does not specify any particular GC algorithm, current major RTSJ implementations use different GC algorithms. A brief survey of the GCs provided by current RTSJ implementations is given below along with examples of the issues of using these GCs with dynamic environments such as RT-OSGi applications. For a thorough introduction into garbage collection in the context of uniprocessor systems and for further insight into the general techniques of garbage collection such as mark-sweep, mark-compact, copying collectors etc, please consult [20]. For an in-depth discussions of the history and state of the art of real-time garbage collection, see [133].

Aicas JamaiacaVM [134] provides a work-based GC [135]. In such a garbage collection scheme, the threads that allocate memory must perform some level of GC. The amount of work to be carried out during memory allocation depends on whether the work-based GC has been configured to be static or dynamic. In the static work-based approach, heap allocating threads perform a fixed number of GC work units per allocation block. The number of blocks is determined based on the worst case live memory of the application. JamaicaVM provides a tool that allows the worst case live memory of an application to be collected, and based on that, the tool generates possibilities for the heap size and associated GC work units required to keep the application from exhausting memory. Configuring the system with these parameters will then ensure that the GC keeps up with garbage creation, and that the application never exhausts memory. This GC is ideal for hard real-time systems, where the application architecture is static and does not change at run-time.

Despite the benefits of work-based garbage collection in hard real-time systems, if the worst case live memory of an application changes from that used to determine the number of GC work units, as would be the case with the dynamically reconfigurable RT-OSGi applications, then, as the number of GC

work units per allocation block cannot be reconfigured by the user during run-time, the amount of GC work carried out may be insufficient to prevent memory exhaustion. For example, an RT-OSGi application's architecture may change from the one used to configure the GC such as by installing new thread containing components or by having threads bind with different implementations of service interfaces. Such application reconfiguration without the ability to reconfigure the amount of GC work will mean that the GC may not be able to collect garbage at a rate fast enough to prevent memory exhaustion. Memory exhaustion would then cause application threads to stop making progress until for example demand-GC occurs and memory is released. Thus it is likely that the real-time constraints of the threads would be violated. Figure 6.1 gives an example of this. In the figure, the application is reconfigured as highlighted. Specifically, T_1's memory allocation per period (A) is changed, T_2's period (T) is changed, and T_4, T_5, and T_6 are introduced into the application. However none of these changes are reflected in the GC parameters.

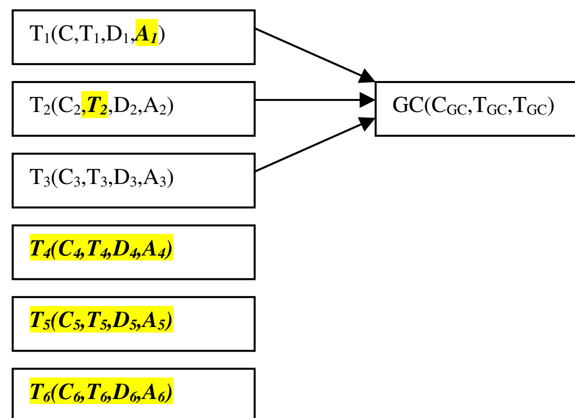

Figure **6**.1 Application Reconfiguration Not Reflected in GC Rate

In the dynamic approach to work-based GC provided by Aicas, based on the live memory of an application, the worst-case number of GC work units is calculated. Unlike the static work based approach, the number of GC work units per allocation will vary during run-time. Even so, the worst case number will be known, and can be used as part of the worst case execution time calculation of an application's threads. If the worst case live memory changes during run-time,

then although memory will not become exhausted because the allocating threads will do as much GC work as is necessary to prevent memory exhaustion, it may means that such threads will perform more GC work than the calculated worst case amount. As a result, the threads' WCET will not be correct, and because the threads' WCET is an important input into schedulability analysis, incorrect WCETs may cause the system to become unschedulable and so real-time threads may violate their timing constraints. In order to prevent this situation, each time the application is reconfigured, it would be necessary to determine what the new worst case number of GC work units per allocation block would be, then perform WCET analysis for all threads, and finally perform schedulability analysis. This is impractical in a system such as RT-OSGi, where reconfiguration is expected to happen on a relatively frequent basis.

The GC provided by IBM WebSphere RT [136] is time-based. Time-based GCs run at precise time intervals for a predetermined length of time i.e. there is essentially one or more periodic threads, with a priority one higher than the highest real-time thread in the user's application, performing GC work each period. In the IBM time-based GC, the GC thread has a period of 20ms and a computation time which is configured by the user. In this way, the user can configure the CPU utilization used by the GC thread. To ensure that the GC utilization is sufficient, IBM provides memory analysis tools such that the necessary GC parameters for an application can be generated. This gives deterministic GC for the life of the application and it is much simpler to take into consideration during real-time analysis than work-based collection, where the computation time of each thread becomes affected by the memory use of the application. Unlike work-based garbage collection though, time-based collectors penalise all threads with a lower priority than the GC thread even if they do not allocate on the heap.

Reconfiguring the pace of GC is easily achieved in time-based collectors modifying the computation time, period, and deadline of the GC thread. Such an approach to GC is well suited to dynamically reconfigurable applications such as RT-OSGi applications. Unfortunately, IBM provides no facilities for changing

the parameters of their time-based GC and so the IBM GC suffers from the same issue as the JamaicaVM static work-based GC in that its parameters may be insufficient to prevent the rate of garbage creation exceedng the rate of garbage collection thus causing memory exhausting and violation of threads timing constraints.

Sun Java RTS [137] provide a Henriksson style GC [138] which can be dynamically reconfigured. The default behaviour of this JVM is to have the GC thread run at a background priority (i.e. a priority lower than the lowest priority thread in the application but above the priority of non-real-time application threads). If the free memory available to an application drops below a user-defined threshold, the priority of the GC is increased from a background level to a "boosted" user-configurable level. In this way, the rate of GC is temporarily increased until the GC completes one or more cycles and increases the free memory above the safe level threshold, it then returns to executing at a background priority level. This model is depicted in Figure 6.2 [137].

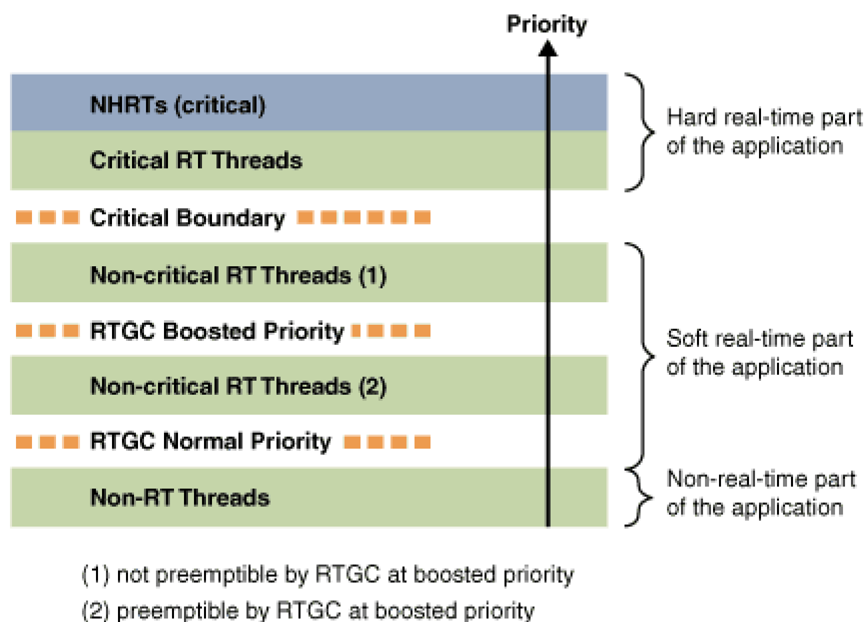

Figure **6**.2 Sun Java RTS Model of Garbage Collection

From Figure 6.2, it can be seen that the threads that have a priority above the "RTGC Normal Priority" level but below the RTGC Boosted Priority level

(termed non-critical in Java RTS) may be subjected to an unbounded amount of interference from the RT-GC thread whilst it executes at its boosted priority level for as long as it takes to return the free level of memory above the unsafe threshold. Therefore such application threads cannot have real-time guarantees i.e. to such threads GC appears to be stop-the-world. Threads with a priority above the "RTGC Boosted Priority" but below the "Critical Boundary" will not be pre-empted by the GC thread but may have to perform demand GC if memory becomes exhausted. Finally threads with a priority above the "Critical Boundary" will not be pre-empted by the GC and will not have to perform demand GC because they have an area of the heap exclusively reserved.

In order for the Java RTS GC model to function correctly i.e. to prevent memory exhaustion, the GC must be able to keep pace with the rate of garbage creation. This typically means dividing the application threads such that the majority are non-critical and can be pre-empted by the GC, allowing the GC thread to be given a large fraction of the CPU utilisation if the free memory level drops below the safe threshold. If the majority of application threads have a priority above the "RTGC Boosted Priority" level, then even when free memory is low, the GC would still be running at background priority as threads with a priority above the "RTGC Boosted Priority" level cannot be pre-empted by the GC, and the GC would not be very effective at preventing memory exhaustion and the associated timing constraints violations of application threads.

Clearly, the Sun Java RTS GC model is only useful when an application has a small number of critical threads. However, many applications require that the majority of their threads not suffer unbounded interference from the GC thread, and thus the default behaviour of Sun Java RTS is unsuitable. A more suitable approach to GC would be to have the GC keep pace with garbage creation such that the free memory should never fall below the free memory threshold, and thus there would be no need to have the GC monopolise the CPU by assigning the GC thread a large CPU utilisation in an attempt to frantically "catch up" with the application's garbage creation. Essentially, a preventative approach to

reaching low free memory levels is better than Sun Java RTS GC's default curative approach.

6.3 Garbage Collector Reconfiguration

Since the Sun Java RTS is dynamically reconfigurable in the sense that its priority can be manipulated during run-time so as to increase or decrease the rate of garbage collection, it is possible to implement a dynamically reconfigurable garbage collector by using analysis to determine the pace of GC based on the current configuration/architecture of the deployed software and thus the rate of garbage creation. As there are a number of research papers dedicated to the topic of configuring a time-based GC, time-based GC is implemented using Sun Java RTS's GC, using accompanying analysis to determine in what way the GC parameters need to be changed based on the changing application architecture. Such a dynamically reconfigurable GC with associated analysis ensures that garbage related memory exhaustion does not occur because the application can only be dynamically reconfigured when the GC can also be dynamically reconfigured to prevent memory exhaustion without breaking application schedulability. Thus reconfigurable GC can support dynamically reconfigurable RT-OSGi applications, whereas the other current GCs may cause timing violations for dynamically reconfigurable applications.

Time-based GC can be implemented on Sun Java RTS's GC by having a GC control thread which can modify the GC thread's priority so as to simulate the GC thread running with a computation time per period (as happens in time-based GC). The GC thread is assigned a background priority (thus appearing inactive), and the GC control thread is assigned whatever period the GC thread is required to run at, and as the period is typically small, according to Rate Monotonic priority assignment, the GC control thread will run at the highest priority in the system, above all application threads and the GC thread itself.

As the GC control thread is released, it raises the priority of the GC thread to a priority higher than all application threads but below its own priority level. The

GC control thread can manipulate the amount of time that the GC thread is permitted to run at high priority for during its period by using a timer. Once the GC thread has used its allocated computation time per period (i.e. the GC control thread's timer expires), the GC control thread will then pre-empt the GC thread and will lower the GC thread's priority back to its background level i.e. to a priority level below all application real-time threads. This behaviour can be achieved by making use of a number of methods of the FullyConcurrentGarbageCollector class provided by the Sun Java RTS implementation of the RTSJ. These methods, i.e. a subset of the FullyConcurrentGarbageCollector class are shown in Figure 6.3.

```
package com.sun.rtsjx;

public class FullyConcurrentGarbageCollector extends GarbageCollector
{
    public static native void startAsyncGC(int paramInt);

    public static int getCriticalBoundary()
    {
        return (int)get(CriticalBoundaryID);
    }

    public static native int getNormalPriority();

    public static boolean set(String paramString, long paramLong)
    {
        int i = getParameterID(paramString);
        if (i == 0)
        {
            return false;
        }
        set(i, paramLong);
        return true;
    }

    public static native boolean set(int paramInt, long paramLong);

}
```

Figure 6.3 Subset of FullyConcurrentGarbageCollector class Used to Support Time-Based GC for RT-OSGi

In Figure 6.3, the method `getNormalPriority()` returns the priority at which the GC should execute during normal mode i.e. when the free memory in the system is not below some user-defined threshold. The `getCriticalBoundary()` method returns the maximum priority level at

which the GC should be allowed to execute at, as discussed in Section 6.2. Since RT-OSGi considers all real-time threads to be time-critical, this method will return a priority higher than the highest priority real-time thread in an RT-OSGi application. The method `startAsyncGC(int paramInt)` starts a GC cycle, and finally, the method `set(String paramString, long paramLong)` is used to set various attributed of the GC such as the normal and boosted priorities as well as the critical boundary priority

A dynamically reconfigurable time-based GC which is able to support dynamically reconfigurable RT-OSGi applications can then be implemented from the aforementioned methods of the FullyConcurrentGarbageCollector class of the Sun Java RTS implementation of the RTSJ. Such an implementation is shown in Figure 6.4. In this figure, a real-time thread is created called the GC control thread. It has a period equal to that which the time-based GC should execute with (which is calculated based on the application thread-set, discussed shortly), and is assigned the highest priority in the system. At the start of its period, the thread starts a GC cycle (i.e. the Sun Java RTS thread) if one is not already in progress (using `startAsyncGC(int paramInt)` and then raises the priority of the Sun Java RTS GC thread from a background priority to a priority higher than the highest priority thread in the application but lower than the GC control thread's priority level (using `.set(String paramString, long paramLong))`. The GC control thread controlling the GC thread then self suspends (by holding the lock for its object and invoking `HighResolutionTime.waitForObject` with its object as a parameter) for an amount of time equal to the "cost" that the GC thread should be allocated at high priority. The result of this is that the GC thread is able to execute in preference to all application threads for its "cost" with the period of the GC control thread. Once the timer expires for the GC controller thread, the GC control thread regains the CPU from the GC thread and returns the GC thread's priority back to a background priority. As a note, it is important to clarify here that this time-based model of GC implemented on the Sun Java RTS no longer follows the standard model of GC in Sun Java RTS that is depicted in Figure 6.2. Rather, the critical boundary is set to be above all application threads and the

RTGC boosted priority is set to equal the critical boundary. Further, the normal priority level is set to be below all application threads. As a result, the GC controller thread (which has a priority higher than both the application and GC threads) will raise the GC thread priority from below that of all application threads to above that of all application threads periodically thus giving a time-based model of GC.

```
import javax.realtime.*;
import com.sun.rtsjx.*;

public class TimeBasedRTGC implements Runnable
{
	final int RTSJ_MAX_PRI = PriorityScheduler.instance
		().getMaxPriority();
	RealtimeThread rtt = null;
	int costVal;
	int normalPri, boostedPri;

	public TimeBasedRTGC(int periodVal, int costVal throws Exception
	{
		this.costVal = costVal;
		normalPri=
			FullyConcurrentGarbvageCollector.getNormalPriority();
		boostedPri=
			FullyConcurrentGarbvageCollector.getCriticalBoundary();
		PriorityParameters priority = new
			PriorityParameters(RTSJ_MAX_PRI);
		PeriodicParameters period = PeriodicParameters(
			new RelativeTime(periodVal,0));

		//create an rtt responsible for giving GC periodic behaviour (GC control thread)
		rtt = new RealtimeThread( priority,period,null,null,null,this);
		rtt.start();
	}

	private void boostRTGC()
	{
		FullyConcurrentGarbageCollector.set
			("NormalPriority",boostedPri);
	}

	private void unBoostRTGC()
	{
		FullyConcurrentGarbageCollector.set
			("NormalPriority",normalPri);
	}
```

```
    public void run()
    {
        //obtain lock, used to block on (with waitForObject below) in order
        //to give the GC thread the CPU
        synchronized(this)
        {
            //on every release of GC control thread
            while(true)
            {
                //start a new RTGC cycle if not in one (i.e. start the
                //Sun Java RTS GC thread)
                FullyConcurrentGarbageCollector.
                    startAsyncGC(normalpri);
                //assign Sun Java RTS its critical boundary priority, which
                //should be configured to be less than the GC control thread but
                //greater thna the application threads
                boostRTGC();
                try
                {
                    //start a timer to equal the amount of time that
                    //the Gc thread should be allowed to execute for
                    RelativeTime cost = new RelativeTime
                        (costVal,0);
                    //blocks this Gc control periodic thread
                    //and allows GC thread to  execute until timer expires
                    HighResolutionTime.waitForObject
                        (this,cost);
                }
                catch(InterruptedException e)
                {
                }
                //set GC priority back to normal
                //(below application threads)
                unboostRTGC();
                //wait for the next release and repeat
                RealTimeThread.waitForNextPeriod();
            }
        }
    }
}
```

Figure 6.4 Implementing Time-Based GC Using Sun Java RTS's FullyConcurrentGarbageCollector Class

6.3.1 Garbage Collection Reconfiguration Analysis

Using the aforementioned simulated time-based GC, it is still necessary to perform some analysis when the application undergoes dynamic reconfiguration to determine the new GC parameters (computation-time, period, and deadline) such that the GC can collect at a pace sufficient to prevent garbage-related memory exhaustion. In this thesis, this analysis is termed GC reconfiguration

analysis. Regarding the actual reconfiguration analysis to be used with RT-OSGi and the modified Sun Java RTS GC, a modified version of Kim's analysis [139] is applied online i.e. at the same time as CPU admission control (discussed in Chapter 5). This analysis is carried out after the CPU admission control discussed in Chapter 5 when the install life cycle operation is invoked. If the component passes the CPU admission control and the GC reconfiguration analysis/memory admission control (discussed in this chapter), then the GC parameters can be reconfigured and the component can be deployed.

In Kim's analysis, the amount of GC work is estimated and an approximation of the maximum amount of time that can be allocated to the GC is calculated while maintaining application schedulability. Based on these values, the worst case length of a GC cycle and the worst case memory requirement of the application are determined. This differs from other research works on configuring the garbage collector. Most other works [140-142] focus on finding the minimum amount of CPU time that can be allocated to the GC while preventing memory exhaustion. More specifically, the deadline for when the GC thread must complete its cycle and make new memory available for allocation essentially becomes the length of time it takes for memory to become exhausted due to a build up of garbage memory from the currently deployed threads. That is, if the free memory and the periods and worst-case allocation per period of each application thread are known, the amount of time it will take for memory to become exhausted can be calculated. This approach is shown more formally in Equation 6.1 [140]. In this equation, D_{GC} = the deadline for completing a GC cycle, H is the heap size, L_{max} is the maximum amount of live memory of the application before the start of a GC cycle, p is the set of application threads, a_j is the amount of memory allocated by a thread, f_j is the frequency of execution of a thread (one divided by the thread's period). As a note, $H - L_{max}$ is used in the equation rather than using the amount of free memory before a GC cycle so as to be more conservative by taking into consideration floating garbage. Floating garbage are objects that die after being traversed and therefore continue to occupy memory until they are reclaimed at the completion of the subsequent GC cycle [133].

$$D_{GC} \leq \frac{\frac{(H - L_{\max})}{2} - \sum_{j \in \rho} a_j}{\sum_{j \in \rho} f_j \times a_j} \qquad 6.1$$

Equation 6-1 Calculating the Necessary GC Cycle Time

In addition to discussing GC configuration analysis, Nilsen [143] further discusses the PERC Ultra JVM, a real-time Java alternative to the RTSJ. More specifically, the notion of a pacing agent is discussed. This agent monitors the worst case live memory and the allocation rate of an application during previous GC cycles and configures the GC for the next cycle based on this historical knowledge. In addition to such automated dynamic reconfigurability, the GC can be dynamically reconfigurable by the user application through the provision of a GC API. However, the GC reconfiguration analysis is not integrated with the PERC Ultra JVM and therefore users would need to implement such GC reconfiguration analysis in their applications. Another issue is that, despite providing real-time functionality to Java applications, the PERC Ultra JVM is not an implementation of the RTSJ and thus it doesn't offer the advantages to OSGi that the RTSJ does. Therefore PERC Ultra JVM and its associated dynamically reconfigurable GC are not discussed further in this thesis.

Clearly, the memory requirement of RT-OSGi applications will be much lower than applications using the aforemntioned alternative forms of GC configuration analysis because the GC is assigned more CPU time than the alternative approaches thus the GC cycle length and free memory requirement of applications will be significantly reduced (because there will be less releases of threads during a GC cycle and thus there will be less accumulation of garbage).

The downside of the RT-OSGi approach is that finding the maximum amount of CPU-time that can be allocated to the GC while preserving application schedulability is quite computationally expensive, whereas the alternative approaches avoid this computationally expensive procedure. However, since the GC reconfiguration analysis will take place at the same time as the CPU

admission control discussed in Chapter 5, the thread executing the life cycle operations and thus GC reconfiguration analysis will not have real-time constraints because, as discussed, without making unrealistic reservations, the exact schedulability analysis's WCET cannot be determined.

The GC reconfiguration analysis used in RT-OSGi is now discussed in more detail. The notation used in the reconfiguration analysis is given in Table 6.1

Symbol	**Explanation**
C_i	Thread i's computation time (ms)
T_i	Thread i's period (ms)
D_i	Thread i's deadline (ms)
A_i	Thread i's memory allocation per period (MB)
R	Size of root-set (bytes)
R_{GC}	Time taken to complete a GC cycle
τ	Set of tasks
n	Number of application threads
H	Heap size (H/2) = semi space size (MB)
T_{GC}	Period of GC thread's controller thread (ms)
C_{GC}	Budget at which GC thread can run at high priority for (ms)
W_{GC}	The amount of GC work (ms)
M	Application memory requirement
c_1	Cost of scanning a word on the target hardware (ns)
c_2	Cost of copying a single byte of memory on the target hardware (ns)
c_3	Cost of initialising a single byte of memory on the target hardware (ns)

Table 6.1 Reconfiguration Analysis Notations

6.3.1.1 Estimating Garbage Collection Work

At admission control time i.e. when the application is reconfigured, the increase in GC work must be estimated. The GC work can be decomposed into three parts, the cost of reference traversal (scanning—including root-set and live object), the cost of object evacuation (or copying), and the cost of memory initialisation.

Reference traversal is concerned with the cost of identifying garbage (or non-live memory). This is achieved by scanning the root-set (R), which consists of global

and local variables, identifying references to objects, and scanning those objects to find references to other objects etc. Once all of the references are traversed, any objects with no references to them are considered garbage and the memory they occupy can be reclaimed. Clearly, the cost of reference traversal depends on the number of reference fields of scanned objects. Since the size of a reference field typically equals the `sizeof(word)`, the number of words comprising the maximum live objects and root-set gives an upper bound on the number of reference fields to scan. This is used (along with the time it takes on the target hardware to scan a single word (c_1)) to estimate the cost of reference traversal.

The cost of evacuating memory is simply the maximum amount of live memory multiplied by the cost of copying a single byte of memory (c_2). Finally, it is assumed that the cost of initialising memory is half the heap size multiplied by the cost of initialising a single byte of memory on the target hardware (c_3). The reason that half the heap size is assumed in the calculation is because the maximum amount of memory that may need initialising is an entire semispace, which is half the heap size.

As a note, the size of roots (R) is determined as the sum of the size of local and global variables. In Java, an upper bound on the size of local variables can be determined as the sum of each threads stack size, which is configurable in the JVM. Static analysis would be required to determine the size of global variables. As an additional note, Kim et al include the cost of barrier processing in the GC cost calculation. However, the barrier processing technique discussed in Kim is different to that supported by Sun Java RTS and thus it is not included. It is likely to be a relatively insignificant term in the equation in any case. For simplicity, unlike Kim, the worst case live memory of an application is taken to be the sum of the memory allocation of each thread, that is, it is assumed that all of the memory allocated by each thread per period is live. Finally, an upper bound on the amount of memory to be initialised is assumed, which is half the heap size. Kim et al use the GC cycle memory requirement (M) however there is a circular dependency: W_{GC} depends on M which depends on R_{GC} which depends on W_{GC}.

The equation used to derive an estimate of GC work (W_{GC}) is given in Equation 6.2. As a note, it is the scanning and copying phases of garbage collection which are the major factors in determining the GC work. The time taken in these phases is dependent on the amount of an application's live memory and not on the amount of dead (or garbage) memory.

$$W_{GC} = c_1\left(\frac{R+\sum_{i=1}^{n} A_i}{sizeof(word)}\right) + c_2\sum_{i=1}^{n} A_i + c_3\left(\frac{H}{2}\right) \qquad 6.2$$

Equation 6-2 Estimating GC Work [139]

In Equation 6.2 it is pessimistically assumed that a thread's live memory equals the memory allocated by the thread (A_i), the implication of this is that the amount of GC work will be overestimated and thus the GC will be assigned more CPU-time than is necessary to complete a garbage collection cycle. The reason for this assumption of allocated memory equating to live memory is because lifetime analysis (which is used to determine whether an object is live or not) is well known to be a very difficult task for developers to perform. Instead, memory profiling can be used along with memory allocation contracts in an identical way to the execution-time profiling and service execution-time contracts are used in WCET analysis as discussed in Chapter 4.

In Equation 6.2, it is also assumed for simplicity that the total live memory allocated by the application is the sum of the memory allocated in one release of each application thread i.e. it is assumed that after the first period, any memory that a thread allocates is garbage. The result of this is that the total live memory allocated by the application does not increase regardless of the number of releases of threads during a GC cycle. It is important that the total memory allocated by an application does not grow beyond that specified in Equation 6.2. If the total live memory allocated by the application does increase beyond the value used in Equation 6.2, the GC reconfiguration analysis becomes invalidated because the GC work, cycle length, and associated application free memory requirement will increase beyond what was estimated. As a result, the application may have falsely passed the memory admission control and memory exhaustion may occur.

From the discussion above, it is clear that allowing threads to allocate live memory in multiple periods whilst using Equation 6.2 may cause memory exhaustion because the application's memory allocation requirement will be greater than that assumed in Equation 6.2. Therefore if it is a requirement for the application's threads to allocate live memory in more than just their first release, Equation 6.2 can easily be modified to accommodate this. This is achieved by replacing the "sum of one release of each application thread live memory" term in Equation 6.2 with the worst-case live memory of the application in a steady state (i.e. whatever the live memory will be once the application threads stop allocating live memory).

Finally, from the discussion about estimating the amount of GC work required, it is clear that various pessimistic assumptions are made in the estimation process. However, this pessimism is necessary as it allows the GC cost to be estimated without having knowledge of the application's object graph; such knowledge is unlikely in RT-OSGi and SOA in general, where services are provided by third parties. As a result, the degree of pessimism has not been determined in this thesis, although as future work, this could be investigated along with the idea of using code annotations in services in order to closer model the actual object graph of RT-OSGi applications.

6.3.1.2 Calculating Garbage Collector Parameters

As the behaviour of the Sun Java RTS's GC has been modified to provide time-based GC, the only parameters that are required are computation time, period, and deadline.

The GC budget (C_{GC}) is calculated by starting with a low base value and iteratively increasing it until the GC CPU utilisation (cost divided by period) is so large as to cause excessive interference to application threads causing the system to become unschedulable. C_{GC} can be determined using Equation 6.3. Note that in Equation 6.3, 1 is the highest priority.

$$C_{GC} = \max\left\{ x \mid \forall t \in \tau_{\{i=0...n\}} : \left\lceil \frac{T_i}{T_{GC}} \right\rceil x + \sum_{j=1}^{i} \left\lceil \frac{T_i}{T_j} \right\rceil C_j \leq D_i \right\} \quad 6.3$$

Equation 6-3 Calculating GC Computation-Time (C_{GC})[139]

The GC period/deadline (T_{GC}) is assigned to be equal to the application thread with the smallest period, and thus, the GC thread will be highest priority according to Rate Monotonic priority assignment. The reason why the GC must run at the highest priority is because the adapted behaviour of the Sun Java RTS GC is to provide time-based GC (as explained earlier in this section), which typically requires the GC to run at the highest priority. Therefore the GC reconfiguration analysis presented here must also assume this.

According to Rate Monotonic priority assignment, the GC's period could be arbitrarily smaller than the application thread with the smallest period, and still have a higher priority than any application threads. Therefore, ideally, in order to compute the GC's period, the period of the GC would be iteratively decreased (much like the GC cost is iteratively increased in Equation 6.3). The result of this combined with Equation 6.3 would essentially find the maximum CPU utilisation that can be assigned to the GC without breaking application schedulability. This would be beneficial because it would effectively maximise the chance of RT-OSGi components passing admission control. Finding the maximum CPU time available to the GC while ensuring application scehdulability means that components never fail admission control on the grounds of their effect on increasing the amount of CPU time required to perform GC. Also, as discussed in Section 6.4, the more CPU-time can be allocated to the GC, the less free memory is required to support the application and therefore the more likely it is that the component will not cause garbage related memory exhaustion and will therefore pass admission control related to the memory resource. Compare this with the alternative approaches of either assigning the GC a fixed large share of the CPU utilisation e.g. 50%, or with the approach of assigning the GC the minimum amount of CPU time possible such that memory does not become exhausted. In the former case, the GC would likely cause components to fail schedulability analysis and thus reduce the total number of components which

can be deployed in RT-OSGi. In the latter case, the GC cycle (discussed in Section 6.3.2.3) which be extremely long and thus a lot of garbage will consume memory (although not exhaust it completely) such that the memory resource is wasted.

Although it would be beneficial to find the maximum amount of CPU time that can be allocated to the GC while maintaining application schedulability, it is too time consuming to try and find the exact value (i.e. both the minimum period and maximum cost for the GC) in dynamic environments such as RT-OSGi. By iteratively increasing the GC cost and keeping the GC period static, it is fairly quick to compute an approximation of the maximum CPU utilisation which can be assigned to the GC thread whilst keeping the application schedulable. Of course, it does not calculate the true maximum since the T_{GC} is not being iteratively decreased. In addition, the granularity of the increment of x in Equation 6.3 may also mean that a C_{GC} smaller than the maximum is being calculated. As a note, regardless of whether the maximum or an approximation of the maximum GC CPU utilisation is calculated, the CPU utilisation may be so small as to be ineffective. For example, if the system is heavily loaded with application components, there may be so little CPU time left over for the GC thread such that the context switch and other overheads may have an overruling effect. It is therefore necessary to check that the GC parameters computed are above what is considered an effective threshold i.e. the C_{GC} and T_{GC} parameters calculated act as an acceptance test for memory admission control. The performance of the GC parameter selection process is explored further in Chapter 8.

6.3.1.3 Estimating Garbage Collection Cycle Time

The GC cycle time is the time it takes the GC to complete all of its work i.e. the work estimated using Equation 6.2. A new GC cycle starts each time a complete garbage collection traversal has been completed and all of the reclaimed memory

blocks have been made available for allocation (e.g. after all reachable objects have been evacuated by a copying collector and the previously occupied memory has been released). During a GC period (T_{GC}), the GC thread can only perform an amount of work equal to the C_{GC} (calculated using Equation 6.3). Therefore a number of GC periods will be required to complete a GC cycle. The amount of time it takes (The GC response time – R_{GC}) can be determined using Equation 6.4.

$$R_{GC} = T_{GC} + \frac{W_{GC}}{C_{GC}} * T_{GC} \qquad 6.4$$

Equation 6-4 Determining the GC Cycle Time [139]

6.4 Memory Admission Control

As discussed in Chapter 5, admission control is a means of controlling the system load by using acceptance tests to filter requests for deployment; only entities passing the acceptance test may be deployed. In Chapter 5, it was made clear that overloading the CPU may cause violation of the timing constraints of real-time threads. However, as discussed in Section 6.2, the garbage collector may also cause timing violations for real-time threads either, indirectly, by failing to prevent memory exhaustion due to an accumulation of garbage, or because of its direct effect on real-time threads due to interference or increasing the WCET of application threads. In either case then, it is necessary to further extend the RT-OSGi life cycle operations so as to take the effects of GC and free memory into account.

To take memory issues into consideration during execution of the admission control of the life cycle operations, it is proposed that the Equations 6.2—6.4 (from Section 6.3) are used. As components undergo installation, Equation 6.2 is used to determine the new total amount of GC work that is required. Equation 6.3 is then used to determine the new maximum amount of CPU time that can be assigned to the GC thread, taking into account the fact that the new threads and existing threads must remain schedulable and not miss deadlines. The GC cycle

length is then calculated using Equation 6.4, and the amount of memory allocated by application threads during a GC cycle is estimated. If the free memory in the system is less than the application's memory requirements during a GC cycle, it is inevitable that memory exhaustion will occur. The reason for this is because, in the class of GCs known as Copying GCs (as with Sun Java RTS), garbage memory is not made available for use again until after the GC has completed its cycle. Therefore it is known that the free memory available in the system will not increase until after a GC cycle has completed.

Equation 6.5 is used as an acceptance test for memory admission control during application reconfiguration. It gives a safe free-memory threshold, if there is at least M free-memory in the system, the application is guaranteed to not experience garbage-related memory exhaustion, and the application can therefore be reconfigured. The reason for this guarantee is because M is the maximum amount of memory that the application requires for allocation before the GC cycle completes and releases the garbage memory allocated in the previous cycle. If on the other hand there is insufficient free memory, the application reconfiguration must be rejected because the maximum amount of CPU-time that the GC can be assigned without making the application unschedulable is insufficient to recycle memory at the required rate. Note how this behaviour differs from that of the default behaviour of Sun Java RTS. The Sun Java RTS GC without time-based behaviour and the associated reconfiguration analysis at admission control-time would always permit a RT-OSGi application to be reconfigured even if it means that the GC would eventually make the application unschedulable by running at a priority higher than one or more application threads for as long as necessary to return the level of free memory above the safe threshold.

If application reconfiguration passes the free memory acceptance test and the GC parameters acceptance test (discussed Section 6.3.1.2), then the GC can finally be reconfigured to run at the new pace dictated by the parameters previously calculated. More specifically, on the time-based modifications to Sun Java RTS's GC, the GC controller thread's period is set to T_{GC}, and the time for

which the GC controller thread permits the GC thread to run at high priority for is set to C_{GC}.

$$M = 2\left(\sum_{i=1}^{n}\left(\left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil + 1\right)A_i\right) + \sum_{i=1}^{n} A_i\right) \qquad 6.5$$

Equation 6-5 Determining Free Memory Requirement of the Application [139]

As a note, in Equation 6.5, the application's memory requirement consists of the memory allocated by one release of each application thread, which will remain live. It also consists of the number of releases of each application thread during the GC cycle and the amount of memory allocated during each release, this is assumed to be garbage memory. The memory requirement is twice the sum of the above two constituents because the GC is a copying GC and therefore uses two semispaces. While a GC cycle is in progress, live memory from the semispace used during the previous GC cycle is copied into the new semispace. Concurrently, in each release, application threads will be allocating garbage memory in the current semispace. Once the GC has finished copying live memory into the current semispace, the GC can initialise the old semispace to make it available for allocation again. At this point the GC cycle completes and the roles of the semispaces are reversed. Only at the end of the next GC cycle will the garbage memory allocated during the current cycle be available for reclamation. Hence the memory requirement must cover two GC cycles worth of allocation.

As a further note, when performing GC reconfiguration analysis, it is clear that the GC parameters (T_{GC} and C_{GC}) need to be calculated based on the entire task set in order to find the maximum amount of time that can be assigned to the GC while accommodating the tasks of the component undergoing admission control and while maintaining schedulability. However, it may not at first be clear why the rest of the reconfiguration analysis cannot be performed by using the task set of the component undergoing admission and by adding this value to whatever previous values were calculated for the existing task set which is already deployed.

The reason why this is not the case is that in order to accommodate the tasks of the component undergoing admission control, the GC parameters may be changed such that the GC is allocated less CPU time than it was previously allocated. As a result of such a change, the R_{GC} and free memory requirement for the existing task set will increase. Thus unless the GC parameters remain the same at admission control time, it is not possible to simply calculate the increase in GC work and free memory requirement of the application based solely on the existing GC reconfiguration values and the tasks belonging to the component undergoing admission control. Therefore, the GC reconfiguration values are re-computed every time a component undergoes admission control.

One issue with calculating the GC reconfiguration requirements rather than the change in requirements is in regard to the application free memory requirement. The free memory check will be pessimistic because it won't automatically take the memory already allocated into consideration. For example, if the free memory requirement of an application is 30 MB before a component undergoes admission control and 50MB after, the free memory test will essentially determine whether there is 50MB free memory available in the system. However, the application may have allocated 30MB of the 50MB already and thus the free memory requirement will only be 20 MB not 50MB. However, it is difficult to determine how much of the memory requirement of the previous application configuration (i.e. the configuration before the most recent request for reconfiguration and subsequent admission control) the application has already allocated. This issue can be solved by having RT-OSGi keep track of the memory requirement determined for the application at the last point in which a component underwent admission control, along with the free memory available in the system at that point in time. In this way, it is possible to determine how much of the memory required by the previous application configuration has already been allocated. For example, assuming the previous scenario whereby before a component undergoes admission control the free memory requirement of the application and the free memory available in the system equal 30MB and 500MB respectively, and 50MB and 470MB respectively after the admission control for the component, then the increase in free memory requirement of the application can be calculated as the new application free memory requirement

(M-in GC reconfiguration analysis) (50MB) – (old free memory available (500MB – new free memory available (470MB)) = 20MB free memory requirement.

6.5 Example of Applying Garbage Collection Reconfiguration Analysis

As an example of using the previous analysis for reconfiguring the GC in the presence of changes to the application i.e. when new components wish to be deployed, consider the application with the temporal specification detailed in Table 6.2.

Thread	**C (ms)**	**T (ms)**	**D (ms)**	**A (MB)**
T1	1	10	10	1
T2	2	8	8	0.5
T3	0.3	5	5	0.5

Table 6.2 Server Temporal Specification

T1 and T2 are already deployed. The following values are obtained from using the garbage collection related equations: GC work W_{GC} = 28.6ms, GC period = 8ms, GC budget = 2.5ms, GC cycle time R_{GC} = 94.6ms, and a GC cycle memory requirement of 38 MB. It is then assumed that a new component needs to be deployed with a single thread (T3).

Before deploying the component containing T3, Equations 6.2-6.5 are used in order to generate the necessary GC configuration and to perform admission control for T3. Firstly, the W_{GC} is calculated. It is assumed that each thread has a stack size of 512KB and therefore the root-set (R) equals the sum of the three threads' stacks which equals 1.5MB, It is assumed that there are no static variables. It is also assumed that the word size of the target hardware architecture is 32bits (4 bytes), and the heap size is 500MB. Finally it is assumed that the cost of scanning a word (c1) equals 1 nanosecond, the cost of evacuating a byte (c2) equals 2 nanoseconds, and the cost of initialising one byte of memory (c3) is 0.1 nanoseconds. Using Equation 1,

W_{GC} = 1 * (3500 000 / 4) + 2 * 2000 000 + 0.1 *(500 000 000 / 2)
=29.884 ms

In terms of GC parameters, the GC period is assigned a period equal to the highest priority thread T3, so T_{GC} = 5ms. Using Equation 2:

X=0.5 0.8 <= 5 3.6 <= 8 **6.6 <= 10**	*X=1* 1.3 <= 5 4.6 <= 8 **7.6 <= 10**	*X = 1.5* 1.8 <= 5 5.6 <= 8 **8.6 <= 10**
X = 2 2.3 <= 5 6.6 <=8 **9.6 <= 10**	*X = 2.5* 2.8 <= 5 7.6 <= 8 **10.6 > 10**	**C_{GC} = X = 2ms**

To Calculate the GC cycle Equation 3 is used:
= 5 + (29.884 / 2) * 5
= 79.71ms

Finally to compute the memory required during a GC cycle Equation 4 is used:
2 * (9 + 8.5 + 5.5 + 2)
= 50 MB

For admission purposes, the GC parameters (C_{GC} and T_{GC}) selected keep the application threads schedulable and at the same time provide GC with enough CPU utilisation (two divided by five) such that the context switch overhead is negligible compared with the CPU utilisation available for GC. For this reason the thread T3 passes the CPU acceptance test. It is also assumed that there is more than the required 50 MB of free memory available and therefore the thread T3 passes the free memory acceptance test. Assuming the component passed the CPU

admission control discussed in Chapter 5, the component containing the thread T3 can now be deployed.

6.6 Memory Allocation Enforcement

If threads allocate more memory than the amount used for calculating the GC work (Equation 6.2), then the rest of the GC reconfiguration analysis discussed above becomes invalidated. More specifically, the GC cycle length will be longer than was estimated in Equation 6.4, and as a result, the free memory requirement of the application will be greater than that estimated in Equation 6.5. The implication of this is that components may be erroneously admitted into the system because the actual free memory will be less than the amount used in the acceptance test for component deployment. As a result, threads may block due to memory exhaustion. In order to prevent this situation, the memory allocation (as specified in Equation 6.2) of threads per period is enforced.

To achieve memory allocation enforcement, the memory allocation monitoring of the RTSJ is used. Such allocation monitoring is provided by using the RTSJ class `MemoryParameters`. This class allows a bound to be placed on the amount of memory a thread can allocate during its lifetime. The RTSJ intended for this functionality to be used in the context of scoped memory. However, it is argued that memory allocation enforcement is also necessary for heap memory, and for this reason, most RTSJ implementations allow the class to be used to enforce heap memory allocation in addition to the intended scoped memory allocation. This fact is exploited for the purposes of providing memory allocation enforcement in the context of heap memory in RT-OSGi.

In RT-OSGi, the memory allocation enforcement is extended such that it appears that a thread's memory bound is per period rather than a lifetime total. For example, this is achieved by periodically increasing the lifetime total permitted allocation of a thread by the amount of allocation per period. Also, the memory allocation overrun handling is extended by creating subclasses of the RTSJ

classes which implement the Schedulable interface specifically for RT-OSGi. Upon detecting a memory allocation overrun, three models of handling the overrun are proposed in RT-OSGi:

1) Hard enforcement – provides the application with no means of recovery and simply discards the current period by calling `waitForNextPeriod()` which will block the thread until its next period.

2) Soft enforcement – fires an asynchronous "overrun memory budget" event. The application adds a handler to the event so that on memory overruns, the application is notified and can attempt to recover from the error. Of course, this recovery phase needs pre-planning since the thread is not permitted to allocate any more objects in the heap!

3) Hybrid enforcement – threads have two memory budgets, a soft and a hard memory allocation budget. Overrunning the soft budget will cause the event handling mechanism to be used. Continuing to allocate memory despite the soft budget overrun may cause a hard budget overrun. Overrunning the hard budget causes the thread to be blocked for its next period.

Figure 6.5 shows an example of implementing such memory allocation enforcement using the memory allocation monitoring of an implementation of the RTSJ. As discussed shortly, this model is not used directly by application developers but is integrated into the class hierarchy used by RT-OSGi developers such that the memory allocation enforcement is provided by RT-OSGi automatically.

In Figure 6.5, the typical structure of a periodic thread is given. The thread executes in a continuous loop performing application logic and then calling the RTSJ's `waitForNextPeriod()` which blocks the thread until the start of its next period. The periodic thread then repeats the application logic that it executed in its previous period. To support memory allocation enforcement, at the beginning of its period, the thread's memory allocation budget is increased from an initial value of zero by an amount equal to the worst-case amount of live memory required by thread acquired from memory profiling. The thread then

attempts to perform its application logic and call to `waitForNextPeriod()`. If during memory allocation monitoring the real-time JVM detects that the thread has exhausted its memory allocation budget, the semantics of the `MemoryParameters` class are to have the JVM throw an OutOfMemoryError which is caught by the application thread. The thread then provides the soft memory allocation enforcement by firstly firing a memory allocation budget overrun asynchronous event, from which a user-provided asynchronous event handler can then be released to perform some corrective action. After this point, the thread can the safely provide hard memory allocation enforcement by calling `waitForNextPerid()`. At the start of the next period, the thread then sets its memory allocation budget for its period. Such periodic calls are necessary because the semantics for the RTSJ method `setMemoryParameters(…)` are to set a memory allocation budget for the lifetime of the thread (i.e. a budget from starting the thread until thread termination) rather than a per period memory allocation budget as is required in RT-OSGi to support GC.

```
public void run()
{
    while(true)
    {
        setMemoryParameters(new MemoryParameters(…));
        try
        {
            //application logic
            waitForNextPeriod();
        }
        catch(OutOfMemoryError oome)
        {
            //memory allocation enforcement
            //Soft enforcement
            softBudget.fire();
            //Hard enforcement
            waitForNextPeriod();
        }
    }
}
```

Figure **6**.5 Example of Memory Allocation Enforcement

The code sample in Figure 6.5 is only meant to be illustrative of the memory allocation enforcement model for RT-OSGi. Application developers do not have to implement the memory allocation enforcement model themselves (as Figure

6.5 might suggest). Rather, this model is integrated with the asynchronous termination model of RT-OSGi which was discussed in Chapter 5. The RT-OSGi class (OSGiRTT) encompasses the above memory allocation enforcement model in its run method, calling the `doInterruptible(..)` method of application defined subclasses of OSGiRTT in a try block rather than calling the application logic directly (as is the case with Figure 6.5). In order to support user-defined asynchronous event handlers for recovering from memory allocation budget overruns, OSGiRTT is extended with a method to allow subclasses to register their event handler with the OSGiRTT defined `softBudget` asynchronous event. As a result, RT-OSGi essentially provides memory allocation enforcement on behalf of the application developer so as not to complicate the programming task of developers.

As a note, it is evident from Figure 6.5 and the discussion associated with it that the RT-OSGi memory allocation enforcement model relies on calls to `waitForNextPeriod()`. As this method can only be called by periodic threads, it means that sporadic threads are unable to currently benefit from such memory allocation enforcement. However, the next release of the RTSJ will abstract away from `waitForNextPeriod()` and provide a more general method `waitForNextRelease()` which is applicable to both periodic and sporadic schedulable entities. In the case of asynchronous event handlers, the memory allocation enforcement model is integrated with an asynchronous termination model similar to the one discussed for real-time threads in Chapter 5. The differences between the memory allocation enforcement model for asynchronous event handlers and real-time threads is that the asynchronous termination is based around the handler's handleAsyncEvent method rather than the real-time thread's run method, and the memory allocation budget replenishment is based around the handler's handleAsyncEvent rather than the real-time thread's waitForNextPeriod method.

Even with the approach to memory allocation enforcement previously discussed, it is not possible to guarantee that memory exhaustion will not occur. The reason for this is because our model assumes that the threads in components do not

allocate memory in each period that can never be reclaimed, for example growing a dynamic data structure and never nullifying the reference to it. Since such a dynamic data structure will never be garbage collected, it will eventually consume the entire memory. Of course this same problem exists independently of garbage collection.

To be able to prevent memory exhaustion requires that memory consumption and not memory allocation be enforced. The distinction between consumption and allocation is that consumption takes memory deallocation into consideration, such that when a thread has reached its memory consumption budget, some of the allocated memory must be deallocated before further memory can be allocated. Unfortunately, it is difficult to provide memory consumption enforcement as it requires the GC to have knowledge about which threads created garbage so that once garbage is collected, the relevant thread can have its memory budget replenished by the amount of garbage collected. No GC in widespread use provides such functionality.

Memory consumption enforcement could, of course, be provided at the application level. For example, RT-OSGi could maintain a lookup table of thread identifiers and their respective memory consumption budgets. When a thread creates an object, the identifier of the thread which created the object (i.e. allocated some memory) could be stored by the object as it is constructed. Using the stored identifier of the object creating thread, the object could then lookup the identifier in RT-OSGi's thread lookup table. After a match is found in the lookup table, the thread's memory consumption budget could be decremented by a value equal to the amount of memory occupied by the object created by the thread. This value can be found by using the `SizeEstimator` class of the RTSJ.

When an object is no longer referenced (i.e. it becomes garbage), before the GC reclaims the memory associated with the object, the JVM will execute the object's `finalize()` method. This method allows the object to perform some cleanup actions before it is discarded. Therefore, the `finalize()` method can

be used by the object to use the stored thread identifier to look up the thread which created the object. After identifying the thread, the thread's memory budget stored in the table can be modified such that it is replenished by an amount equal to the amount of memory occupied by the object. To reduce the burden on the application developer, the thread identifier caching by objects could be performed by the RT-OSGi Framework through byte-code rewriting rather than involving the developer in this process.

While providing application level memory consumption enforcement is a possibility, there are a number of issues which potentially make it impractical. Firstly, `SizeEstimator` only estimates the memory requirement of the object itself. It does not include memory required for any objects allocated at construction time. Thus providing an automated approach to recursively applying `SizeEstimator` may be both complex and time consuming. Secondly, the memory consumption enforcement may be either ineffective or may interfere with the timing constraints of real-time threads.

Application level memory consumption enforcement may be ineffective if the JVM thread responsible for calling the `finalize()` method of objects provided by the JVM runs in the background i.e. at a priority below those used by real-time threads. In this case, the memory consumption enforcement will have a delayed affect since the accounting of memory deallocation will occur with a potentially considerable delay as the finalizer thread waits to become the most eligible thread for execution by the scheduler. As a result, a thread's memory consumption budget may be inappropriately deemed exhausted when in fact, the thread garbage memory of the thread is waiting to be accounted for by the low priority finalizer thread.

If the finalizer thread executes at a priority higher than all application real-time threads such that the memory consumption budget accounting occurs with little delay, it may affect the timing constraints of the application threads. The finalizer thread will pre-empt all application threads and thus it must be accounted for

during schedulability analysis to help guarantee the timing constraints of the real-time threads of the applications.

For these reasons, application level memory consumption enforcement is not presently used in RT-OSGi. A more appropriate approach to memory consumption enforcement in RT-OSGi is to utilise a partitioned heap. In this way each component in RT-OSGi is assigned its own local heap which threads in other components are unable to allocate in. In this way there is no explicit need to account for memory allocation and deallocation since a component's memory consumption budget clearly becomes the size of its own heap. Although more suitable than application level memory consumption enforcement, providing a partitioned heap is challenging in the case of RT-OSGi. The reason for this is because of the component-based and service-oriented nature of RT-OSGi applications. RT-OSGi components should not be spatially isolated as this defies the nature of such applications. In RT-OSGi components provide Java packages to one another and also share services). Using a partitioned heap would require changes to the service model of OSGi by replacing synchronous service calls with asynchronous ones such as by using Java Remote Method Invocation (RMI) to enable communication between components executing with their own local heap spaces. Since it is undesirable to re-design RT-OSGi in such a way, partitioned heaps are not discussed further in this thesis, and at least for now, RT-OSGi supports memory allocation enforcement but does not support memory consumption enforcement. Memory consumption enforcement is considered future work. Partitioned heaps will also be considered as future work. The reason for this is because, despite the fact that the partitioned heaps would make communication between RT-OSGi components more complex and would require major design changes to RT-OSGi, they would allow components to have memory reservations. The advantage of this is that it would allow components to be successfully deployed with no/soft real-time guarantees but without affecting the temporal constraints of other deployed components. For example, it would allow threads in soft/non-real-time components to block on allocation when there is insufficient memory in their heap partition until memory is reclaimed in that partition, without affecting the memory reservations of other components. This is not currently possible in RT-OSGi.

6.7 Summary

In real-time systems the real-time garbage collector (RT-GC)) must recycle memory often enough to prevent memory exhaustion without disrupting the temporal predictability of real-time threads. The RTSJ does not specify any particular RT-GC algorithm and therefore current major RTSJ implementations use different RT-GC algorithms. However, these RT-GCs may cause timing faults in dynamically reconfigurable real-time systems such as RT-OSGi applications either through being unable to prevent memory exhaustion through the accumulation of garbage memory, or through increasing the WCET of the application's heap-using real-time threads. To deploy dynamically reconfigurable real-time systems, it is necessary to have a RT-GC that can adapt its rate of garbage collection in accordance with the dynamic reconfiguration of the application to prevent the aforementioned issues. In RT-OSGi, this is achieved by implementing a dynamically reconfigurable time-based GC using the dynamically reconfigurable GC provided by the Sun Java RTS JVM and by providing GC reconfiguration analysis. Furthermore, based on the reconfiguration analysis, memory admission control is provided to ensure that application reconfiguration may only take place when the GC will not cause timing faults in application threads. The GC reconfiguration analysis and memory admission control take place after CPU admission control (discussed in Chapter 5) when the install life cycle operation is invoked on a component If the component passes both the CPU and memory admission control, the GC is reconfigured and the component is deployed. Finally, to support the GC once the application reconfiguration passes the memory admission control, the memory allocation budgets of application threads are enforced in RT-OSGi.

7

Case Study: Chronic Disease Management

7.1 Introduction

In this chapter, a case study application is introduced which gives motivation for and demonstrates the expressive power of using RT-OSGi. RT-OSGi enables service-oriented component-based applications to be developed which have real-time capabilities i.e. RT-OSGi components can exploit the full power of the RTSJ such as real-time threads, fixed priority scheduling, and real-time garbage collection. In addition, RT-OSGi applications are also dynamically reconfigurable, which means that an application can be evolved/maintained/adapted without affecting the application availability. Therefore, the case study must have the following requirements:

- Real-time guarantees – RT-OSGi components can exploit full power of the RTSJ
- High system availability requirement
- Reconfiguration/evolution/maintenance
- Is not a safety critical system – it would be difficult to prove the correctness of dynamic systems such as RT-OSGi

A chronic disease management case study which meets these requirements is now discussed.

RT-OSGi can be applied to potentially improve the prognosis of chronic disease by helping patients to manage their disease. The life expectancy of patients may be improved by using RT-OSGi for both the early detection of short term acute complications of chronic disease, as well as the early detection of risk factors that may contribute to the long term chronic and acute complications of disease. In this thesis chapter, the focus is on the early detection of short term acute complication of chronic disease.

7.1.1 Introduction to Short Term Acute Complications in Chronic Disease Management

In many chronic diseases, there is a risk of acute complications relating to either the disease itself, or, from the medication used to help manage the disease.

As an example of acute complications from the medication used to treat a chronic disease, consider diabetes mellitus (simply referred to as diabetes from here on). In type 1 diabetes (insulin dependent), in order to control the level of blood sugar, a patient must inject insulin. However, one of two acute complications can occur related to the injection of insulin:

1) Excessive insulin injection will cause blood sugar levels to drop below the normal level causing hypoglycaemia (low level of blood sugar). Severe hypoglycaemia may lead to coma, seizures, or even brain damage and death.
2) Insufficient insulin injection may result in insulin deficiency which promotes gluconeogenesis, glycogenolysis, and ketone body formation. In excess, this may cause hypotension (low blood pressure), shock, and death.

As an example of acute complications relating to the disease itself, consider the respiratory disease Asthma. Asthma is a common inflammatory condition of the lung airways characterised by the symptoms of chest tightness, coughing, and shortness of breath. Acute exacerbations of symptoms (known as asthma attacks)

occur spontaneously, and as with the acute complications of diabetes, patients suffering from asthma attacks may require hospital admission.

It is of great benefit if the onset of these potentially life threatening acute complications of chronic disease can be detected at an early stage by a real-time monitoring application that corrective action can be taken to help prevent a medical emergency from arising.

In the remainder of this chapter, such a real-time monitoring system is proposed for simultaneously managing multiple chronic diseases. Although the monitoring system is applicable to a number of chronic diseases, for simplicity a concrete example of monitoring for hypoglycaemia in insulin-dependent diabetics is discussed.

7.2 Application Requirements

It is possible for the acute complications of both diabetes and asthma to be detected early if one or more of the patient's vital signs are being continuously monitored. For example, before the onset of the acute complication hypoglycaemia in diabetes, there is a significant change in a patient's brain's electrical activity, which is measurable with Electroencephalography (EEG). With hypoglycaemia, there are also changes in heart rate, blood pressure, and respiratory rate. Similarly, before an Asthma attack, there is a significant change in oxygen saturation detectable via a pulse oximeter. RT-OSGi can be used to monitor a number of wireless wearable patient vital signs sensors, and can use complex event processing to help infer when an acute complication of chronic disease is occurring. Such a monitoring application has reconfiguration, high availability, and real-time requirements as discussed below.

7.2.1 Application Reconfiguration Requirement

A patient's health is dynamic, over time it will change. Therefore, rather than having isolated applications for managing different chronic diseases, it is more

desirable to have a single system (such as RT-OSGi) which can be reconfigured according to the patient's health status. There are a number of ways in which RT-OSGi in chronic disease management may require reconfiguration:

1. Removing Faulty Sensors – because vital sign sensors are wearable and will be in constant use by the patient, they are more susceptible to damage. Therefore the application may need to be reconfigured to replace faulty vital signs sensors.

2. Adding New Sensors – New chronic diseases from a number of causes, requires new sensors to be installed. Sensors may also need to be temporarily installed to aid in finding the correct dosage of a new medication, such as to find a dosage which balances adequacy of treatment with tolerable side effects. For example, when a patient requires anticoagulant medication, the patient may borrow a blood coagulometer from their health care provider and install it into the chronic disease management RT-OSGi application in order to balance the efficacy of reducing the likelihood of cardiovascular disease whilst minimising the risk of causing severe episodes of bleeding.

   Finally, sensors may be added and removed from the system as a means of minimising resource requirements. In RT-OSGi, it is possible to minimise the resource usage by minimising the component set currently active. This can be achieved by having the patient install additional (to the continuously active wearable vital signs sensors) sensor components when they want to take sporadic sensor measurements, or by having the monitoring application install additional sensor component on-demand, when complex event processing detects possible acute complication arising. For example, the application may dynamically install a peak flow sensor, blood pressure sensor, respiratory rate sensor, and blood glucometer depending on the acute complication detected. These sensor components can then be removed when they are no longer required.

3. Updating Complex Event Processing Component [144] – New chronic disease management may not require any new sensors, but rather new complex event processing rules for detecting new short term acute disease complications based on the existing set of sensors. That is, an update to the complex event processing component of the application for recognising a new complex event from the existing simple events.

As a note, real-time systems with high availability requirements may not be required to operate for 24 hours a day and may in fact have a natural down-time period during the day. It could therefore be argued that dynamic reconfiguration is not required in such applications and thus RT-OSGi is of little benefit to such systems since the maintenance and reconfiguration activities could be scheduled to take place during the hours of the day in which the application does not need to be operational. However, this is not always possible. In some applications such as the chronic disease management application proposed in this chapter, the reconfiguration/updates/maintenance cannot be postponed until the natural system downtime because the maintenance activity is urgent and the application will operate with a much lower utility while ever the maintenance activity is delayed. For example, consider the chronic disease management application when one of the vital sign sensors used to monitor various variables of a patient suffering with chronic disease(s) begins to malfunction. As a result of the malfunctioning sensor, the ability of the application to detect disease complications will be severely compromised, for example if an EEG sensor begins to malfunction, the ability of the chronic disease management application to detect hypoglycaemia is compromised. In this case, the maintenance activity to correct the EEG sensor issue cannot be delayed from the time of occurrence (say the early morning) until the next scheduled application down-time e.g. late evening. Similarly, the maintenance activity cannot be started immediately without dynamic reconfigurability since this would involve taking the application offline and making it unavailable for use. In terms of the chronic disease management monitoring application, this would mean that the ability of the application to detect other complications of disease(s) such as asthma attacks is compromised while the application is taken offline in order to correct the faulty

EEG sensor. Therefore the dynamic reconfigurability of RT-OSGi can be seen to be to essential to meeting the application requirements of the chronic disease management case study application discussed in this chapter of the thesis.

7.2.2 High Application Availability Requirement

The early detection of short term acute complications in chronic disease management has high application availability requirements.

With early short term acute complication detection, complications of disease can occur at any-time of day. The more hours of the day that the patient's vital signs are monitored, the more likely it is that acute complications can be detected early and corrective action taken to prevent a medical emergency from occurring.

As the vital signs sensors are wearable and wireless, they are not restrictive to the patient and should be worn throughout the day and in some cases during the night when the patient is sleeping. Night monitoring is particularly important in patients with diabetes and asthma. For example, because the patient is in a fasting state during sleeping, they are more likely to experience hypoglycaemia.

As discussed, with such high availability requirement of short term acute complication monitoring, the fact that RT-OSGi is dynamically reconfigurable is essential. Taking the system offline for reconfiguration (non-dynamic reconfiguration) would mean that the system is unable to monitor the patient's vital signs for the duration of the reconfiguration and thus if a patient is in the process of becoming acutely ill while the application is offline for reconfiguration, the vital signs monitoring will not be operational and will not be able to give early detection of such a situation. For example, if a diabetic patient has been recently diagnosed with asthma, the application must be reconfigured to start monitoring for acute exacerbations of asthma. However, it is extremely undesirable to have to take the system offline and thus have to stop monitoring for the acute complications of diabetes whilst the system is being reconfigured.

7.2.3 Real-Time Requirement

The process of monitoring vital signs is a real-time requirement. Periodic real-time threads are required to read the sensor values, and asynchronous events are required as part of complex event processing. Late sensor values will affect the application's ability to detect at an early stage short term acute complications in chronic disease management.

In terms of complex event processing, the process of recognising a complex event (i.e. an acute complications of chronic disease) and notifying the user has timing constraints. Clearly, detecting the complications late, or detecting them early but not notifying the patient until a much later stage compromises the correctness of the application.

7.3 Case Study Design

In this section, the design of the chronic disease management application case study is discussed.

7.3.1 Overview

Monitoring for short term acute complications in chronic disease management can be achieved by using RT-OSGi. The general idea is to continuously monitor the following patient vital signs: heart rate (using Electrocardiography (ECG)), oxygen saturation (using a pulse oximeter), body temperature (using a thermometer), and electrical activity of the brain (using Electroencephalography (EEG)). These vital signs can be recorded using wearable wireless sensors, which are recently emerging on the home-based healthcare market [145-147]. In the case where a change in a single vital sign is insufficient to indicate an acute complication of disease, complex event processing (CEP) can be used to infer disease complication based on combinations of values read from the multiple vital sign sensors. If complication is suspected, the patient can be notified via

their smartphone, and additional sensors can be dynamically installed to help the patient confirm the suspected diagnosis. Such early disease complication detection allows the patient to take some corrective action before a medical emergency occurs.

Figure 7.1 shows the general interaction between the patient's sensors, the RT-OSGi components, and the patient's smartphone. In Figure 7.1, the thermometer is denoted (TH), the pulse oximeter (PO), the EEG electrodes (occipital 1 and 2 (O1,O2) and temporal 1 and 2 (T1,T2)), and the ECG electrodes (right arm (RA), left arm (LA), and left leg (LL)). After reading the various sensor data, RT-OSGi performs some data processing, and communicates the result with the patient via their smartphone. The RT-OSGi components for the chronic disease management application are shown in Figure 7.1. The components in the figure along with their interactions will be discussed in more detail in the next section.

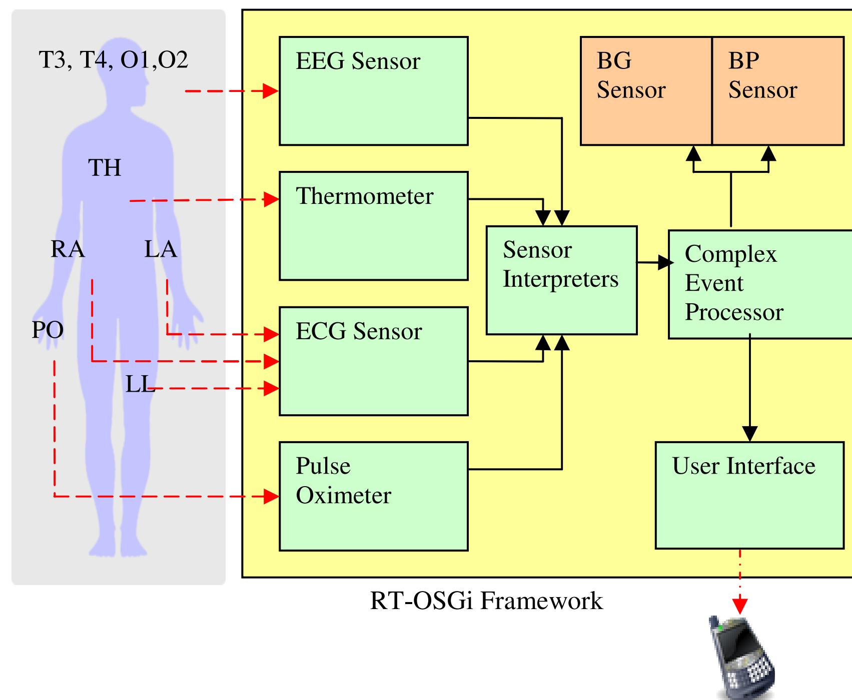

Figure 7.1 Interaction between the Patient's Wireless Wearable Vital Signs Sensors, the RT-OSGi Components, and the Patient's Smartphone

7.3.2 Sensor Components

In order to monitor vital signs, each vital sign has a corresponding RT-OSGi component created for it e.g. an ECG component for ECG monitoring, and a pulse oximeter component for pulse oximetry etc. Each sensor component consists of at least one implementation of a SensorService, the component may contain more than one implementation if monitoring a vital signs requires reading data from multiple pieces of hardware e.g. the three lead ECG component reads data from three different pieces of hardware (ECG electrodes) and therefore has three implementations of SensorService. Each sensor service implementation, through its getSensorValue() method, encapsulates the RTSJ code necessary to obtain the data from the sensor hardware attached to the patient. Each implementation of the SensorService service's getSensorValue() method assumes memory-mapped sensors with data and control registers. The sensor data access can be performed by using the RTSJ's raw memory access classes. An example of this is given in [148].

The SensorService service interface used by all sensor components is shows in Figure 7.2.

```
package uk.ac.york.casestudy.service.sensor;

public interface SensorService
{
    public float getSensorValue();
}
```

Figure 7.2 SensorService Service Interface

SensorService service implementations are registered in RT-OSGi as shown in Figure 7.3. The example shows the registration of a service implementation for one of the three pieces of ECG electrode hardware. The SensorService service implementation is registered with the sensor type as a service property, and service requesters use the sensor type property as part of service discovery. The benefit of using SensorService is that all sensors are accessed in a uniform way i.e. by calling getSensorValue(), regardless of the type of sensor being used.

```
SensorService ecgLA = new ECGElectrodeLA();
Properties props = new Properties();
props.put("sensorType","ECGElectrodeLA");
ServiceRegistration reg = ctxt.registerService(
  "uk.ac.york.casestudy.service.sensor.SensorService",ecgLA,props);
```

Figure 7.3 Registering an Implementation of the SensorService

The service implementation registration shown in Figure 7.3 makes one of the ECG electrode hardware available to other components in RT-OSGi. Figure 7.4 shows how the ECG service implementation can be discovered by any service requesters in RT-OSGi components.

```
ServiceReferences[] ref = ctxt.getServiceReferences(
     "uk.ac.york.casestudy.service.sensor.SensorService"
     ,"(sensorType=ECGElectrodeLA)");
if(ref != null)
{
     SensorService ecgLA = (SensorService) ctxt.getService(ref[0]);
}
```

Figure 7.4 Discovering an Implementation of SensorService

After registering one or more SensorService services, each sensor component creates an RTSJ periodic real-time thread which periodically calls their service implementation's getSensorValue() method in order to obtain data from their associated sensor hardware. After performing a small amount of computation on the sensor data, the periodic thread then passes the processed sensor data to a data buffer maintained by the sensor's corresponding data interpreter service registered by the sensor interpreter component. As an example of a sensor component's periodic thread, consider the ECG sensor component. The ECG component creates a periodic thread to obtain the sensor values from the three ECG electrodes via the associated service implementations. After doing so, the thread must then perform a few simple calculations in order to obtain what is known as the ECG lead values. In this case study a three lead ECG device is assumed and therefore the three lead values must be calculated from the raw data acquired from the three ECG electrodes. The three lead values are then passed to

the ECGInterpretation service registered by the sensor interpreter component. The periodic thread's run method is shown in Figure 7.5.

```
public void run()
{
    //get electrode values
    la = ecgLA.getSensorValue();
    ra = ecgRA.getSensorValue();
    ll = ecgLL.getSensorValue();

    //compute ECG lead values from electrode values
    lead1 = la - ra;
    lead2 = ll - ra;
    lead3 = ll - la;

    //pass three bipolar lead data to interpretation service
    ecgInterpretation.setLead1(lead1);
    ecgInterpretation.setLead2(lead2);
    ecgInterpretation.setLead3(lead3);
}
```

Figure 7.5 The ECG Interpreter Thread's Run Method

In terms of the temporal specifications of the sensor components' periodic threads, they all have small computation times as they are not computationally intensive threads, and all of the sensor threads except for the thermometer thread must obtain data samples from their associated sensor hardware at a relatively high frequency. As an example, again consider the ECG periodic thread. The ECG sensor measures the electrical activity associated with each heart beat, as the average person's heart rate is 70 beats per minute, measuring the changes in electrical activity during the various phases of each heart beat, the ECG electrode monitoring thread must sample at a high frequency, typically 500 MHz (2ms) is recommended by the medical research community. Therefore, the ECG sensor component's thread has a period and deadline of 2ms, and a computation time of 0.5 ms (given the small amount of computation that the thread must perform in each period). The temporal specification of all of the case study application's threads is shown in Table 7.1 in Section 7.4.4.

7.3.3 Sensor Interpreter Components

Each sensor interpreter component contains a service for interpreting the data sent from one of the four vital signs sensor components. As mentioned, each sensor component's periodic thread calls the corresponding interpreter component's sensor interpreter service in order to pass in the data values read from the sensor hardware. Each sensor interpreter component also contains a thread for executing its interpreter service. The role of these periodic threads is to analyse the data received from its associated sensor component, and if any significant trends are identified, an asynchronous event is fired. This event is then handled by the complex event processor component, which is discussed shortly. Figure 7.6 shows the ECG sensor interpreter component's ECGInterpretationImpl service, which is called periodically by the component's ECG interpreter thread. From this figure, it can be seen that the ECGInterpretation service uses the sensor data periodically sent from the ECG sensor component in order to calculate the patient's heart rate. If the heart rate is deemed either abnormally fast or slow, the complex event processor is notified via an asynchronous event.

```
package uk.ac.york.casestudy.impl.sensor;
import uk.ac.york.casestudy.service.interpreter.ECGInterpretation;

public class ECGInterpretationImpl implements ECGInterpretation
{
      ArrayList history = new ArrayList();
      public void analyseECG()
      {
            float rrInterval = rr – previousRR;
            float heartRate = 60 / rrInterval;
            history.add(heartRate);
            boolean tachycardia = (heartRate > 100) ? true : false;
            boolean bradycardia = (heartRate < 51) ? true : false;
            if(tachycardia == true)
            {
                  tachycardiaAsyncEvent.fire();
            }
            if(bradycardia == true)
            {
                  bradycardiaAsyncEvent.fire();
            }
      }
      …
  }
```

Figure 7.6 ECGInterpretation Service Implementation

In terms of the temporal specifications of the four sensor interpreter components' interpreter threads, the computation time, period, and deadline is much larger than that of the corresponding sensor components' threads. The reason for this is because the trend in sensor data is typically more helpful than looking at sensor readings in isolation thus sensor data analysis requires multiple sensor readings to have been buffered before any potential health risks can be identified. For example, the ECG Interpreter component's interpreter thread has a period much larger than that of the ECG component's sensor monitoring thread. The reason for this is because calculating heart rate and identifying arrhythmias and other cardiac abnormalities requires the ECG waveform to be analysed over at least a couple of cardiac cycles.

7.3.4 Complex Event Processor (CEP) Component

As discussed, every time a thread in the interpreter component identifies a significant trend in a sensor's data readings, an asynchronous event is fired. These asynchronous events are then handled by the CEP component. Each time an interpreter thread in the sensor interpreter component fires an asynchronous event, an event handler in the CEP component records the new patient condition identified by the event firing interpreter thread. After this, the handler then checks whether the combination of the newly identified condition and any other conditions that have been recorded are indicative of an acute complication of a chronic disease currently being managed. If so, a complex event is inferred from the simple asynchronous events that were fired for the notification of each individual condition i.e. a disease complication is inferred from significant changes in a number of vital signs sensors. After a complex event is inferred, the patient is notified of the potential acute complication of disease by having the CEP component pass the relevant data to the user interface component, which packages the data and sends it to the patient's smartphone. In addition, dynamic reconfiguration takes place in the form of new sensor components being installed. The dynamically installed sensor components aid the patient in trying to confirm or reject the disease acute complication diagnosis given by the CEP component.

The handleAsyncEvent method of an event handler of the CEP component is shown in Figure 7.7.

```
public void handleAsyncEvent()
{
     setStatus(decresaeEEGAplha);
     setStatus(incresaeEEGTheta);
     setStatus(incresaeEEGDelta);
     //detection for hypoglcaemia
     if(getStatus(highHeartRate) == true)
     {
           UI.notifyUser("hypoglycaemia");
           //to confirm hypoglycaemia
           Bundle b = ctxt.installBundle("file:bundle/glucometer.jar");
           //to confirm hypoglycaemia
           Bundle b = ctxt.installBundle
               ("file:bundle/sphygmomanometer.jar");
     }
}
```

Figure 7.7 CEP Asynchronous Event Handler's handleAsycEvent Method Implementation

Finally, Figure 7.8 shows the case study application's components, services, threads, and asynchronous event handlers. Asynchronous event firing and service method calls are also shown. In Figure 7.8, notice how the monitoring application is designed such that if one sensor or sensor interpreter fails or otherwise requires adaptation, the other sensors and sensor service are able to continue to function. Such dynamic reconfiguration and its effects on the real-time constraints on the application are discussed further in Section 7.4.4.

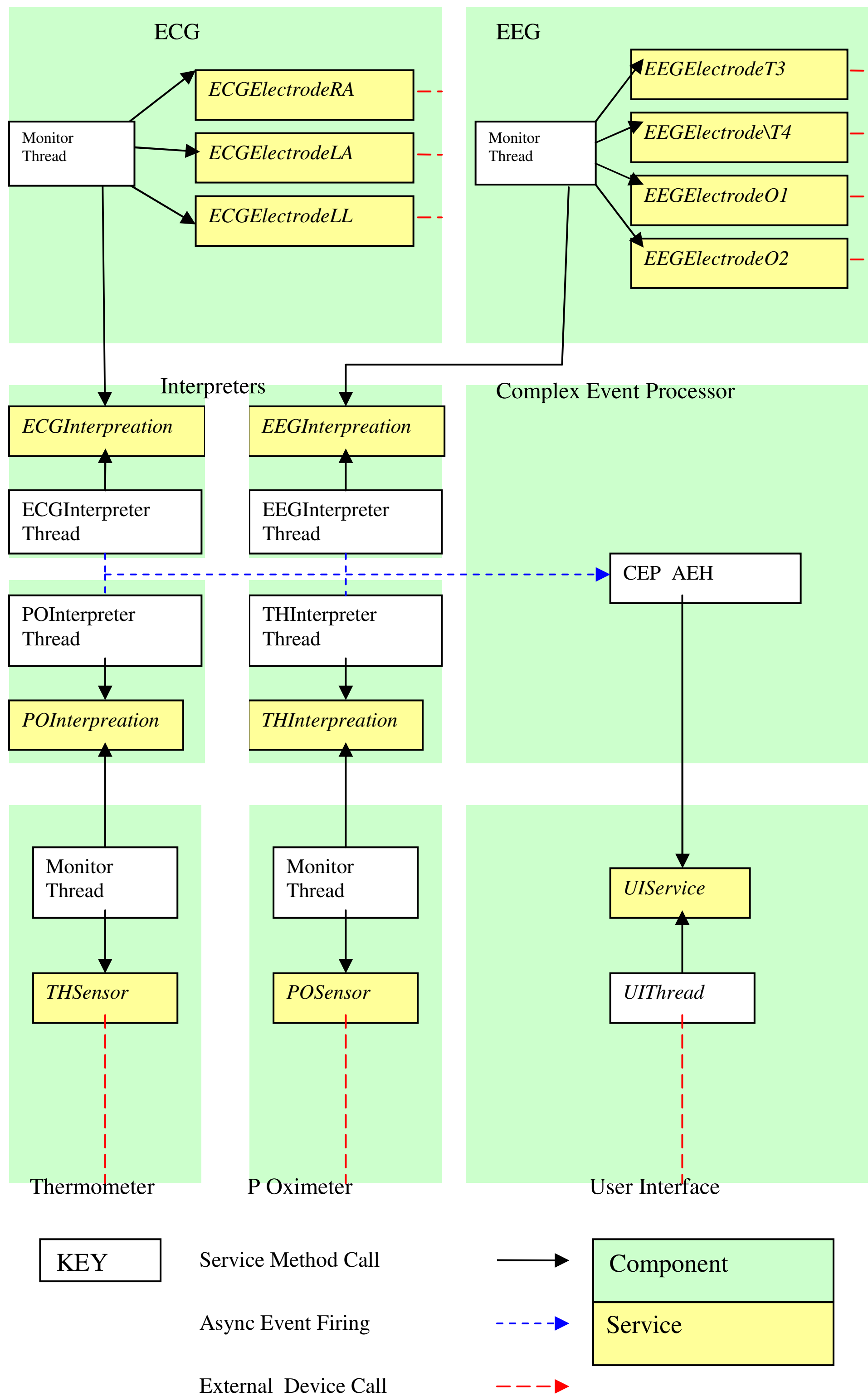

Figure 7.8 Chronic Disease Management Application Architecture

7.4 Application of the Chronic Disease Management Application – Monitoring for Hypoglycaemia in Diabetes

Having discussed the chronic disease management application, it is necessary to discuss its application to a specific chronic disease. In this section, the application of the chronic disease management systems to monitoring for the early detection of hypoglycaemia in insulin-dependent diabetes patients is discussed. Before discussing the components and timing requirements of the application, it is necessary to stress the fact that it is often not possible for insulin-dependent diabetic patients to perceive the symptoms of hypoglycaemia. Such patients are therefore unable to rely on the recognition of hypoglycaemia symptoms in order to monitor and correct hypoglycaemia. The reasons for this are discussed in detail in [149-151], suffice it to say that insulin-dependent diabetics have reduced physiologic defences against hypoglycaemia and are therefore more susceptible to suffering bouts of hypoglycaemia. The effect of frequent bouts of hypoglycaemia is a reduced perception of its symptoms. With this in mind, the motivation for the proposed RT-OSGi chronic disease monitoring application is clear.

In the chronic disease management application, it would be desirable to continuously monitor the blood glucose level of insulin-dependent diabetics such that, upon detecting hypoglycaemia, the patient can be notified and action taken before the hypoglycaemia worsens and becomes a medical emergency. Unfortunately, continuously monitoring blood glucose levels is invasive, requiring a permanently attached device which punctures the skin and perhaps wirelessly transmits the measured blood glucose level to a receiver used by the RT-OSGi application. Of course, such an approach is not pleasant for the patient and is generally impractical with issues such as the risk of infection. Infrared spectrometry has recently gained attention as a less accurate but no-invasive alternative to measuring blood glucose, however this approach is still under investigation. As a result, in the RT-OSGi chronic disease management application, it is proposed that the physiologic response to low blood sugar rather than the blood sugar itself be continuously monitored by the application.

Electroencephalography (EEG) measures the electrical activity of different parts of the brain. The EEG recordings mirror the functional state of the brain, and since glucose is the obligate substrate for cerebral (brain) metabolism, even moderate hypoglycaemia has a well established ability to cause gradual cerebral deterioration along with the associated encephalographic abnormalities in subjects with and without diabetes. More specifically, the EEG patterns correlate with the cerebral metabolic rate for glucose, as assessed by energy charge potentials. A decrease in the cerebral metabolic rate for glucose is associated with a slowing of the EEG pattern; an increase in slow waves precedes convulsive, polyspike activity, and an isoelectric EEG. [152]. As a note, these changes in EEG are not specific to hypoglycaemia and may also occur in other metabolic encephalopothies, e.g. hypoxia (lack of Oxygen). However, like hypoglycaemia, hypoxia is also potentially life-threatening and therefore it is beneficial that this may be detected by the chronic disease management application while monitoring for hypoglycaemia.

A key point discussed above is that during changes to the EEG, the patient with hypoglycaemia has gradual cognitive dysfunction. Changes in the EEG precede the worsening of cognitive performance during hypoglycaemia in hypoglycaemia- unaware patients by less than 20 minutes [152]. In one study [153], in an experiment in which 15 subjects were hypoglycaemia induced, 14 out of 15 patients experienced changes in EEG indicative of hypoglycaemia approximately 9 minutes before severe cognitive impairment occurred i.e. the point where it is doubtful whether the patient could have taken corrective action. In one patient the time was only 3 minutes.

Clearly, this emphasises just how high the availability requirements for the chronic disease management application are when used for monitoring for hypoglycaemia in insulin-dependent diabetes patients. For example, if the system is taken offline just as the EEG warning alarm is to be fired, the alarm will not be fired until the system is brought online again, i.e., after the application has been recompiled and reloaded. This offline reconfiguration process may take several minutes for large high level language application. However, if the system is offline for three minutes then the patient is not warned about the hypoglycaemia

until six minutes before severe cognitive dysfunction, this means that the patient has not only lost one third of their possible window for response to the alarm, but also, bearing in mind the increasing cognitive dysfunction that the patient experiences over the nine minutes, the patient will be experiencing a much higher level of severe cognitive dysfunction. Therefore, the importance of the dynamic reconfigurability of RT-OSGi in the context of monitoring for hypoglycaemia in insulin-dependent diabetes is clear.

As a note, the hypoglycaemia detection application is not safety critical. The reason for this is because the scientific methodology on which the warning detection is based on (i.e. measuring the physiologic response to hypoglycaemia rather than invasively measuring the blood glucose level directly) has not been clinically proven to be effective outside of the constrained settings in which it has been medically tested. Further medical research would be required before any kind of home-based health care system for the early detection of hypoglycaemia based on EEG measurements could be routinely used. Rather the chronic disease management application is designed to provide a best effort attempt at detecting hypoglycaemia.

7.4.1 Dynamic Reconfiguration Examples

In this section, examples of performing dynamic reconfiguration on the chronic disease management application in the context of hypoglycaemia monitoring and detection are discussed.

7.4.1.1 Installing New Components – Admission Control and GC Reconfiguration

In this section, an example of dynamic reconfiguration in the form of adding a new component to the chronic disease management application discussed in Sections 7.3 is given. Table 7.1 shows the temporal specification of the components currently installed in the chronic disease management application. These temporal specification are derived from [154].

Component	C (ms)	T (ms)	D (ms)
ECG	0.5	2	2
EEG	0.5	4	4
Pulse Oximeter	0.5	1000	1000
Thermometer	0.5	1000	1000
ECG Interpreter	100	1200	1200
Pulse Ox Interpreter	100	1200	1200
Thermometer Interpreter	100	1200	1200
EEG Interpreter	100	1200	1200
Complex Event Processor	20	2000	2000
User Interface	100	10000	10000

Table 7.1 Temporal Specifications of Installed Components' Servers

Assuming that one of the short term acute complications of disease being monitored for is hypoglycaemia in diabetes, and that the Interpreter services for ECG and EEG indicate a significant change in their respective sensor values, both services fire asynchronous events, which are handled by the complex event processor (CEP) component. When the CEP event handler is released a second time i.e. after one of the Interpreter services has already notified it of either a significant change in EEG or ECG, the CEP handler will notice that significant changes have occurred in EEG and ECG and can infer the complex event of hypoglycaemia. In addition to alerting the patient through their smartphone via the user interface component, the CEP component will also dynamically reconfigure the chronic disease management application by installing a component which enables the patient to check their blood glucose and blood pressure to help confirm the diagnosis.

As part of this dynamic reconfiguration, admission control must take place. The temporal specification of the blood glucose/blood pressure sensor component is shown in Table 7.2.

Thread ID	C (ms)	T (ms)	D (ms)
Blood Glucose	400	20000	20000
Blood Pressure	200	20000	20000

Table 7.2 Temporal Specification of Blood Glucose/Pressure Component

After reading the component's temporal specification, the admission control associated with component install (discussed in Chapter 5) must be performed. The first stage of this is to generate server parameters for the blood glucose/blood pressure sensor component based on its threads' temporal specifications. Using the server parameter selection algorithm (propose in [123]) with this component, the server budget is calculated as 700ms, and the server period is calculated as 10350ms .

After determining the blood glucose/blood pressure component's CPU resource requirements through server parameter selection, it is necessary to check whether the component can be assigned the required amount of CPU time without affecting the schedulability of the currently deployed components. The results of schedulability analysis are shown in Table 7.3. In the table, C = computation time, T = period, D = deadline, A = memory allocation per period (in Bytes), R = response time, BR = the response time calculated from the Boolean schedulability analysis, and RUB = response time upper bound. Note that the emphasis in this table is on CPU-admission control. Since the priority assignment algorithm does not have a direct impact on this, the priority assignment is not shown in Table 7.3.

C (ms)	T (ms)	D (ms)	A (B)	R (ms)	BR (ms)	RUB (ms)
0.5	2	2	1000	0.5	0.5	0.5
0.5	4	4	1000	1	1.5	1.16
0.5	1000	1000	1000	1.5	236	2.1
0.5	1000	1000	5000	2	375	2.9
100	1200	1200	6000	162	462.5	163.2
100	1200	1200	6000	322	478	357.9
100	1200	1200	6000	482	572	623.5
100	1200	1200	6000	642	645	1007.5
20	2000	2000	20000	674	1330.5	1336.5
100	10000	10000	500000	834	4938.5	1739.7
700	10350	10350	1000	3271	6660.5	4386.5

Table 7.3 Response Times for Servers

From Table 7.3, it can be seen that according to the sufficient schedulability test (RUB), the system is unschedulable as two servers have RUBs greater than their deadline. However, as indicated by the exact tests (Boolean response time (BR) and response time (R)), the system is in fact schedulable after dynamic reconfiguration, and therefore the blood glucose/blood pressure sensor component can be assigned a priority range as they have passed the CPU acceptance test of admission control. However, before admitting the blood pressure/blood glucose component into the system, GC reconfiguration analysis must take place and the application must pass the free memory related acceptance tests of admission control. The results of GC reconfiguration analysis are shown in Table 7.4 along with the GC configuration that was used before the blood glucose/blood pressure component underwent admission control. In the table, W_{GC} = the amount of GC work that is required, C_{GC} = the computation time of the GC controller thread, T_{GC} = the period of the GC controller thread, R_{GC} = the response time of the garbage collector, i.e. the time it takes the GC to complete a GC cycle based on the C_{GC} and T_{GC} that it was assigned, and M = the amount of memory allocated by application threads during a GC cycle.

GC Configuration Values	Previous Configuration	New Configuration
W_{GC} (ms)	27.38	27..51
C_{GC} (ms)	0.6	0.45
T_{GC} (ms)	2	2
R_{GC} (ms)	91.78	111.51
M (MB)	3.418	3.45

Table 7.4 Application Dynamic Reconfiguration Effect on GC Configuration

Note that although the GC controller thread is assigned the same period (T_{GC}) as the application thread with the smallest period, the GC controller thread is assigned a priority one higher than the application thread and hence it will be able to pre-empt it. Note also that schedulability analysis does not need to be performed again with the GC parameters because the GC parameter selection algorithm calculated the parameters such that application schedulability is maintained.

As discussed in Chapter 6, C_{GC}, T_{GC}, and M are used as acceptance tests for admission control. Only if the GC parameters are sufficiently large to be of use (i.e. C_{GC} and T_{GC} are significantly larger than the context switch overhead), and only if there is sufficient free memory in the JVM to ensure that the application threads can allocate an amount of memory equal to M, is the dynamic reconfiguration of the application (in this example, the addition of the blood pressure/glucose component) permitted. It is assumed in this example that thee conditions are satisfied thus the blood pressure and blood glucose components can be admitted into the system.

As a note, in this example of dynamic reconfiguration, that is, when adding the blood glucose/pressure component to the application in order to attempt to filter out false positive results from the EEG sensor for hypoglycaemia, the timing constraints of the chronic disease management application are completely unaffected. Thus the application maintains high availability levels. Without the

dynamic reconfigurability of RT-OSGi, the entire application would need to be taken offline thus compromising the timing requirements of the chronic disease management case study application.

As a further note, in this example, admission control was performed for the blood glucose/blood pressure component when the component was required to be installed. The reason for this was to simply illustrate admission control in the context of the case study application. However, in reality, the admission control would have been performed upfront at the time when the first component of the chronic disease management application was installed. The reason for this is because the blood glucose/blood pressure component is a core part of the application and therefore it must be guaranteed resources upfront of being deployed. This upfront reservation for a group of component that constitute an application was discussed in Chapter 5.

7.4.1.2 Replacing Existing Components – Effect on Application Timing Constraints and the RT-OSGi Mode Change Protocol

The second example of dynamic reconfiguration in the chronic disease management application is the replacement of a component, i.e. the installation of a new version of a component followed by the removal of the old version. In the example, the component being replaced is the EEG interpreter component. The EEG interpreter thread in the new version of the component is capable of interpreting the EEG data not only for detecting signs of hypoglycaemia in insulin-dependent diabetics (as was the case with the old version of the thread), but also for attempting to detect the early signs of an epileptic seizure. As a result of this, the computation-time requirement of the new version of the EEG interpreter thread is larger than that of the old version.

Since the EEG interpreter service of the EEG interpreter component is invoked from both the interpreter thread within the component and from the monitoring thread within the EEG sensor component, not having RT-OSGi manage and

coordinate the transition from the execution of the old version of the component to the new version may cause timing faults for both the thread of the component being replaced (specifically the EEG interpreter thread), but also for any threads in any components using the EEG interpreter component (namely the EEG sensor component's monitoring thread which calls the EEG interpreter component's EEG interpreter service).

To avoid component replacement from causing timing faults in the component being replaced and any other components which use services registered by the component being replaced, RT-OSGi provides the following transition scheme (mode change protocol [155]) for component replacement. Firstly, the new version of the component to be replaced is installed. The install operation (and any other life cycle operation invoked by the user) is processed by an execution-time server that executes at a user configured priority, in this example (and in the thesis in general), the priority is assumed to be lower than that of all other application components' threads.

As discussed in Chapter 5, admission control for components must typically occur at deployment time (rather than in advance of that time) and there is a risk that the new version of the component may fail admission control when it is needed for deployment. In such a situation, there is no other option but to either modify the temporal specification of the new version and/or remove components that are currently deployed in order to free up some resources. As a note, in the case where the new component version is known before deployment of the old version and in the case where it isn't, the temporal specification of the new version does not have to be the same as the old version. Of course, the smaller the computational and memory demands of the new version, the more likely it is that it will pass admission control. This is an important point because at least for a short period of time, the old and new versions of the component will be deployed in parallel thus it is desirable to consume as little resources as possible between them. Note that in terms of analysis, the schedulability analysis discussed in Chapter 5 is carried out with both the old and new version of the component. While more pessimistic than mode change schedulability analysis, the pessimism is minimal because the analysis is used in the context of the

deployment of only two versions of a single component, rather than the more extensive deployment changes which occur with application mode changes.

After being installed, the user invokes the start operation on the new version of the component. The life cycle processing thread of RT-OSGi will then execute this operation. As part of the new component's start operation it will undergo activation (initialisation). During this time, it will register the new version of the service on which other components depend. After service registration, it will fire a service registration event and synchronously call any handlers for the service registration event. The handlers (one for each service-using thread in the application) will then set a flag in an application thread which is dependent on the service being replaced as part of component replacement. As a note, this notification process is non-real-time as it occurs from the non-real-time life cycle processing thread. Furthermore, the fact that the notification process is non-real-time is not an issue since removal of the old version of the component does not occur until after all threads have received notification of the component replacement process and have notified to RT-OSGi that it is safe to remove the old version. Finally, as part of the initialisation of the new component, any necessary threads are created and started. In order to ensure that these threads can execute before the deadline of the next release of the old version of the thread in the component being replaced, the deadline of the new threads must be less than or equal to their counterparts in the old version of the component. The significance of this is demonstrated in Chapter 8.

All service requesting threads poll the flag that was set by their synchronous service registration event hander at the end of their period and are thus able to complete their execution before being notified of the requirement to transition to using the new versions of the services they use. If the flag is set, the service requesting threads calls the unget method of the service factory registered by the old version of the service. In OSGi, a service factory enables the service provider to customize an instance of a service to each requesting component thus supplying each component with a copy of the service rather than the default approach of OSGi which is to have all components share an instance of a single service object. In addition, service factories can also be used as a general means

of customising the process of acquiring and releasing references to a single instance of a service object too. As a result, RT-OSGi utilises service factories as a means of enabling application developers to transfer state from the old version to the new version of a service during component replacement. As a note, because service state is application dependent, it is not possible to have an RT-OSGi mechanism automatically map the state from one service to another, rather, the service factory approach simply enables the service provider to provide their own mapping. The use of such service factories has been implemented for experimental purposes as proof-of-concept.

After executing the service factory unget method of the old service provider in order to save the state of the old version of a service, service requesters then obtain a reference to the new version of the service, loading the state of the previous service's state by calling the get method of the service factory provided by the new service provider. Clearly, the old and new service factories must cooperate in order to enable seamless saving and loading of service state. Moreover, the service factories must be designed such that they consider the fact that only a single service requester is required to save and load service state but yet there will typically be multiple service requesting threads.

Once a thread has obtained a reference to the new service and the service state is transferred from the old service, service requesters notify RT-OSGi that they have now made the transition to using the new service of the new component. Once all service requesters of the old service have made this notification, the old component containing the old version of the service can then be removed from the application. This is achieved by having RT-OSGi compare the number of notifications of transition by service requesters to the total number of service requesters of the old service. Once these two numbers are equal, the old version of the component can be removed without violating the timing constraints of the service requesting threads. As a note, service requesting threads do not have to transition to the new service as soon as they have polled the flag at the end of their period, instead they could execute for any number of periods before transitioning. What is important is that when a thread transitions, it notifies RT-OSGi, as discussed above.

It is imperative that all threads notify RT-OSGi before the old version of the component is removed. The reason for this is because threads poll for the need for service transition at the end of every period and, since the periods of threads typically differ, threads will not transition to the new service simultaneously. As a result, there is a possibility that a service requesting thread in the old version of a component which depends on another thread requesting the same service in order to function correctly, for example a reader and writer-type situation with the service, may terminate as part of component removal before the other thread obtains a reference to the new version of the service. This means that, in the case of the reader writer example, the reader thread in the new component is attempting to read from the new service but the writer thread has not yet obtained a reference to the new service and is still writing to the old service, the old reader may then terminate meaning that the values written to the old service are not being read anymore and so there is a break in the components availability and timing constraints of the application are violated since the new reader thread cannot function correctly until the writer thread starts to write values to the new service. This example is more thoroughly discussed shortly in the context of reading and writing data samples in the chronic disease management application.

The steps taken by threads dependent on services registered by a component being replaced are summarised in Figure 7.10. Steps 1 – 4 are executed by the life-cycle processing thread of RT-OSGi, which as discussed in Chapter 5, execute under a server and is included in application schedulability analysis. Steps 5 – 10 are executed by each service requesting thread in the application, the WCET of these steps is included in the calling thread's WCET analysis. The WCET service factory methods which are called in steps 5 and 6 are included into the calling thread's WCET analysis by using execution-time contracts much in the same way as service method invocations. This was discussed in Chapter 4. As a result of these features, the whole transition process of service requesters during component replacement does not disrupt the temporal constraints of rest of the application.

1) New version of component is installed and started
2) Replacement service event fired
3) Event handlers synchronously called
4) Handlers set notification flag in service requesting threads within their component
5) At end of period, service requesters check flag
6) If flag equals true, save service state
7) Obtain new service, loading service state saved by old service if no other thread has already done so
8) Notify RT-OSGi that transition to new service complete
9) Block for next period
10) Execute new service on subsequent releases

Figure 7.9 Steps Taken by Service Requesters during Component Replacement

The application of the steps summarised in Figure 7.10 is now applied to an example in the chronic disease management application. The component to be replaced is the EEG Interpreter component, the component replacement scenario is shown in Figure 7.11. Note that in terms of application state when temporarily running the old and new versions of a component in parallel, it is the component developer's responsibility to ensure correct application state.

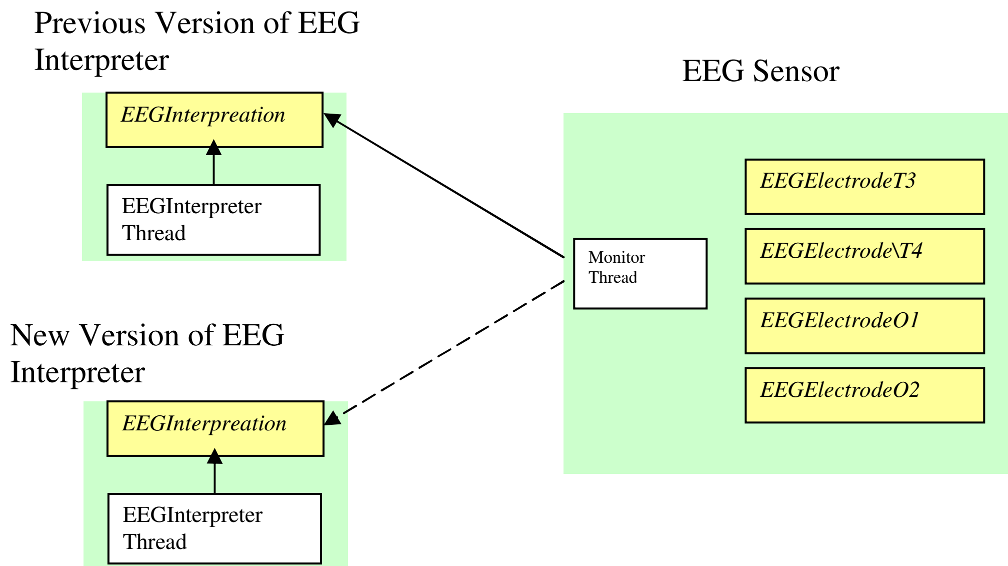

Figure 7.10 Required Transition for the EEG Sensor Component

Following the component replacement transition scheme previously discussed, firstly, the new version of the EEG interpreter component is installed and started and the new version of the EEG interpreter service is registered. The new version of the EEG interpreter thread is also started, but since the EEG sensor component's monitoring thread is writing data samples to the old version of the service, there are no values written to the new service and thus the thread has no values to read.

After the registration of the new version of the EEG interpreter service, RT-OSGi fires a synchronous event for the registration of the new version of the EEG interpreter service and synchronously calls any event handlers registered by the application. Since the EEG sensor component contains the only service requester outside of the EEG Interpreter component that requires the EEG interpreter component's EEG interpreter service, the EEG sensor thread's synchronous event handler for handling the service events related to the EEG interpreter service is the only handler to be synchronously called by the life cycle processing thread.

In the case of the EEG monitor component's synchronous event handler, when called, the handler should set a flag in its monitoring thread to notify it that the

EEG Interpreter service which it is currently using will shortly be replaced and thus it should obtain a reference to the new version of the service. At the end of its period, the monitoring thread polls the flag and determines that the EEG interpreter service is being replaced. It firstly calls the old version of the EEG interpreter service's service factory in order to save any service state. As discussed the saving of service state is application dependent. In this example, the only service state is a data buffer which stores the data samples passed to the EEG interpreter service from the EEG sensor monitor thread. Since the EEG sensor thread executes at a higher frequency than the EEG interpreter thread, that is, it has a smaller period (as can be seen in Table 7.1), the EEG sensor thread writes data samples to the EEG Interpreter service's data buffer at a faster rate than which they are read by the EEG interpreter component's EEG interpreter thread reads them. Therefore, these values must be copied to the new service so that the new interpreter thread can read these values before reading any new ones written by the EEG sensor. Therefore, the service factory saves this buffer in non-volatile memory. The size of the area reserved will not be a concern since the buffer size is fixed and known in advance because the number of data samples in the buffer can be calculated based on the rate of reading and writing i.e. on the periods of the EEG monitor and EEG interpreter threads.

After calling the service factory of the old version of the EEG interpreter service in order to save the service's state, the EEG sensor thread then calls the service factory of the new EEG interpreter service to load the data buffer previously saved by the old version of the interpreter service and to obtain a reference to the new version of the EEG interpreter service. At this point, the new EEG interpreter thread which has already been started can then start to read the data samples copied from the buffer of the old version of the EEG interpreter service to the new one, before starting to read the data samples appended to the buffer by the EEG monitoring thread without breaking any temporal constraints.

The EEG monitoring thread will then signal to RT-OSGi that it has transitioned to the new version of the EEG interpreter service and that the old version of the EEG interpreter component can be removed, which in turn will terminate the old version of the EEG interpreter thread. If the EEG interpreter component was

instead taken offline without waiting for notification from the EEG monitoring thread, then there would be a break in the availability of the EEG interpreter functionality since the new EEG interpreter thread will have no data samples to read because the EEG monitoring thread will still be writing data samples to the old version of the EEG interpreter service for which the old version of the EEG interpreter thread which read these data samples has already been terminated. Therefore the timing constraints of the EEG interpreter will be broken for a time period equal to the difference between time at which the old EEG interpreter thread was terminated, and the tie at which the EEG monitoring thread started writing data values to the new EEG interpreter service.

Finally, the EEG sensor thread will then block for the beginning of its next period. In its next period it will be writing data samples to the new version of the EEG interpreter service and the transition will have caused no interruption in operation and thus no timing constraint violations. The component replacement is further discussed in Chapter 8.

Concluding this section on the example of component replacement in the context of the chronic disease management application, it is important to note that the service state transfer would be more complicated if, in the example, the interpreter thread which reads the values from the EEG interpreter service was in a different component from the EEG interpreter service. This would mean that the EEG interpreter thread would need to transition to use the new version of the EEG interpreter service much like the EEG monitoring thread has to do in the example. As an example of the added complexity of saving service state in this scenario, assume that the EEG sensor component's monitoring thread is at the beginning of its period, and will therefore not be alerted to the fact that it should start using the new version of the EEG interpreter service until the end of its period. As a result, it will write one final value to the old version of the EEG interpreter service's data buffer. However, the EEG interpreter thread may be at the end of its period when the new version of the EEG interpreter service is registered and therefore will receive the notification of this before the EEG monitoring thread. As a result, it will save the data buffer and load it to the new version of the EEG interpreter service before the EEG monitoring thread writes

its data sample to the old version of the EEG interpreter service. In this scenario, it is therefore imperative that the EEG monitor thread copies this last value to the new version of the EEG interpreter service when it makes its transition after the EEG interpreter thread. This prevents the value from never being read. Of course, when implementing this, it is essential that the service factory methods responsible for saving and loading service state make a distinction between the EEG interpreter and EEG monitoring threads when they invoke these methods. This is to ensure that, for example, the EEG monitoring thread doesn't transfer the data buffer if it is notified of the new EEG interpreter service registration before the EEG interpreter thread, as this would leave the interpreter thread with no data samples to read until its transitions to using the new version of the EEG interpreter service. This discussion illustrates the complexity of service state transfer in some applications. As a result, it is clear why application developers need to take responsibility for ensuring the correctness of state transfer and why RT-OSGi is capable of only providing application developers with a mechanism for performing the state transfer process.

In Chapter 8, an example run of the application is given to demonstrate the lack of down-time during dynamic reconfiguration, further analysis is also given.

7.5 Summary

Since the focus of this thesis is on providing the ability to maintain/evolve/reconfigure real-time applications which have high availability requirements, a case study was selected so as to show how RT-OSG can be used to deploy such applications while maintaining high levels of availability.

The case study chosen was a chronic disease management application with the purpose of the application being to monitor for complications of disease. Furthermore, the application of this management system to the monitoring of hypoglycaemia in insulin-dependent diabetics was discussed. The chronic disease management application clearly has real-time monitoring requirements, and requires maintenance/evolution/reconfiguration for a number of reasons such as replacing faulty sensors, updating the complex event processing component

and adding new sensors so as to monitor for new health conditions so as to be reconfigured in accordance with a patient's health conditions. Moreover, the chronic disease management application also has high availability requirements due to the fact that the ability of the application to detect complications of disease is compromised if the application has to be taken offline for maintenance and reconfiguration purposes. This is particularly true when the chronic disease management application is used to monitoring for hypoglycaemia in insulin-dependent diabetes. The utility of the application is substantially reduced when the application is taken offline for even a short period of time e.g. a. couple of minutes. The reason for this is because the time between the possible detection of hypoglycaemia through monitoring and the time in which the patient is likely to not have severe enough cognitive dysfunction from the hypoglycaemia and is thus able to respond to the hypoglycaemia alarm is only typically nine minutes! Thus having the application go offline for only a couple of minutes out of nine severely reduce the time window in which the patient has left to react to the alarm.

Finally, since episodes of hypoglycaemia further diminish the defences against subsequent episodes of hypoglycaemia including the development of hypoglycaemia unawareness. The utility of the case study presented in this chapter is therefore further enhanced when applied to hypoglycaemia detection.

8

RT-OSGi Evaluation

In order to verify the correctness of the design of RT-OSGi, a prototype is implemented. Based on this implementation, the effectiveness of RT-OSGi in meeting the goals of this thesis, i.e., to develop dynamically reconfigurable real-time systems in order to improve their availability and utility during evolution/maintenance situations where other non-dynamically reconfigurable applications would need to be taken offline, is demonstrated. Furthermore, from the prototype RT-OSGi, any overheads and the affect of any pessimistic assumptions made in the RT-OSGi model on application schedulability are measured. The prototype of RT-OSGi is discussed in Section 8.1 and an evaluation of both the ability of RT-OSG to meet the goals of this thesis and the overheads associated with RT-OSGi are discussed in Section 8.2

8.1 RT-OSGi Prototype Implementation

There are a number of implementations of the OSGi Framework available which provide the source code. Such OSGi implementations can therefore be modified in order to make them more suitable for developing dynamically reconfigurable real-time systems, essentially an RT-OSGi. In terms of selecting an OSGi Framework implementation on which to build RT-OSGi, the main selection criteria are well structured and easily comprehensible implementation code. The reason for this is because, perhaps, the most challenging aspect of building RT-OSGi is the task of program comprehension of the standard OSGi Framework on

which it is based. The Apache Felix OSGi Framework implementation was chosen to build RT-OSGi on for this reason.

The implementation of the RTSJ on which RT-OSGi (the modified real-time version of Apache Felix) will execute on is Sun Java RTS. The reason for this choice of the RTSJ implementation was explained in Chapter 6. Sun Java RTS is the only implementation of the RTSJ which provides a GC which can be dynamically reconfigured. As discussed in Chapter 6, dynamically reconfigurable GC is essential for RT-OSGi applications. The Sun Java RTS JVM and class libraries are not modified in any way in order to support RT-OSGi and therefore any version of the Sun Java RTS JVM which supports dynamically reconfigurable GC is adequate. Furthermore, in the future, RT-OSGi may be deployed on any other RTSJ implementations which begin to provide support for dynamically reconfigurable GC. This is a major advantage of the approach taken in this thesis to implementing RT-OSGi using application level features and not modifying the underlying JVM/OS.

Finally regarding the OS, the Red Hat distribution of Linux, kernel version 2.6.21-57.el5rt patched with the SMP PREEMPT and RT patches is used to host Sun Java RTS and RT-OSGi. This operating system was chosen for convenience because it is freely available. However, as with Sun Java RTS, RT-OSGi does not require any modifications to the underlying OS. As a result, any OS may be used provided that a distribution of Sun Java RTS (or any future JVM which supports dynamically reconfigurable GC) is available for it. Figure 8.1 shows the run-time environment used to deploy RT-OSGi

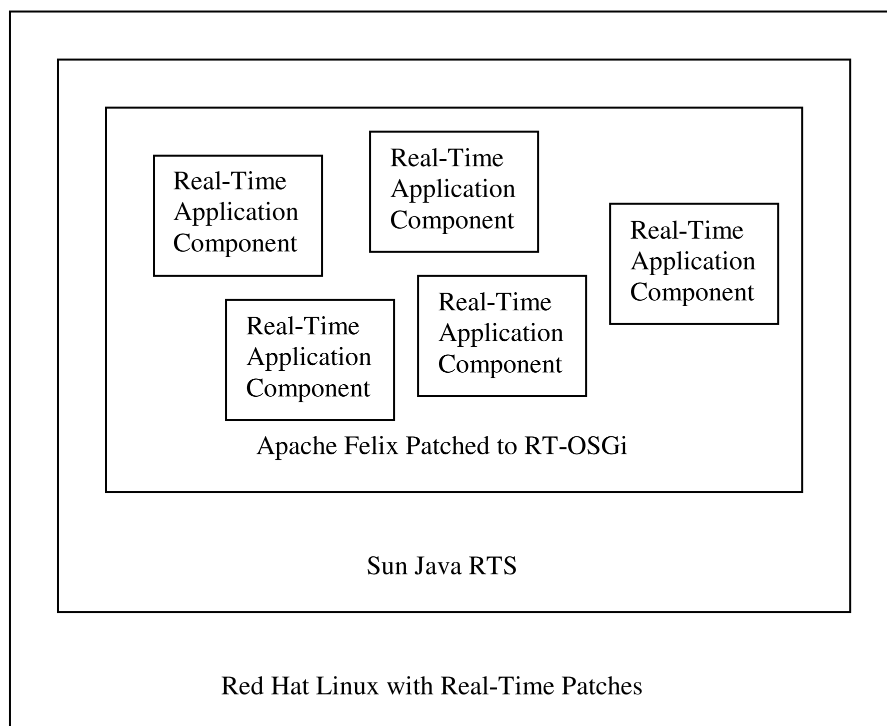

Figure 8.1 Run-Time Environment Used for the RT-OSGi Prototype

8.1.1 Deploying RT-OSGi Applications – Apache Felix Modifications and Extensions

As discussed throughout this thesis, the major modifications required to the standard OSGi Framework in order to make it capable of deploying dynamically reconfigurable real-time systems with high availability requirements are: temporal isolation, CPU admission control (including server parameter selection, schedulability analysis, and priority range assignment), GC reconfiguration analysis and GC thread reconfiguration, memory admission control, memory allocation enforcement, and asynchronous thread termination. All of these features are linked with application dynamic reconfiguration, e.g. admission control and GC reconfiguration are linked with the installation of components, and asynchronous thread termination is linked with the removal of components. Since dynamic reconfiguration in OSGi/RT-OSGi occurs through the life cycle operations, it is these operations that require the greatest modifications in order to transform the Apache Felix standard OSGi Framework implementation into an implementation capable of meeting the goals of this thesis (RT-OSGi). As a result, the most significant modifications to Apache Felix are made to the Java

class which implements the life cycle operations, namely, the class "Felix" in the package `org.apache.felix.framework`.

The life cycle methods in the class Felix (`org.apache.felix.framework.Felix`)) are modified to provide the aforementioned features of RT-OSGi. The following Java packages are introduced into Apache Felix and used by the life cycle operations in order to implement the required RT-OSGi functionality:

1. `org.apache.felix.framework.priorityassignment` – to perform the priority range assignment discussed in Chapter 5.
2. `org.apache.felix.framework.schedulabilityanalysis` – to perform the Boolean RTA and RUB discussed in Chapter 5.
3. `org.apache.felix.framework.serverparameters` – to perform the server parameter selection discussed in Chapter 5.
4. `org.apache.felix.framework.gcreconfig` – to perform the GC Reconfiguration analysis discussed in Chapter 6.

Finally, other than the changes to the class Felix, there are little other changes made to Apache Felix. The exception to this is the introduction of a class into the `org.apache.felix.framework` package that implements an interface "RTBundle". The RTBundle interface introduces the concept of a real-time component to Apache Felix and is discussed further in the following section. Aside from these modifications/extensions which transform Apache Felix into RT-OSGi, RT-OSGi generally retains the same structure as Apache Felix. This is an important point for RT-OSGi maintenance reasons.

8.1.2 Developing RT-OSGi Applications – RTSJ Class Extensions and OSGi Manifest Extensions

While the modifications and extensions to Apace Felix are aimed at providing a suitable environment for deploying dynamically reconfigurable real-time systems with high availability requirements, the RTSJ extensions classes discussed here

enable application developers to target their applications at the RT-OSGi Framework. The Java package `uk.ac.york.rtosgi` contains the following five classes to be used by RT-OSGi application developers: OSGiRTT, OSGiAEH, OSGiSchedulable, OSGiPGP, and RTBundle. OSGiRTT and OSGiAEH extend the RTSJ classes RealtimeThread and AsyncEventHandler respectively. These two classes also extend the OSGiSchedulable interface which provides a number of methods which need to be implemented by any schedulable entities in RT-OSGi, much like the purpose of the Schedulable interface in RTSJ.

The OSGiRTT and OSGiAEH should be used in RT-OSGi components rather than their standard RTSJ counterparts. The reason for this is because the OSGiRTT and OSGiAEH incorporate code to support the cost enforcement (discussed in Chapter 4), asynchronous thread termination (discussed in Chapter 5), memory allocation enforcement (discussed in Chapter 6), and safe component replacement/mode change protocol (discussed in Chapter 7) required by the RT-OSGi environment. As discussed in Chapter 4, the class loaders of RT-OSGi can enforce the use of such classes by throwing exceptions when references to the standard RTSJ schedulable objects are encountered during class loading.

The OSGiPGP class extends the ProcessingGroupParameters class of the RTSJ. It is used by RT-OSGi to provide execution-time server behaviour, with each application component being assigned an OSGiPGP. Instances of the OSGiRTT and OSGiAEH classes obtain a reference to an OSGiPGP through a reference to their component. Since the Bundle interface of standard OSGi (which represents an OSGi component) clearly does not contain methods to set and get OSGiPGP references (as servers are not a part of standard OSGi), the RTBundle interface is used to provide such methods in addition to other methods which are required as part of the notion of a real-time component in RT-OSGi for example methods related to priority assignment. Figure 8.2 shows the RTBundle interface methods. To summarise the RTBundle interface, its purpose is to act as an intermediary/bridge between application schedulable objects and the RT-OSGi Framework itself, much like the Bundle interface of standard OSGi allows standard Java threads to interface with the standard OSGi Framework.

```
public interface RTBundle extends Bundle
{
	public void lowerPriority();
	public void raisePriority();
	public void setPGP(ProcessingGroupParameters pgp);
	public ProcessingGroupParameters getPGP();
	public void addSchedulable(OSGiSchedulable so);
	public void setRequiredNumberOfPriorities(int req);
	public int getRequiredNumberOfPriorities();
	public int getElement(int index);
	public void setElement(int index, Integer priority);
	public void setPriorities(int index,
		int serverRequiredPri,ArrayList freePriorities);
	public boolean bundleHasPGP();
	public ArrayList getSchedulables()
}
```

Figure 8.2 The RTBundle Interface of RT-OSGi

In addition to the requirement for RT-OSGi application developers to substitute the use of the standard RTSJ schedulable classes in components with the aforementioned RT-OSGi counterparts in the `uk.ac.york.rtosgi` Java package, developers must also make use of some additional headers in their components' OSGi manifest files defined specifically for RT-OSGi. These real-time manifest headers are shown in Figure 8.3.

```
Real-Time: true | false
Schedulable-Specification: PATH
Required-Priorities: NUMBER
```

Figure 8.3 Additional Manifest Headers Required for Real-Time Components

The "Real-Time" manifest header in Figure 8.3 is used by the RT-OSGi Framework to determine whether or not a component undergoing installation has real-time requirements or not. If the header is present in the manifest file and is set to false or if the header is absent (as would be the case with legacy standard OSGi components), the component is treated as non-real-time and is accessible through an object implementing the Bundle interface of standard OSGi. If the header is present and is set to true, the component has an object created for it implementing the RTBundle interface previously discussed and is subject to

admission control and all of the other features associated with RT-OSGi applications discussed throughout this thesis.

The “Schedulable-Specification” manifest header must be defined by application developers if the Real-Time header is present and set to true, i.e. it must be present in all real-time components in RT-OSGi. The header should give the path to the file within the component which contains the temporal specification (i.e. computation-time (ms), period (ms), deadline (ms) and relative priority (where priority ranges from 0 to the number of unique priority levels required) and memory allocation per period (Bytes) requirements of all of the schedulable objects that will be created and started by the component. An example of an XML file containing such information about a component’s threads is shown in Figure 8.4. The temporal and memory allocation data in the file is used by real-time component admission control in order to perform schedulability analysis and GC reconfiguration analysis etc.

```
<?xml version ="1.0"?>
<specification>
    <task>
        <cost> 400</cost>
        <period> 1300</period>
        <deadline>1300 </deadline>
        <priority>3 </priority>
        <allocation> 1000000</allocation>
    </task>
    <task>
        <cost>1000 </cost>
        <period> 6800</period>
        <deadline>6800 </deadline>
        <priority>2 </priority>
        <allocation> 500000</allocation>
    </task>
    <task>
        <cost> 800</cost>
        <period>4600 </period>
        <deadline>4600 </deadline>
        <priority> 1</priority>
        <allocation>500000 </allocation>
    </task>
</specification>
```

Figure 8.4 Example of Defining the Temporal and Memory Allocation Requirements of a Component’s Schedulable Objects

The final manifest header in Figure 8.3, "Required-Priorities", must also be present when the Real-Time header is present with a value of true. This header is used by RT-OSGi for checking whether there are sufficient free priorities available in the system to support the component as part of the real-time component admission control process.

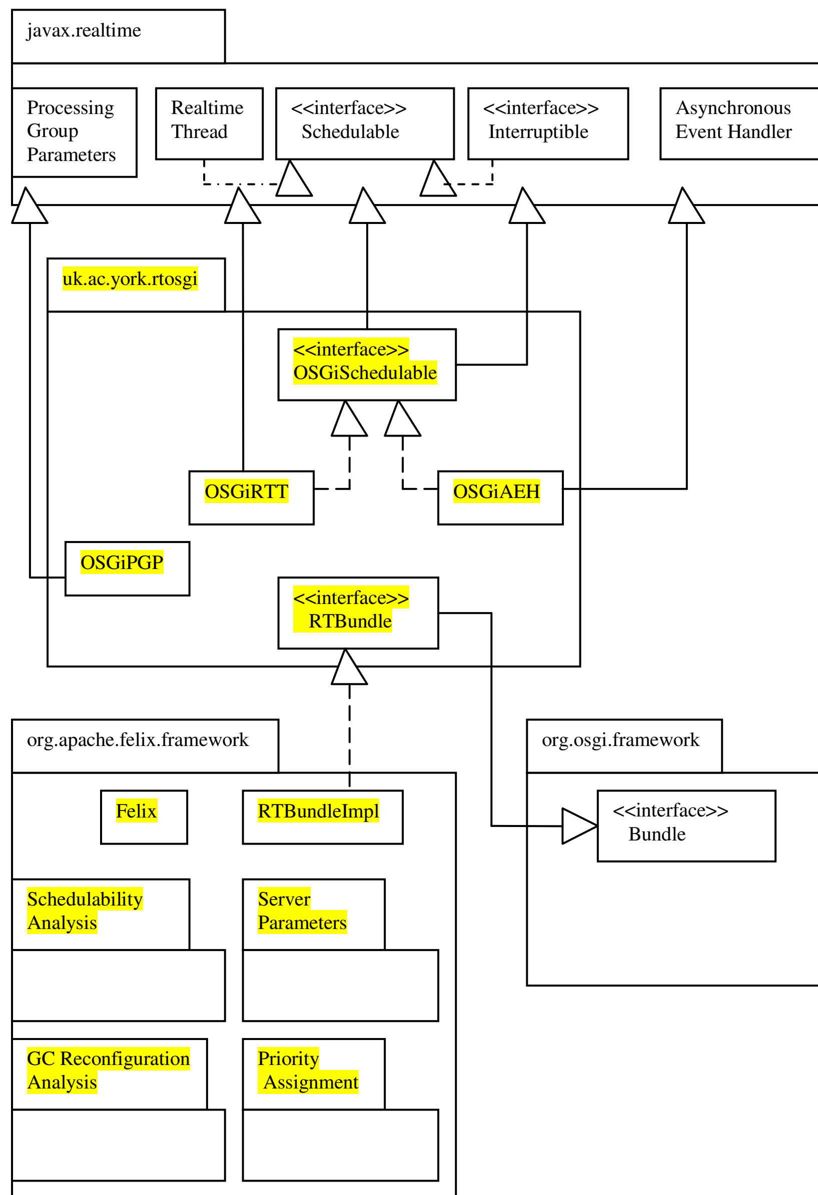

Figure 8.5 Relationship Between the RT-OSGi, RTSJ and OSGi Packages

To conclude this section on the RT-OSGi prototype, Figure 8.5 shows the relationship between the Java packages introduced into Apache Felix (in order to transform Felix into RT-OSGi.), and the Java packages which comprise the Apache Felix OSGi Framework implementation and the RTSJ. The highlighted

packages and classes are the extensions introduced as part of RT-OSGi and are the implementation of the contributions of this thesis. Notice that the Java interfaces comprising the OSGi Framework specification and the Java interfaces and classes comprising the RTSJ are not modified. Rather, the additional classes of RT-OSGi are packaged as part of the Apache Felix implementation of the OSGi interfaces in the `org.apache.felix.framework` package.

8.2 RT-OSGi Evaluation

Due to the broad scope of RT-OSGi, it is not possible to evaluate all of its features. Rather, the areas perceived to have most significance are evaluated and the other features are discussed as future work in the form of advanced evaluation in Chapter 9. Furthermore, some areas of RT-OSGi cannot really be evaluated quantitively and are instead evaluated through proof by construction through the RT-OSG prototype discussed in Section 8.1.

The main areas of RT-OSGi that were evaluated through simply demonstrating that they function correctly in the prototype RT-OSGi implementation are: asynchronous thread termination, cost monitoring and cost enforcement, memory allocation monitoring and memory allocation enforcement, time-based GC and GC thread dynamic reconfiguration. Clearly, it is difficult to evaluate such features in any other way. The five main areas of RT-OSGi that can be more thoroughly evaluated and discussed are the following:

1. The need for deploying RT-OSGi on an RTSJ JVM rather than a standard JVM
2. The ability to perform dynamic reconfiguration on real-time applications without affecting its temporal constraints thus maintaining high levels of application availability.
3. The ability to support dynamically reconfigurable real-time applications with high levels of availability without being subject to unreasonably high execution-time overheads.

4. The ability to support dynamically reconfigurable real-time applications with high levels of availability without leading to poor application schedulability results.
5. Although it cannot be thoroughly evaluated, the backwards compatibility of RT-OSGi with standard OSGi components and the learning curve required for standard OSGi Framework and RTSJ application developers to begin developing RT-OSGi applications.

These five areas of evaluation/discussion are mostly related to the approach of RT-OSGi in meeting the goals of this thesis rather than the prototype implementation, with the exception of evaluating the overheads of RT-OSGi which is carried out through execution-time measurements of the RT-OSGi prototype life cycle operations with an empty implementation of the chronic disease management application (i.e. the components are implemented but contain no application logic, only the necessary RT-OSGi meta-data required for deployment). The other areas of evaluation focus on the approach of RT-OSGi. One of these areas of evaluation, namely the evaluation of the effects of application dynamic reconfiguration on application availability and its ability to continue to meet real-time requirements is evaluated by executing a simple example RT-OSGi application and observing its behaviour under different dynamic reconfiguration scenarios. The effect of the pessimism in the RT-OSGi model on the schedulability of real-time applications, and the effect of RTSJ JVMs and standard JVMs on the response times of real-time application threads are both evaluated based on performing analysis of the chronic disease management application. Finally, the evaluation of the usability is more of a discussion of the reasons why RT-OSGi is not difficult for RTSJ and OSGi Framework developers to adopt. These evaluation areas are discussed in further detail in the remainder of this chapter.

8.2.1 Comparison of Thread Response Times in a Standard JVM and in an RTSJ JVM

The first stage of evaluating the ability of RT-OSGi in meeting the goals of this thesis, which is to provide an environment capable of developing and deploying

dynamically reconfigurable real-time systems so as to maintain high availability and thus utility of the real-time system during software maintenance and evolution, is to compare the ability of a standard JVM and an RTSJ JVM in enabling real-time threads to meet their deadlines in the absence of dynamic reconfigurability. To evaluate this, the response times of the threads within the chronic disease management application (discussed in Chapter 7) were calculated using analysis techniques for both a standard Java Virtual Machine, and on a Real-Time JVM supporting the RTSJ. Table 8.1 and the graph in Figure 8.6 show the response times of the chronic disease application's real-time threads when using fixed priority pre-emptive scheduling (of the RTSJ) and the response times of such threads when using the standard Java thread scheduling of standard Java.

As a note, the response times were not calculated by execution-time measurements but by analysis. The analysis used is discussed shortly.

Component	C (ms)	T (ms)	D (ms)	Standard JVM Response Time (ms)	RTSJ JVM Response Time (ms)
ECG	0.5	2	2	45.5	0.5
EEG	0.5	4	4	45.5	1
Pulse Oximeter	0.5	1000	1000	45.5	1.5
Thermometer	0.5	1000	1000	45.5.5	2
ECG Interpreter	100	1200	1200	1000	162
Pulse Ox Interpreter	100	1200	1200	1000	322
Thermometer Interpreter	100	1200	1200	1000	482
EEG Interpreter	100	1200	1200	1000	642
Complex Event Processor	20	2000	2000	250	674
User Interface	100	10000	10000	1000	834
Blood Glucose/Pressure	700	10350	10350	7000	3271

Table 8.1 Response Times in a Standard JVM and in an RTSJ JVM

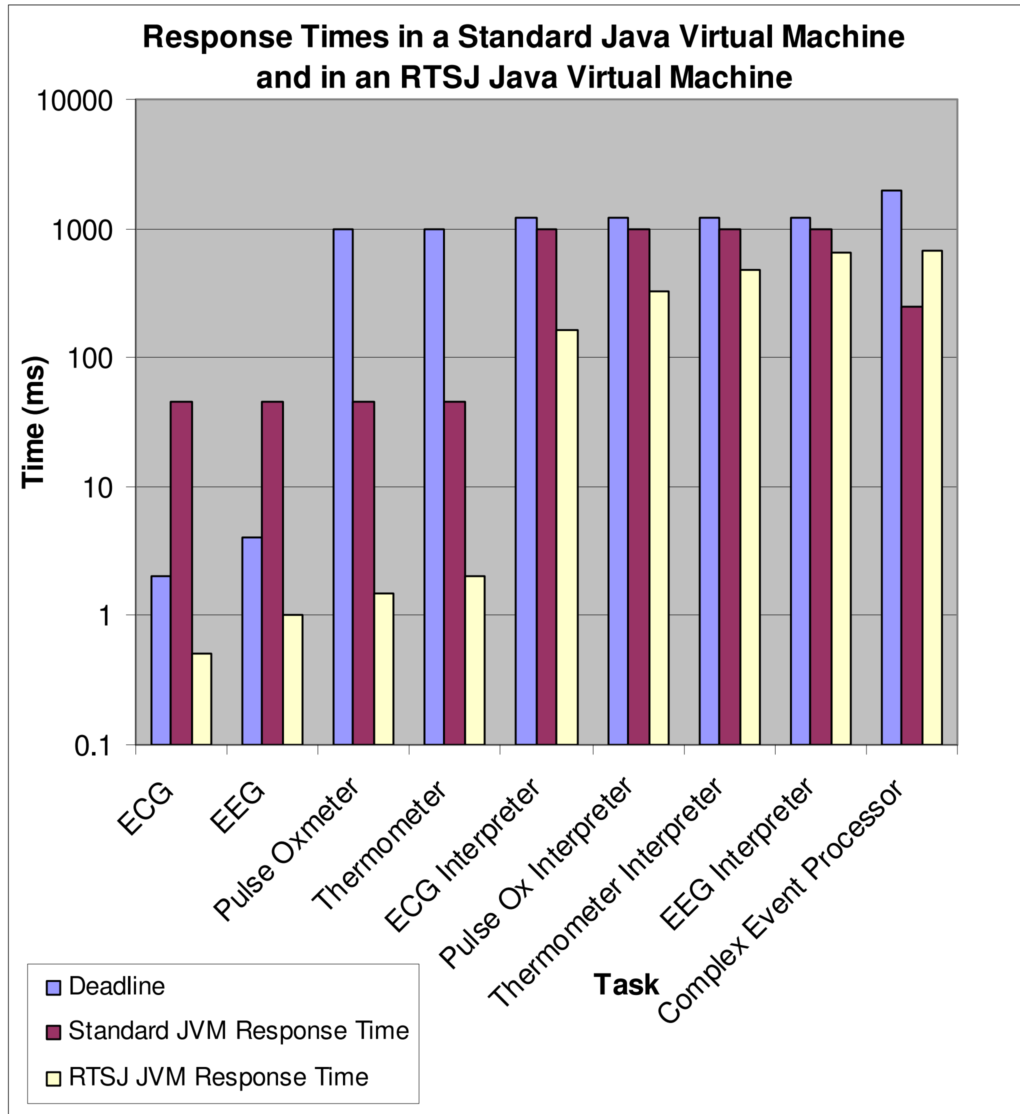

Figure 8.6 Response Times of the Chronic Disease Management Application Threads in a Standard JVM and in an RTSJ JVM

As can be seen from Table 8.1, ignoring the fact that periodic threads cannot be deployed on a standard Java JVM, the response time in a standard JVM is greater than the deadline for the first two tasks in the table (namely the ECG and EEG tasks) thus the real-time constraints of the application can not be met on a standard JVM. Furthermore, although the rest of the task set of the chronic disease management application presented in Chapter 7 meet their deadline, their response times are significantly greater than the response times achieved when deploying the application on an RTSJ JVM, with the exception of the CEP task.

The significance of the RTSJ is therefore clear in terms of the ability for RT-OSGi applications to meet their deadlines.

The reason for the difference in response times and deadline misses in the case of the standard JVM is because of the poor scheduling semantics of standard Java. As discussed in Chapter 1, no guarantee is given that the highest priority runnable thread is always executing. Furthermore, different Java priorities may be mapped to the same operating system priority [148]. This behaviour was therefore assumed when the response times for the chronic disease management application was calculated through analysis. More specifically, it was assumed that the OS utilises round-robin scheduling with a time slice/quantum of 5ms in order to schedule threads within the same priority level. Thus the chronic disease application's response times were calculated for the standard JVM case by essentially assuming that all of the threads were assigned the same native priority by the standard JVM and therefore all scheduled using round-robin scheduling. It was further assumed during the analysis that all of the chronic disease management application tasks were released simultaneously, with the task being considered for response time analysis being the last in the run queue and thus having its time slice last before the task at the beginning of the run-queue gets its second time slice (the critical instant). If there are N threads in the ready queue and the time slice equals Q, then each thread gets Q in N * Q time units The worst case response time of a thread requiring C units of computation time when scheduled using time slicing can be calculated as follows for the standard JVM case: Response time = C / Q * (N * Q).

In the case of the RTSJ JVM, it supports fixed priority pre-emptive real-time scheduling. Standard response time analysis [128] was therefore used for generating the response times. Of course, the various overheads such as context switch time and queue manipulation time etc should be taken into consideration when RT-OSGi is used outside of an academic environment..

8.2.2 Dynamic Reconfiguration Effects on Application Availability and Application Timing Constraints

Although the evaluation in Section 8.2.1 showed that RT-OSGi executing on an RTSJ implementation allows the timing constraints of real-time threads to be met, the evaluation did not take into consideration dynamic reconfiguration and its effect on timing constraints of the application. As the goals of this thesis are on the use of the OSGi Framework to improve the availability of real-time systems, the effectiveness of RT-OSGi in meeting this goal must be evaluated. Therefore, the second stage of evaluating RT-OSGi is in evaluating the RT-OSGi approach to dynamic reconfiguration by demonstrating that it is capable of allowing the real-time constraints of the chronic disease application to be met while dynamic reconfiguration takes place.

There are three cases of dynamic reconfiguration which need discussing: component addition, component removal, and component replacement. In order to evaluate theses three dynamic reconfiguration scenarios, a simple example application was implemented for execution on the RT-OSGi prototype. The purpose of this simple application is to perform experiments on it to demonstrate that application dynamic reconfiguration takes place without affecting the timing constraints of the application.

The example application used in the dynamic reconfiguration experiments is a simplified version of the EEG component replacement scenario discussed in Chapter 7. Essentially there is a component containing a data-buffer service and a thread which reads the data from the data-buffer service (reader thread). There is also a component which contains a thread which generates data and writes it to the data-buffer service (writer thread). In the component replacement scenario, there is also another component with a modified version of the thread responsible for reading data from the data-buffer service (new reader thread). Finally there is the RT-OSGi life cycle operation thread which is responsible for processing life cycle operations invoked by the user, in these examples, it is responsible for processing the request to install, remove and replace a component. The experiments are discussed in further detail in the upcoming subsections and the

temporal specification of the application used in the experiments is given in Table 8.2. Note that the example has some "slack-time" built into it and thus the experiment admittedly does not stress test the system. Although other experiments have been performed which include less slack-time, the extent to which the amount of slack-time in the system affects deadline misses has not thoroughly been investigated in this work. This is however considered an interesting line of future work.

Thread	Computation	Period	Deadline	Priority
Writer	100	400	400	4
New Reader	200	1000	1000	3
Old Reader	150	1000	1000	2
Life Cycle	300	1200	1200	1

Table 8.2 Temporal Specification of the Dynamic Reconfiguration Example Application

8.2.2.1 Component Installation

In this experiment, the component containing the reader thread and data-buffer service were initially deployed. The "install" life cycle operation was then invoked with the URL of the component containing the writer thread, and the invocation processed by the RT-OSGi life cycle operation processing thread.

In order to determine whether or not application dynamic reconfiguration in the form of component installation affects the timing requirements of the currently deployed reader thread, the reader thread was assigned an RTSJ deadline miss handler [100]. A deadline miss handler is an asynchronous event handler that is released by the JVM when the JVM detects that a schedulable object (such as a thread) has missed a deadline. The handler allows the application developer to be notified and take some action on deadline misses.

In the example, the reader thread's deadline miss handler was not released at any point during observed execution, i.e. the handler was not released prior to, during, or after invocation of the install operation. This outcome was expected. The reason for this is due to the way in which RT-OSGi processes life cycle operations. Unlike the standard OSGi Framework, the life cycle operation processing thread in RT-OSGi executes under a server with an execution-time budget and a period. Although the life cycle processing server provides no timing guarantee for the completion of life cycle operations, the life cycle processing server is included in schedulability analysis so that the effect of the life cycle operation processing on the timing constraints of application threads are bounded and taken into account. As a result, the invocation and processing of life cycle operations does not cause timing faults for application threads. In addition, for component installation, acceptance tests are used as part of admission control to ensure that the component can only be added if it will not interfere with the timing constraints of components already deployed in RT-OSGi. The acceptance tests include response time analysis (schedulability analysis), GC reconfiguration analysis, a free memory check, and a free priorities check. Furthermore, the threads in components passing the admission control acceptance tests have their memory allocation budgets and CPU-time budgets monitored and enforced to ensure that such threads do not use more resources than specified during admission control thus preventing them from using resources allocated to other components' threads.

From executing the component installation experiment and from the reasoning which support the results of the experiment, it is clear that the RT-OSGi model of life cycle processing and the admission control do indeed prevent the installation of new components from causing timing faults for real-time threads in components already deployed. Moreover, the effectiveness of response time analysis and GC reconfiguration analysis (the main acceptance tests for RT-OSGi admission control) has widely been evaluated in the literature.

As a note, since the life cycle operations can also be invoked from application threads in addition to the RT-OSGi life cycle processing thread, this scenario was also evaluated through an experiment. As was the case above, no deadline misses

were reported during the experiment. This was expected because of the fact that the install operation has been extended with admission control with CPU and memory budgets enforced (as discussed above). does. Thus the behaviour is equivalent. Moreover, the fact that the WCET of the admission control is too large to be bounded for analysis purposes is unimportant because the life cycle invoking thread (the writer thread in the experiment) calls the install operation from a non-real-time context thus its timing constraints are not affected by calling the install operation.

8.2.2.2 Component Removal

The experiment for evaluating whether or not dynamic reconfiguration in the sense of removing a component on which no other components are dependent on is essentially the same as the experiment performed for component installation evaluation discussed in Section 8.2.2.1. The only difference is that the life cycle operation invoked is the component removal operation on the component containing the writer thread that was previously installed in the experiment in Section 8.2.2.1.

The result of the experiment was that the deadline miss handler was at no point released thus indicating that the reader thread did not experience any deadline misses while the writer thread’s component was being removed. This was the expected result. The reason why deadline misses should not occur when an application is dynamically reconfigured in the sense that a component on which no other components in the application are dependent on is removed is because the life cycle thread/application thread has server parameters which are included in schedulability analysis and therefore the invocation of the uninstall operation will not cause unaccounted interference on the application threads (as discussed in Section 8.2.2.1). Furthermore, the uninstall operation does not increase the CPU load or memory load of the application nor does it affect other deployed independent components in any other way. As a result, the uninstall operation in RT-OSGi does not affect the timing constraints of the rest of the application.

If dependencies exist such that the component being removed exports Java packages which are imported and used by other application components, the component can still be safely removed without affecting the timing constraints of the rest of the application since the exported Java package persist until no importers of the package exist. If dependencies exist in terms of threads in the component being removed cooperating with threads in other application components, or if the component being removed registers a service required by other components, the component must be replaced in order to prevent timing faults in other components, component replacement is discussed in Section 8.2.2.3.

8.2.2.3 Component Replacement

In order to evaluate the ability in RT-OSGi to replace components without breaking timing constraints of other components, the component replacement process is demonstrated with the example application which was used in both the component addition and component removal experiments discussed in Section 8.2.2.1 and Section 8.2.2.2 respectively.

For the component replacement experiment, it is assumed that the component containing the reader thread needs to be replaced with a new version which contains a more optimised version of the reader thread which has a smaller computation time. Assuming the mode change protocol for service transition discussed in Chapter 7 Section 7.4.4.2 is executed, the timing requirement of the entire application, including threads in the component being replaced (the component containing the reader thread) are maintained while the old version of the reader component is replaced with a new version. Note that the fact that the new version of the component has a smaller computation-time requirement is not important. Schedulability analysis is used as part of admission control and so any changes in temporal behaviour are possible in the new version of the component provided that the system remains schedulable.

The component replacement experiment was carried out slightly differently from the component addition and component removal experiments in that the effect of component replacement on the ability of the component being replaced to maintain its real-time guarantees cannot be captured by deadline miss handlers alone. The reason for this is because in the case of component replacement, the ability of the threads in the new component to essentially take over from the threads in the old version of the component before what would have been the next deadline of the threads in the old version of the component had they not terminated must be assessed. Clearly, this behaviour cannot be assessed through deadline miss handlers alone. Instead, the experiment printed out the release times of the old reader thread and the writer thread, and once the new component was installed, the new reader thread also. The time at which the old reader thread terminated was also recorded and its next release time and deadline were calculated based on its previous release time. Based on the next release time printed for the new reader thread, its deadline was calculated and compared to what would have been the next deadline for the old reader thread. In each execution of the experiment, the new reader thread's deadline always occurred before the old thread's deadline. Furthermore, the deadline miss handler for the new reader thread was never released which demonstrates that it was able to always take over from the old reader thread without causing any timing constraint violations for the application. This result from the experiment was expected because the RT-OSGi mode change protocol along with cooperation from the application will ensure that:

1. The writer and old reader threads are able to complete their current release, i.e., the changeover process is deferred until at least the end of each threads current period
2. Action taken after being notified of the component replacement before the next release, in the case of the writer thread, copy the service state and obtain a reference to the new data-buffer service, in the case of the old reader thread, terminate.
3. New reader thread starts reading sensor values from the new data-buffer service no later than what would have been the deadline of the next release of the old reader thread.

The above steps of RT-OSGi ensure that there is no break in timing constraints between the old and new reader threads. The new one takes over from the old one, while most importantly, respecting the deadline of the old version. To demonstrate that this is the case, the possible orders of task execution and behaviours during discovery of the requirement for component replacement are illustrated and discussed.

The period of the writer thread is smaller than that of the reader thread (see Chapter 7 Table 7.1) and therefore (according to the Rate Monotonic priority assignment algorithm used in RT-OSGi) is higher priority. It is assumed that the new version of the reader thread has a period equal to the old version and thus the two versions share the same priority. As will become apparent after the upcoming discussion, the period of the new version of the reader thread could also be smaller than the old version and still guarantee that it is released before the deadline of what would be the next release of the old version of the reader thread after the release in which it terminates. The life cycle thread's period and thus priority are application dependent, in this example it is assumed that the life cycle thread has a period larger than both the reader and writer threads and therefore executes with a lower priority. Figure 8.7 shows the critical instant and the possible schedules for the example application tasks (the temporal specification of which is given in Table 8.2) during the component replacement dynamic reconfiguration.

In Figure 8.7, the critical instant for the threads used in the example application used in the component replacement experiment is shown in A) in order to show that the example is indeed schedulable. The possible schedules that the task will follow during dynamic reconfiguration are shown in B) and C). The point labelled x in the schedules in B) and C) is the point when the new version of the reader component containing the new versions of the data-buffer service and reader thread is installed, and as a result, the point at which the new reader thread is created and started. Point y is the point where the writer thread takes action as a result of discovering that the component replacement is required. Point z is the point where the old version of the reader thread takes some action as a result of discovering the need for component replacement.

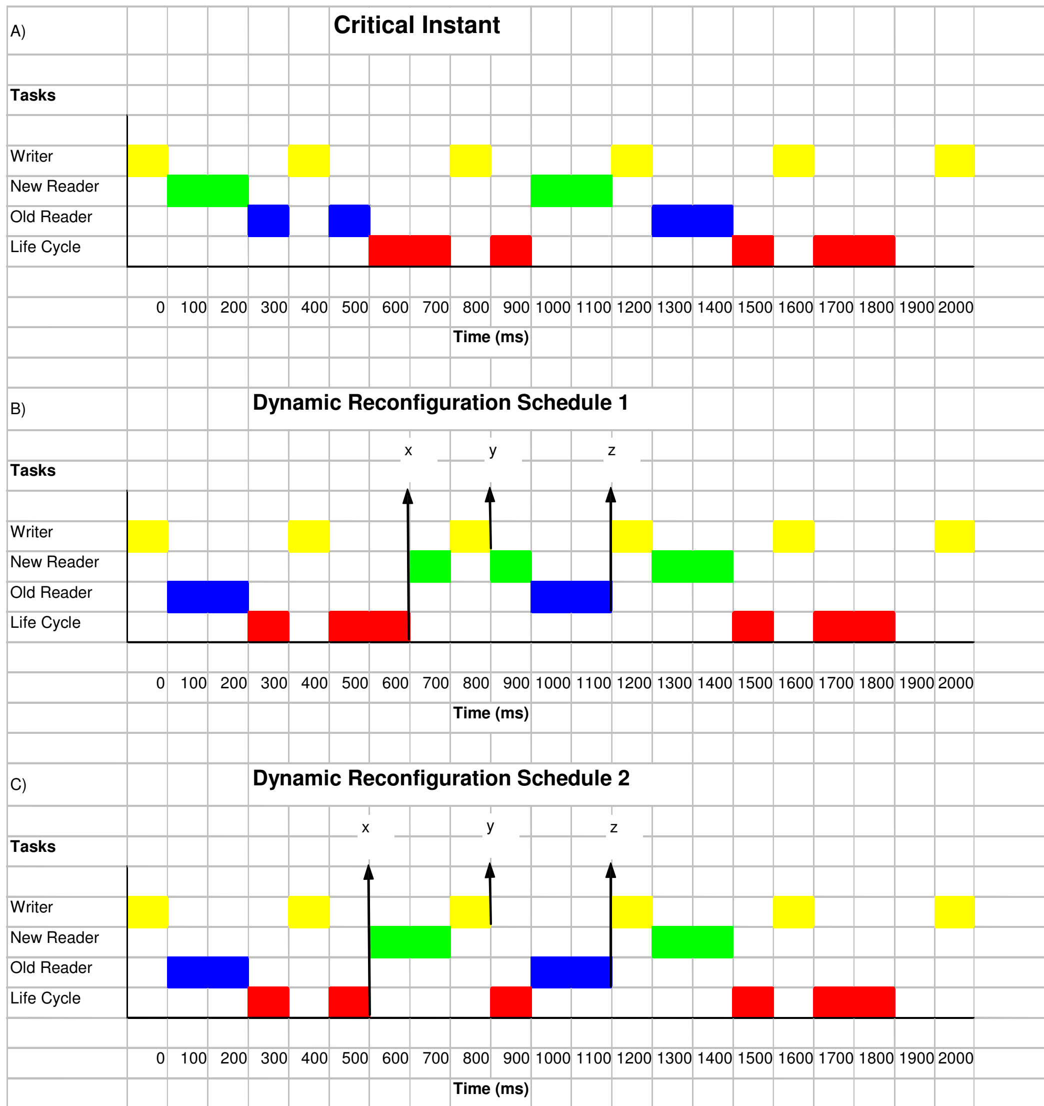

Figure 8.7 The Possible Schedules for the Example Dynamic Reconfiguration

It can be seen that regardless of whether the tasks follow scheduled B or C, the new version of the reader thread always executes before the old version, and as a result, its subsequent period will be after the point where the old reader thread determines the fact that the new reader component has been installed and that it should now terminate as part of the component replacement process, but before what would have been the old reader thread's next deadline had it not terminated. Thus there will be no break in availability of the reader functionality even during component replacement.

However, while the transition from the old to the new version of the reader component can be guaranteed to not affect the availability of the reader functionality, the application must aid in this process. For example, depending on the order of discovering the need for component replacement, i.e., depending on which thread is first to complete its period after both threads have been notified by RT-OSGi that component replacement is required, the application logic of both the old and new versions of the reader thread should differ. If the new reader thread is started and executes the data-buffer service's readSensor() method before the writer thread has discovered and transitioned to using the new data-buffer service (for example if it is not pre-empted by the writer thread before executing the service method), then the new reader thread will have no data samples to read from the new version of the data-buffer service since the writer thread will not be writing the data samples to the new service nor will it have copied the data buffer of currently written values from the old service to the new service. In this case, the new reader thread should just wait for its next release, and the next release of the old reader thread in which it discovers the new service should call the readSensor() method before terminating to allow the new reader thread to continue on its behalf.

In the case where the writer thread pre-empts the new reader thread before it calls the readSensor() method of the new data-buffer service, before completing its period, the writer thread will discover the need to transition to the new data-buffer service and will copy the service state of the old data-buffer service to the new one and will start writing sensor data samples to the new service, as a result, when the new reader regains control of the CPU, it will have sensor data available to read. As a result, the old version of the reader thread does not need to read a data sample before termination since the new reader thread will read one on its behalf no later than the end of what would have been the deadline of the next release of the old version of the reader thread.

The reason why the old version of the reader thread cannot execute before the first release of the new reader thread is because of the execution eligibilities of the threads in the application. The highest priority thread is the writer followed by the old version of the reader thread followed by the life cycle processing

thread. Since the new version of the reader thread must have a shorter or equal period, the new version of the reader thread will have a higher execution eligibility than the old version of the reader because it will be started during the life cycle processing thread thus even in the case where it has the same priority as the old version of the reader thread, it will already have started execution before being pre-empted by the writer thread, since the old version of the reader thread will then at some point be released, it will then go to the back of the run queue for its priority behind the new version of the reader thread which arrived first and has already used a fraction of its computation-time. Therefore the old version of the reader thread will never run before the new reader thread completes its first release in the example. Had the period of the new reader thread been smaller than that of the old version of the reader thread, then it would be possible that the new reader completes multiple releases before the old reader thread is next released. Based on this, it is apparent that this result generalises to any example (not just the one evaluated). Provided that the threads of a new version of a component have equal or smaller periods than the old version of the threads being replaced, the new threads will always be able to take over the duty of the old versions without affecting the timing constraints of the application. This is a strong requirement. If the new threads have a larger period then there will be a delay between the old threads terminating and the new ones taking over resulting in one or more deadline misses for the application.

In conclusion, the above evaluation of the RT-OSGi component replacement mode change protocol discussed in Chapter 7 shows that dynamic reconfiguration of an application, including the challenging case of component replacement, can occur without causing threads in the application to miss their deadlines. The cooperation of application developers is however necessary in order to ensure that the application logic is correct during mode changes where the old and new versions of threads and services temporarily coexist during the changeover process. Furthermore, the application developers also need to provide support in terms of saving and loading service state to ensure that the application logic remains correct during the changeover process for service requesting threads. The execution-time overheads of supporting component

replacement and dynamic reconfiguration in general are discussed in the following section.

8.2.3 Execution-Time Overhead of Dynamic Reconfiguration

As discussed in Chapter 5 and Chapter 6, the life cycle operations (install, update, uninstall, start and stop) require various extensions in RT-OSGi in order to support the real-time requirements of applications. However, these extensions result in an increase in the execution-time of the life cycle operations and these execution-time overheads requires evaluating.

In this section, various execution-time measurements were recorded by executing the life cycle operations of both the standard OSGi Framework and the RT-OSGi prototype on a machine with the following execution-environment: AMD Athlon i686 751MHz CPU with 256 KB of cache memory, and 246.5MB of RAM. The operating system was the Red Hat distribution of Linux, kernel version 2.6.21-57.el5rt patched with the SMP PREEMPT and RT patches. Regarding the actual execution-time measurement process, the execution-times were recorded over 1000 execution measurements, with the WCET the highest time measured in the 1000 runs. As a note, the execution-times measured in this section are specific to the Apache Felix OSGi Framework implementation on which RT-OSGi is based. If Apache Felix naively implements the OSGi Framework specification, it is possible that the execution-times of the life cycle operations will be unnecessarily large. This is discussed further in Chapter 9 in future work. As a further note, the execution-times measured in this section on overheads are not strictly the worst-case because it is difficult to be certain that the worst-case has been found based on a measurement-approach to WCET analysis, as discussed in Chapter 4.

The first set of overhead measurements was obtained by implementing empty versions (i.e. without the application logic) of the case study application components. The components simply contain the meta-data necessary for OSGi/RT-OSGi to deploy the components, as it is the execution-time of the life

cycle operations of OSGi/RT-OSGi that is of interest and not the case study application itself. Ten of these components were then deployed and the WCET was measured for the installation of the final application component (the blood glucose/pressure component) in both standard OSGi and RT-OSGi. After the installation, the other life cycle operations were also invoked on the blood glucose/pressure component and the WCETs were recorded. The WCETs of all of the life cycle operations are displayed in Table 8.3. The reason why the WCETs of the life cycle operations were measured for the chronic disease application in this way is because the execution-time of the install operation is dependent on run-time factors and it has been determined (as discussed shortly) that an application that highly utilises the CPU causes the greatest WCET for the install operation. The WCET of the other life cycle operations were also measured under this application configuration although their WCET remains the same regardless of the number of components deployed any case. Note that the percentage WCET increase is entirely due to the overhead of performing analysis as part of the admission control, i.e. analysis for the new application configuration. No analysis for the reconfiguration process itself is necessary because the resources used by the dynamic reconfiguration thread are bounded (through an eecution0-time server) and are accounted for in application schedulability analysis. As a result, the dynamic reconfiguration process will not result in deadline misses during application reconfiguration. Furthermore, the actual reconfiguration process itself remains the same in RT-OSGi as it does in the standard OSGi Framework. The implication of this is that in theory, the overhead of RT-OSGi life cycle operations could be negated by performing the admission control analysis on another computer.

Life Cycle Operation	WCET in Standard OSGi (ms)	WCET in RT-OSGi	% WCET Increase in RT-OSGi
Install	47	223	474%
Update (in active state)	128	N/A	N/A
Update (in idle state)	60	N/A	N/A
Uninstall	24	29	20.8%
Start	5	6	20%
Stop	4	6	50%

Table 8.3 WCET of the Life Cycle Operations in both Standard OSGi and RT-OSGi

As can be seen from Table 8.3, the percentage increase in execution-time is quite high for all life cycle operations in RT-OSGi in particular the install operation. As a note, the reason the execution-time overhead for the update operation in RT-OSGi is not given in Table 8.3 is because, as discussed in Chapter 5, the update operation is of little use in RT-OSGi because of its semantics. Furthermore, the reason why there are two execution-time measurements for the update operation in the standard OSGi Framework is because when a component is active, i.e. it has had the start operation called on it, calls to update on that component will then result in the update operation calling the stop operation followed by the update procedure followed by the start operation to return the component to its active state. As a result, calling update on an active component results in the execution-time overheads of the start and stop operations too hence the difference in execution-time between the two update operation entries in Table 8.3.

Despite the fact that install with admission control is always called from a non real-time context and therefore the WCET of the operation is not of much importance, further discussion of this overhead is desirable. The execution-time of the install life cycle operation for RT-OSGi shown in Table 8.3 (223 ms) is broken down into the execution-time overheads of its constituent parts so as to determine which extensions to the standard OSGi life cycle operation are mostly responsible for the high execution-time overhead. The measurements were performed such that extreme values were used as input into each constituent

algorithm in order to determine which inputs lead to the shortest and longest measured execution-times for the algorithm. The input which leads to the shortest observed execution-time of each algorithm is termed “simple scenario” and the input which resulted in the largest observed execution-time is termed “complex scenario”. The measured execution-times of these scenarios for the algorithms which constitute the extensions to the install life cycle operation are shown in Table 8.4. As a note, it is not possible to experience the execution-time associated with the complex scenario for each constituent algorithm of the install operation as recorded in the table simultaneously since, for example, the scenario which leads to the longest execution-time for the RTA algorithm would likely lead to the shortest execution-time for the GC reconfiguration analysis. This is discussed further shortly.

Admission Control Phase	**Simple Scenario Execution-Time (ms)**	**Complex Scenario Execution-Time (ms)**
Read temporal specification	0.5	0.5
Server parameter selection	14	1050
RUB	5	25
Boolean RTA	5	44
Standard RTA	5	51
GC Reconfiguration Analysis	X = 0.5 = 31 X = 5 =7 X = 50 =7	X= 0.5 = 105861 X= 5 = 8243 X= 50 = 549
Priority Range Assignment	7	15

Table 8.4 Execution-Time Overhead of Installing a Component with Admission Control

The scenarios leading to the shortest measured execution-time (simple scenario) and the longest measured execution-time (complex scenario) listed in Table 8.4 of the various algorithms which constitute the RT-OSGi install life cycle operations can be explained as follows. The simple scenario for the RUB, Boolean RTA and standard RTA algorithm occurs when a single task is deployed since an application consisting of a single task is trivially schedulable provided that the task’s execution-time is no greater than its period/deadline naturally.

Therefore the simple scenario for these algorithms was measured by executing these algorithms with only the blood glucose/pressure component. The complex scenario for these algorithms occurs when the task set used as input to the algorithms is large and heavily utilises the CPU. Therefore the largest execution-time was measured when all of the components of the chronic disease management application were deployed (which was the scenario used in determining the execution-time of the install operation in Table 8.3), which heavily utilises the CPU.

In terms of the server parameters selection algorithm, the simple scenario occurs when there is only one task as the server can be set to have the same parameters as the sole task that executes under it. The complex scenario occurs when the threads used as input to the server parameter selection algorithm have a large range of periods. The example used to measure the complex scenario was three threads with temporal parameters C =1, T = 10000, and C = 5, T = 20, and C = 1, T = 4. The execution-time is high in this case because the algorithm has to compute the CPU demand every 4 ms (smallest period, second thread has harmonic period) until the 10000ms point (largest period) is reached. This supports the results presented in [123].

With regard to the GC reconfiguration analysis, the simple scenario occurs when the system is heavily loaded. The reason for this is because the most computationally expensive part of the reconfiguration analysis algorithm is the GC parameters selection process, which attempts to find the maximum amount of CPU time that can be allocated to the GC thread by iteratively increasing the CPU allocation to the GC thread until the point where it determines that the application threads would become unschedulable. Clearly, the more loaded the CPU, the less iterations occur before the application becomes unschedulable and the algorithm terminates. Thus the shortest execution-time occurs when the system is heavily loaded. The example scenario for which the simple scenario measurement took place was the installation of the blood glucose component when all other chronic disease management application components had already been deployed. Furthermore the larger the increment used by the algorithm (X in Table 8.4, discussed in Chapter 6), the less number of iterations will be required

by the algorithm before the application becomes unschedulable. As a result, the algorithm terminates faster, albeit at the expense of potentially causing application dynamic reconfiguration to fail unnecessarily. In the simple scenario case in Table 8.4, this unnecessary failure of dynamic reconfiguration occurs when the increment X = 5 and also when X = 50.

Clearly, the inverse of this situation, i.e. the less the task set utilises the CPU, the more times the algorithm will iterate thus increasing its execution-time and leading to the complex scenario. This is especially true when the increment (X) is small e.g. 0.5 ms increments. The scenario chosen for measuring the complex scenario was therefore the installation of the blood glucose/pressure component in isolation, i.e. when no other components exist in the system.

While the complex scenario and thus the longest execution-time of the server parameter selection algorithm can easily be avoided by either having application developers not provide thread-containing components with a diverse range of periods or by having the server parameter selection algorithm generate the server parameters for a component offline (which is possible because the algorithm does not require any knowledge of threads in any other components), the complex scenario of the GC reconfiguration analysis and the complex scenario of the RTA cannot be avoided since these situations occur when the system is lightly or heavily loaded as discussed. Clearly these scenarios cannot be avoided. The effect of the large execution-time associated with the complex scenarios of the GC reconfiguration analysis and response time analysis algorithms on the execution-time of the install operation can be seen in Table 8.5 and Figure 8.8. Table 8.5 shows the installation time for each component of the chronic disease application. The first entry in the table "ECG" was the first component to be installed and the "Blood Glucose/Pressure" component was the eleventh and final component of the application to be installed. The execution-times for each component are plotted in the graph in Figure 8.8.

Component	**Installation Execution-Time in RT-OSGi (ms)**
ECG	219
EEG	132
Pulse Oximeter	128
Thermometer	126
ECG Interpreter	112
Pulse Ox Interpreter	123
Thermometer Interpreter	134
EEG Interpreter	145
Complex Event Processor	151
User Interface	160
Blood Glucose/Pressure	223

Table 8.5 Execution-Times of Sequentially Installing the Chronic Disease Management Application's Components

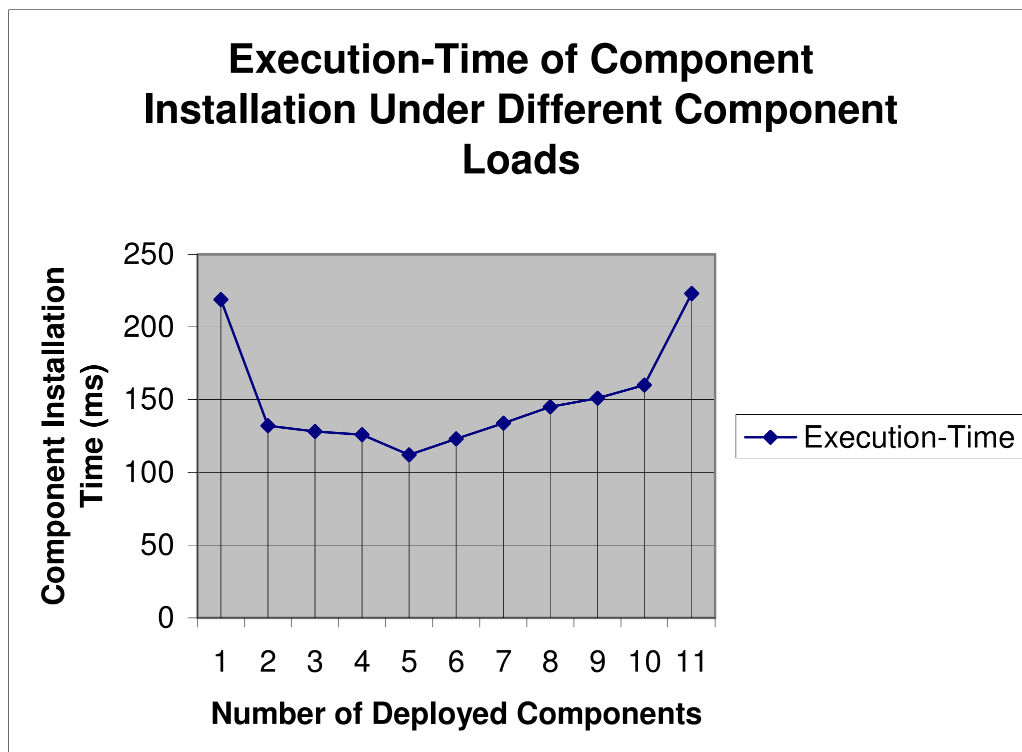

Figure 8.8 Graph Showing Changes in Execution-Time of Install Operation when the Number of Component Deployed Increases

From looking at Table 8.5 and Figure 8.8, one can see that the execution-time of the install operation in RT-OSGi is very high when the first component is installed (the ECG component in the example), then gradually decreases until the fifth component has been installed (ECG Interpreter component), and finally the execution-time increases until the installation of the last component (Blood

Glucose/Pressure). This pattern of execution-times matches the pattern of the execution-time of the install operation's GC reconfiguration analysis and response time analysis. When the first component is installed, the complex scenario (and thus largest execution-time) of the GC reconfiguration and the simple scenario (and thus shortest execution-time) of response time analysis occurs. With subsequent component installations, the execution-time of GC reconfiguration analysis decreases and the execution-time of the response time analysis increases. The final component to be installed then results in the simple scenario of the GC reconfiguration analysis and the complex scenario of the response-time analysis to occur.

In terms of drawing conclusions from Table 8.5 and Figure 8.8, one can determine that in the worst cases measured, i.e. when the first and last components in the example were installed, the execution-time overhead is very high. Although such high overhead is undesirable, the fact that these life cycle operations are not called from a real-time context means that the high overhead has no implications for real-time threads and the ability of RT-OSGi to meet the goals of this thesis. Nevertheless, in Chapter 9, a means of significantly reducing the execution-time overhead will be discussed as potential future work for RT-OSGi.

8.2.4 RT-OSGi Pessimism and Application Schedulability

From previous sections of this chapter, it has become clear that RT-OSGi is capable of supporting dynamically reconfigurable real-time applications despite its execution-time overheads. However, many assumptions made in the RT-OSGi model are very pessimistic and have an effect on application schedulability. This means that the likelihood of components passing admission control and being deployed is reduced when compared to deploying the equivalent application as a standard (i.e. non OSGi) RTSJ application. In this section, the sources of pessimism, namely, the effect of temporal isolation event handlers, overestimations in the amount of GC work required, over allocation of the CPU in the server parameter selection process, and the WCET overhead of having

threads support component replacement in RT-OSGi are discussed in the context of the chronic disease application.

8.2.4.1 Temporal Isolation Pessimism

As discussed in Chapter 4, cost enforcement and temporal isolation are not provided at the OS/JVM level and therefore are implemented at the application level in RT-OSGi. Providing the temporal isolation behaviour requires that each application component have two asynchronous event handlers associated with them in order to raise and lower thread priorities in accordance with the server budget exhaustion and replenishment. In order to evaluate the effect of providing temporal isolation at the application level, the WCET of the handlers was measured from executions and response time analysis was then carried out on the chronic disease application to determine the response times of the application threads both in the presence of the temporal isolation handlers and without the handlers in order to determine the effect that they have on application schedulability. The results are shown in Table 8.6. As a note, as discussed in Chapter 7, the temporal specifications of the chronic disease management application's threads were obtained from the medical literature and not from measurements.

In Table 8.6, the highlighted entries are the temporal isolation handlers for the chronic disease application. As can be seen, there are a large number of handlers required to support temporal isolation in the application. Furthermore, from the table it can be seen that the application is not schedulable when the temporal isolation handlers are included in schedulability analysis as the last six entries in the table have response times greater than their deadlines. A further observation from Table 8.6 is that the temporal isolation handlers have a small WCET (0.2 ms) since all that the handlers are required to do is to either raise or lower the priorities of the threads executing under the component's server. The reason why the handlers greatly affect the schedulability in the example chronic disease application is because the ECG component has a server with a very small period (2ms) and so the 0.2 ms WCET of the two handlers for the ECG component

becomes significant. It is therefore necessary to change the temporal specification or even remove the ECG component in order to retain application schedulability in the presence of the temporal isolation asynchronous event handlers. Chapter 9 will discuss future work as a means of removing the pessimism of the application level temporal isolation on application schedulability.

8.2.4.2 Server Parameter Selection Pessimism

The server parameter selection algorithm used in RT-OSGi is extremely pessimistic such that it drastically over allocates the CPU to each component. In order to evaluate the level of pessimism associated with server parameter selection, it is assumed that the four sensor interpreter threads (ECG, EEG, Thermometer and Pulse Oximeter) are required to execute under a single server rather than the current situation which is to have each thread execute under its own server with each thread's server being assigned identical parameters to the thread. The temporal specification of these four threads is reproduced in Table 8.7 (T_1 – T_4 in the table) along with a number of other fictitious threads that are assumed to require execution under a server. Table 8.8 shows the result of server parameter selection (using the analysis proposed by Zabos [123]) for the example scenario along with the CPU utilisation of the server parameters when compared with CPU utilisation of the sum of the individual threads which execute under the server. Thee results are also shown for the fictitious servers. The results from Table 8.8 are also shown in Figure 8.9.

Thread Name	Cost	Deadline	Response Time (Cost Enforcement)	Response Time (No Cost Enforcement)
ECG COH	0.1	1.1	0.1	
ECG RH	0.1	1.1	0.2	
ECG	0.5	2	0.7	0.5
EEG COH	0.1	2.1	0.799	
EEG RH	0.1	2.1	0.899	
EEG	0.5	4	1.6	1
Pulse Ox COH	0.1	500.1	1.7	
Pulse Ox RH	0.1	500.1	1.8	
Thermometer COH	0.1	500.1	1.9	
Thermometer RH	0.1	500.1	2.9	
ECG Interpreter COH	0.1	600.1	3	
ECG Interpreter RH	0.1	600.1	3.1	
EEG Interpreter COH	0.1	600.1	3.2	
EEG Interpreter RH	0.1	600.1	3.5	
Pulse Ox Interpreter COH	0.1	600.1	3.6	
Pulse Ox Interpreter RH	0.1	600.1	3.7	
Thermometer Interpreter COH	0.1	600.1	3.8	
Thermometer Interpreter RH	0.1	600.1	3.9	
Pulse Oximeter	0.5	1000	5.99	1.5
Thermometer	0.5	1000	7.39	2
CEP COH	0.1	1000.1	7.49	
CEP RH	0.1	1000.1	7.59	
ECG Interpreter	100	1200	295.39	162
Pulse Ox Interpreter	100	1200	583.6	322
Thermometer Interpreter	100	1200	874.7	482
EEG Interpreter	100	1200	1165.9	642
CEP	20	2000	2053	674
User interface COH	0.1	5000.1	5209.8	
User interface RH	0.1	5000.1	5210.1	
BP/BG COH	0.1	5175.1	5210.2	
BP/BG RH	0.1	5175.1	5211.3	
User Interface	100	10000	10099.2	834
BP/BG	700	10350	10825.2	3271

Table 8.6 Temporal Isolation Handlers' Effect on Application Schedulability

Thread	C	T	D	Server
T_1	100	1200	1200	S1
T_2	100	1200	1200	S1
T_3	100	1200	1200	S1
T_4	100	1200	1200	S1
T_5	400	1300	1300	S2
T_6	800	4600	4600	S2
T_7	1000	6800	6800	S2
T_8	200	20000	2000	S3
T_9	400	20000	2000	S3
T_{10}	2500	28000	28000	S4
T_{11}	200	1840	1840	S4
T_{12}	10	1000	1000	S4
T_{13}	190	567	567	S4
T_{14}	30	225	225	S4
T_{15}	10	30000	30000	S5
T_{16}	10	30	300	S5
T_{17}	10	150	150	S5

Table 8.7 Temporal Specification of Threads

ID	C	T	D	Threads' CPU Utilisation	Server's CPU Utilisation	Server Over allocation
S1	500	850	850	33%	58%	44%
S2	1634	2034	2034	62%	80%	23%
S3	700	10350	10350	3%	6%	50%
S4	521	568	568	67%	91%	27%
S5	115	135	135	10%	85%	89%

Table 8.8 Server Parameters and Comparison with CPU Utilisation of Threads

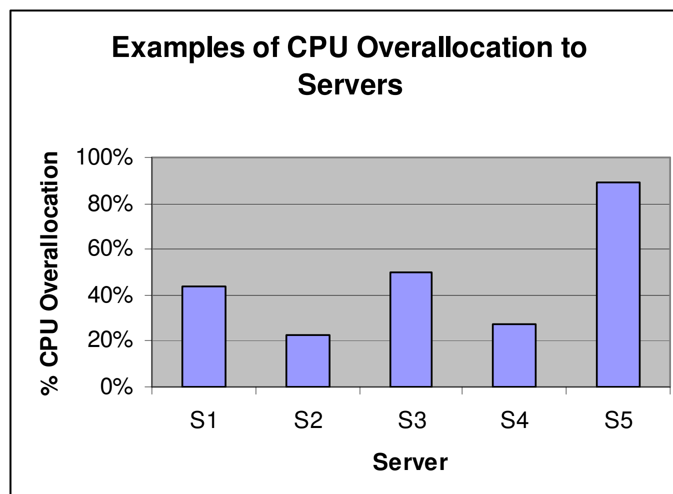

Figure 8.9 Pessimism of Server Parameter Selection

As can be seen in Table 8.8, the server parameter selection algorithm of RT-OSGi over allocates the CPU to servers. For example, the server for the four interpreter threads (T1 –T4 in Table 8.7) is assigned 44% more CPU time than is necessary. A more extreme example is the threads T15 – T17 whose server is allocated 89% more CPU time than is required. Clearly such gross over allocation of the CPU to servers has impacts on application scehdulability and encourages the practice of using as few servers as possible thus leading to a monolithic application structure which completely defies the principles of CBSE and SOA on which RT-OSGi is based. As future work, Chapter 9 will discuss the possibility of improving application schedulability by either evaluating other server parameter selection algorithms with RT-OSGi or by avoiding the use of server parameter selection algorithms altogether.

8.2.4.3 GC Reconfiguration Analysis Pessimism

In RT-OSGi, the amount of GC work to be carried out tends to be overestimated because it is pessimistically assumed that an application's root-set is the sum of each thread's stack size. Furthermore, it is also pessimistically assumed that the amount of live memory of an application equals the total allocated memory, i.e. no attempt is made to classify the allocated memory as either live or garbage memory. Overestimating the amount of GC work in this way means that the application may fail schedulability analysis because the GC thread will be allocated more CPU time than it actually needs to compete a GC cycle. The effect of this however is not evaluated because these assumptions are necessary because in the worst-case, all of the allocated memory may be live memory and the root-set may indeed be the sum of each thread's stack size.

8.2.4.4 Component Replacement Pessimism

Any schedulables in components which wish to have the capability to be replaced incur the WCET overhead associated with polling for notification of replacement, new service acquisition and possibly the transfer of service state from the old version of a service being replaced to the new version. The WCET overhead was measured as 0.3 ms in the simple reader-writer example application component replacement experiment discussed in Section 8.2.2.3 on

the hardware specification discussed in Section 8.2.3. The reason for this small overhead is because of the simplicity of the service state transfer which was simply the case of copying the data buffer containing the data written by the writer thread in the example application. In more complicated scenarios, the state transfer may be more time consuming than 0.3. Regardless of the value of the component replacement transition overhead, it must be included in the temporal specification of RT-OSGi applications' threads.

8.2.5 Backwards Compatibility with Standard OSGi and the Usability of RT-OSGi

RT-OSGi is backwards (downwards) compatible with components designed for the standard OSGi Framework. The source code and Meta-data of such components does not need to be modified in any way in order to be deployed on the RT-OSGi Framework. Careful attention was paid to the design of RT-OSGi such that modifications to the standard OSGi API were not necessary. As this is the case, and as the API is simply a set of Java interfaces and not implementations, RT-OSGi is both source and binary compatible with standard OSGi components since it is only the implementations of the OSGi interfaces that have been changed in RT-OSGi. According to the Java Language Specification [77], binary compatibility is therefore preserved as: Changing the name of a method, the type of a formal parameter to a method or constructor, or adding a parameter to or deleting a parameter from a method or constructor declaration creates a method or constructor with a new signature, and has the combined effect of deleting the method or constructor with the old signature and adding a method or constructor with the new signature. If any pre-existing binary references a deleted method or constructor from a class, this will break binary compatibility; a `NoSuchMethodError` is thrown when such a reference from a pre-existing binary is linked. Clearly, this is not the case in RT-OSGi because the OSGi interfaces are unchanged in RT-OSGi. Moreover, RT-OSGi utilises new OSGi manifest-headers in the component's manifest file (Meta-data file) which are used by the RT-OSGi to determine whether a component is real-time or not, and if so, what its temporal specification etc are. When these manifest headers are absent, as would be the case in standard OSGi components, RT-

OSGi presumes the component to be non-real-time, thus allowing the component to be deployed despite the absence of the Meta-data for RT-OSGi.

Although it is possible to deploy standard OSGi components in RT-OSGi, these components cannot be mixed with RT-OSGi real-time components. The reason for this is that such standard OSGi components contain standard Java threads which are unable to specify computation-time and memory allocation budgets. While the computation-budget is not particularly important because these threads execute with non real-time priorities and thus are unable to deprive real-time threads of the CPU, non-real-time threads may exhaust memory by having unrestricted use of the memory resource, as a result, non-real-time threads may cause temporal faults in real-time threads. Furthermore, they may hold the resolution lock, which may be required by real-time threads.

In terms of the level of difficulty for standard OSGi and RTSJ developers to learn how to develop application with RT-OSGi, every effort has been made to make this transition as simple as possible. As mentioned, the OSGi API remains the same thus the principles of OSGi programming remain the same. This is as opposed to developing a new component model on top of standard OSGi which was deemed inappropriate as it would involve having the application developers learn another component model and would reduce the likelihood of RT-OSGi being widely adopted. However, naturally, the RTSJ developer must learn the OSGi programming model and the OSGi developer must learn about the RTSJ in order to develop RT-OSGi applications. Fortunately, learning the RT-OSGi is no different from learning the standard OSGi model and so there is lots of literature available to the RTSJ developer on how to learn the OSGi programming model. Also, in the case of the OSGi developer, RT-OSGi avoids the need for Scoped Memory, the most challenging feature of the RTSJ to learn and thus learning the RTSJ is not overly challenging. As a result learning RT-OSGi should not be a difficult task for both standard OSGi and RTSJ developers.

Once RT-OSG developers are familiar with both the programming model of the standard OSGi Framework and the RTSJ, they have to take the following points into consideration when developing RT-OSGi applications. Instead of using the

RTSJ Schedulable classes, the RT-OSGi Schedulable classes (such as OSGiRTT and OSGiAEH) should be used instead. When using the RT-OSGi service model, service implementations should be registered with WCET properties and these properties should be used by service requesters as part of the service discovery process. The OSGi update operation should be avoided because its semantics result in stopping the old version of the component and then starting the new version resulting in a blackout period where the component is unavailable and thus timing faults will occur in real-time applications. The component replacement procedure discussed in Chapter 7 should instead be used to avoid this situation. The RT-OSGi's asynchronous event handling should be used in preference to the synchronous event handling of the standard OSGi Framework. The service interfaces and service factories should be annotated with WCET and memory allocation information. Application developers should be aware of the fact that, unlike the standard OSGi Framework, RT-OSGi imposes strict timing constraints on component activation. RT-OSGi will throw an exception if the activator's start and stop methods do not complete within a time-bound set by RT-OSGi. This ensures that the WCET of activating third party components is known to application developers and this prevents their threads from having Denial of Service (DoS) attacks executed on them. As a result, components requiring lengthy component activation/initialisation should create and start additional threads from the activator's methods in order to perform these lengthy computations.

8.3 Summary

In this chapter a prototype of RT-OSGi was briefly discussed. The reason it was not discussed more thoroughly in this chapter is because the prototype has essentially been discussed throughout the entire thesis.

In the prototype, the underlying standard OSGi Framework implementation on which RT-OSGi is based (Apache Felix) was modified and extended with the features discussed throughout this thesis so as to provide an environment capable of deploying dynamically reconfigurable real-time applications with high

availability requirements. Furthermore, a Java package was introduced which contains classes for application developers to use in order to target the RT-OSGi Framework.

The approach of RT-OSGi to meeting the goals of this thesis was evaluated by using analysis, by deploying and executing a simple reader-writer example application on the prototype, and by deploying "dummy" versions of the chronic disease management application components on the prototype. From this evaluation, it was found that RT-OSGi is capable of deploying applications which meet real-time requirements in both the steady state and during dynamic reconfiguration (including component replacement), and thus RT-OSGi is capable of deploying dynamically reconfigurable real-time systems with high availability requirements, the goal of this thesis. However, it was also found from execution-time measurements that the execution-time overhead of RT-OSGi is quite high, although this does not affect the ability of RT-OSGi to meet the goals of the thesis since the execution-time overhead is not encountered by real-time threads and does not occur from a real-time content. Finally, from the evaluation, it was found that the pessimistic assumptions made in RT-OSGi and the overhead of cost enforcement and temporal isolation at the application level have a profound effect on application schedulability. While this does not prevent RT-OSGi from meeting the goals of this thesis, it certainly impacts on the utility of the current design and implementation of RT-OSGi. The implications of this are discussed in Chapter 9 in the conclusions and future work.

9

Conclusions and Future Work

This chapter concludes this thesis by summarising the thesis goals, hypothesis and contributions, and by discussing the overall conclusions of the research work carried out. Future research directions are also discussed followed by a brief note on the key message that this thesis tries to convey.

9.1 Thesis Goals and Hypothesis

Real-time systems, like any other software, require software maintenance/evolution in order to remain useful. This maintenance/evolution typically must take place offline, i.e. when the application is terminated. However, taking an application offline causes it to exhibit no utility. This is undesirable. Furthermore, in real-time systems, taking the application offline and thus making it unavailable for use will likely have severe safety and/or financial implications. It is therefore desirable to be able to deploy real-time systems on a software architecture which supports application dynamic reconfiguration. Dynamic reconfiguration enables an application to be evolved without taking the application offline thus maintaining high levels of application availability. A software architecture capable of providing such dynamic reconfigurability to none-real-time systems is the OSGi Framework.

The goal of this thesis was to extend and modify the OSGi Framework to be capable of performing dynamic reconfiguration of real-time systems without the reconfiguration process affecting the timing constraints of the application such

that the application remains available for use without missing any deadlines during software maintenance/evolution. Thus the emphasis was on providing a general means of providing software evolution/maintenance to real-time systems rather than on providing a fault tolerance approach to masking software failures. The thesis hypothesis is re-stated below:

The OSGi Framework has proven ideal in developing dynamically reconfigurable Java applications based on the principles of CBSE and SOA. With dynamic reconfiguration, software applications continue to remain available and have utility even while they are undergoing maintenance/evolution. One domain where OSGi has yet to make an impact is real-time systems. By integrating the OSGi Framework with the RTSJ, and by providing certain extensions to the OSGi Framework, OSGi can be used to: develop real-time systems which are dynamically reconfigurable meaning that application maintenance can take place without taking the system offline and without affecting the application's real-time constraints. Such dynamic reconfiguration of real-time systems will allow them to remain available and have utility during software maintenance and evolution activities.

There has been little work in the context of using the OSGi Framework to develop real-time systems. The only work in this area focuses solely on providing a means of deploying native real-time threads from OSGi components However, this related work does not address how real-time guarantees can be made during dynamic reconfiguration, i.e. CPU and memory admission control, GC reconfiguration, temporal isolation, asynchronous thread termination, and mode changes etc are not addressed. As a result this related work is not capable of meeting the goals of this thesis. Other related work which is in the context of dynamic reconfigurable real-time systems but not the OSGi Framework has one of two failings. The related works either assumes that the configuration/evolution states that the application can be dynamically reconfigured to must be known pre-deployment time, or the dynamic reconfiguration does not guarantee that the real-time constraints continue to be met. In the former case, the real-time requirements of the application continue to be met but at the expense of limiting dynamic reconfiguration such that it must be pre-determined and is therefore

incapable of meeting the goals of this thesis since it is impossible to predict all of the ways in which the real-time application will require evolution/maintenance at the time that the application is first deployed. In the latter case, dynamic reconfiguration may occur in unforeseen ways but the related works fail to specify an environment that will ensure that the application will continue to meet its real-time requirements during and after dynamic reconfiguration by failing to provide many of the features proposed in this thesis such as temporal isolation, admission control and GC reconfiguration etc. Thus these works are also incapable of meeting the goals of this thesis.

Unlike the related work, the contributions of this thesis do indeed meet the goals of this thesis by providing a complete solution to performing unplanned dynamic reconfiguration on real-time applications, maintaining high levels of application availability without affecting the application's timing requirements. Furthermore, the work in this thesis is also unique in that it is the first to integrate the OSGi Framework and the RTSJ thus gaining the benefits and reaching the user-base of these technologies. The degree to which the contributions of this thesis meet the thesis goals and prove or disprove the thesis hypothesis is discussed in Section 9.2.

In terms of the generalisation of the results of this thesis, the identified issues associated with dynamic reconfiguration in the context of real-time systems can be used by any computer scientist/software engineer interested in deploying dynamically reconfigurable real-time systems in any programming language and environment. The majority of issues identified are not specific to the Java programming language, the RTSJ, or the OSGi Framework. Thus in the future for example, a framework for the dynamic reconfiguration of real-time systems developed in the Ada programming language can use the output of this research as a guideline to ensuring deadline misses do not occur during or after application dynamic reconfiguration.

In addition to the dynamic reconfiguration issues which were identified in the context of real-time systems, another result which can be generalised is the GC reconfiguration analysis. The GC reconfiguration analysis can be applied to any

dynamically reconfigurable environment such as service-oriented systems, rather than the OSGi Framework specifically.

9.2 Overall Conclusions

From implementing and evaluating RT-OSGi, it was shown that real-time systems can be developed and deployed with RT-OSGi while maintaining real-time constraints and availability during application reconfiguration. This is in contrast to typical real-time systems which would normally have to be taken offline during such maintenance/evolution.

The prototype RT-OSGi has shown through proof by construction that it is possible to actually implement the extensions to the standard OSGi Framework discussed throughout this thesis. Furthermore, the prototype and its evaluation collectively provide evidence to support the thesis hypothesis restated in Section 9.1. More specifically, in the evaluation of RT-OSGi, the comparison of the response times of threads deployed on the standard OSGi Framework on a standard JVM with the response times of the same threads deployed on RT-OSGi on an RTSJ implementation showed that the deadlines of threads can only be met with the support of a real-time JVM. In addition to the comparison in response times, the evaluation of RT-OSGi also demonstrated that during dynamic reconfiguration, RT-OSGi does not cause applications threads to miss their deadlines. This is trivially true for the cases of adding new components and removing components on which no other component is dependent on, since the only potential for causing deadline misses would be the effect of the thread executing the dynamic reconfiguration inflicting an unbounded amount of interference on application threads. However, the life cycle operation processing thread responsible for executing user-derived dynamic reconfiguration and the application threads all have execution-time budgets enforced. As a result, dynamic reconfiguration does not affect the application's availability or real-time requirements. In the case of component replacement, the evaluation of RT-OSGi showed that the threads in the new version of the component take over from the threads of the old version of the component before what would have been the next deadline of the old treads which have terminated. Although the evaluation

of component replacement was only in the context of a single reader/writer example, the result generalises to any application provided that the deadlines of the threads in the new version of the component are equal to or less than the deadlines of the threads belonging to the old component being replaced. This will always ensure that the new threads get to run before what would be the next deadline of the old version of the threads. Of course, it is the application developer's responsibility to ensure application logic correctness while both the old and new versions of the component coexist and to copy any state from the old version to the new version. Fortunately, as discussed, the OSG Framework provides the service factory methods to provide structure to this "decommission" process.

From the above discussion, it is therefore clear that RT-OSGi meets the goals of this thesis. However, as discussed in the evaluation, RT-OSGi exhibits a number of overheads. The WCET of the life cycle operations in RT-OSGi are larger than their standard OSGi counterparts since these operations are extended in RT-OSGi in order to accommodate safe dynamic reconfiguration in the context of real-time application deployment. Amongst the life cycle operations, the install operation is the only life cycle operation with significant execution-time overhead. Although install is not called from a real-time context and is therefore not an issue in terms of affecting the real-time constraints of the application, it is nevertheless undesirable to inflict such large execution-time overhead on calling threads.

As discussed in Chapter 8, the reason for the large execution-time overhead on the install operation is the GC reconfiguration analysis and response-time analysis algorithms which are part of the install operation. In the case of response-time analysis, as discussed in Chapter 5, RUB and Boolean RTA are used in place of the standard RTA algorithm in an attempt to reduce the execution-time of exact schedulability analysis. Unfortunately, no other steps can be taken to reduce the execution-time of response-time analysis thus the WCET associated with it is unavoidable in RT-OSGi. In the case of GC reconfiguration analysis, it is possible to reduce the execution-time overhead. This is discussed further in Section 9.4.

The evaluation of RT-OSGi also showed that there are a number of forms of pessimism in RT-OSGi which impact on the schedulability of applications. The most significant of these is the effect of temporal isolation. In RT-OSGi temporal isolation is provided at the application level with two event handlers required per server/component. These event handlers must be included in schedulability analysis Although the computation-time required per handler is very small, this overhead becomes significant when there exists a server in the application which has a small period since the handlers are also required to have similar sized periods in order to provide the cost enforcement for the server.This overhead is arguably the most significant drawback of the current design of RT-OSGi.

Despite the application-level temporal isolation overhead, RT-OSGi is still capable of meeting the goals of this thesis. However, the overhead is likely to reduce the total of number of components that can be deployed in an RT-OSGi application and therefore will reduce the utility of RT-OSGi.

It appears that the next release of the RTSJ will make cost monitoring a mandatory feature and thus all implementations of the specification must provide it. It is hoped that subsequent releases of the RTSJ will make cost enforcement rather than cost monitoring a mandatory feature This would alleviate the need for RT-OSGi to provide it at the application-level and thus remove the major source of overhead of RT-OSGi.

With regards to the effects of the server parameter selection, the algorithm used currently in RT-OSGi grossly over-allocates the CPU to each server thus reducing the total number of servers/components that can be deployed in RT-OSGi. Given the fact that the only reason RT-OSGi uses servers in any case is so as to reduce the application-level temporal isolation overhead by having two event handlers per server/component rather than per thread, and given the fact that, as discussed, subsequent releases of the RTSJ will hopefully make cost enforcement a mandatory feature of the specification following the announcement that the next release will make cost monitoring a mandatory feature, servers and thus server parameter selection will no longer be required in

RT-OSGi. As a result, the overhead of sever parameter selection will be removed from RT-OSGi in the future.

In conclusion, RT-OSGi in its current version is capable of developing and deploying dynamically reconfigurable real-time systems which remain available for use and continue to meet timing requirements during reconfiguration. Thus RT-OSGi applications continue to have utility during maintenance/evolution, where non-dynamically reconfigurable real-time systems would need to be taken offline and exhibit no utility during this period thus incurring financial or safety penalties. When future versions of the RTSJ are released, the design of RT-OSGi can be modified to remove the two main sources of overhead: application-level temporal isolation and server parameter selection. Furthermore the proposed future work discussed in the upcoming section, namely reducing the overhead associated with GC reconfiguration analysis and designing RT-OSGi to run on multi-core/multi-processor architectures, should increase the utility of RT-OSGi.

9.3 Future Work

As discussed in Chapter 8 and section 9.3, the GC reconfiguration analysis poses significant execution-time overheads to the install life cycle operation. The first future work proposal is to reduce this execution-time overhead. This can be achieved by extending RT-OSGi with the GC reconfiguration analysis proposed by Robertz [140] (discussed in Chapter 6) and having RT-OSGi utilise the Robertz approach and the current approach used in RT-OSGi in different circumstances.

The reason for integrating the Robertz approach with RT-OSGi is for increasing the likelihood of application components passing admission control. In the GC reconfiguration analysis used in RT-OSGi, the GC thread is assigned a period equal to the application thread with the smallest period and the maximum possible computation-time per period. This approach however may result in the application failing admission control if the period of the application thread with the smallest period is very small (i.e. less than a couple of milliseconds). In this

case, the maximum amount of computation-time that the GC thread may be assigned may be so small that the context switch overhead becomes significant and makes the GC thread execute ineffectively, resulting in the application reconfiguration being rejected. This application reconfiguration rejection is unnecessary however because the GC thread could potentially have had a larger period which makes the application schedulable. It is feasible to have the GC thread not run with the smallest period (and thus at the highest priority according to RM priority assignment) in RT-OSGi because, unlike many other applications, RT-OSGi provides cost enforcement. The result of the cost enforcement is that there is little risk of threads with higher priorities than the GC having unbounded interference on the GC thread and thus not allowing it to make progress, which is a typical concern and the reason why the GC is often assigned the highest priority. Note that this work was not carried out as part of the current RT-OSGi design because the utility of such a hybrid approach was not discovered until after experiments with the current GC reconfiguration analysis of RT-OSGi were performed.

Since Robertz's approach to GC reconfiguration analysis attempts to determine the minimum amount of CPU time that can be allocated to the GC thread that will prevent memory exhaustion, it is more likely than with the current RT-OSGi approach that components will pass CPU admission control because the GC will not utilise the CPU heavily. However, because the minimum amount of CPU time is allocated to the GC thread in Robertz's approach, the GC cycle will take a long time to complete, and as a result there will be a large accumulation of garbage in the memory. The implication of this is that the amount of free memory will become very low towards the end of the GC cycle and therefore components are less likely to pass memory admission control.

It is now evident that the GC reconfiguration analysis used in RT-OSGi is not well suited to applications which have high CPU requirements, similarly, it s evident that the approach by Robertz is not well suited to applications which have high memory requirements. Therefore, as future work, it is proposed that RT-OSGi utilise both approaches. When an application has high CPU demands, in particular when it has threads with very small periods, the approach by

Robertz should be used during admission control. Likewise, when the application has higher memory requirements, the current RT-OSGi approach should be used during admission control. This combination of GC reconfiguration approaches should improve the success rate of components passing admission control.

A further future work proposal in the context of the current GC reconfiguration analysis of RT-OSGi is to use the CPU load as a means of determining the increment (X) in the GC parameter selection process. If the application's CPU utilisation is high, the increment should not be too large since it may cause the application to become unschedulable on the first iteration of the algorithm, even though the application may have been schedulable by assigning the GC a computation-time less than X. Similarly if the application lightly utilises the CPU, a large increment is preferred to a smaller one since an increment too small will result in many unnecessary iterations of the algorithm thus unnecessarily increasing the algorithm's execution-time.

A more challenging future work proposal is to redesign RT-OSGi to support execution on multi-core/multi-processor platforms. At the inception of this project, uniprocessor systems were predominant. Since that point, multi-core/multi-processor platforms have dramatically increased in popularity and are now even commonplace in home computing. Since many of the extensions to RT-OSGi assume a uniprocessor system such as the schedulability analysis, server parameter selection and GC reconfiguration analysis, deploying RT-OSGi on a multi-core/multi-processor platform will invalidate the results of the analysis and thus will nullify any real-time guarantees of the application. Although this issue can be solved by having the affinity of the Sun Java RTS, RT-OSGi and application threads set to the same processor, the computational power of the underlying hardware will not be exploited. Therefore, it is proposed as future work that RT-OSGi by redesigned to be suitable for deployment on a multi-core/multi-processor platform.

RT-OSGi would also benefit from dynamic priority scheduling. Currently, when a component passes admission control, the priorities of currently deployed components' threads may have to be reassigned in order to ensure that the

application threads respect the fixed priority assignment policy used by RT-OSGi (Rate Monotonic priority assignment). Not doing so would invalidate the real-time guarantees that the application has. Clearly, fixed priority scheduling is not well suited to a dynamic environment like RT-OSGi. Unfortunately, the only scheduler that the RTSJ mandates is a fixed priority pre-emptive one, and no implementation of the RTSJ has yet provided additional dynamic priority schedulers. If in the future the RTSJ provides dynamic priority scheduling, RT-OSGi will be redesigned to accommodate this.

The Sun Java RTS JVM which RT-OSGi is dependent on for its GC reconfiguration capabilities only targets the Solaris and Real-Time Linux operating systems. Unfortunately these operating systems can only support soft real-time systems deployment and therefore RT-OSGi is constrained by this and can only support soft real-time constraints. However, RT-OSGi is designed for hard real-time systems. For example, response time analysis, server parameter selection and GC reconfiguration analysis all make hard real-time pessimistic assumptions and as there is no reason why RT-OSGi should preclude hard real-time systems development, it is proposed as future work that as soon as an RTSJ implementation which targets a hard real-time system begins to provide support GC reconfiguration, RT-OSGi will be evaluated in its suitability to develop real-world hard real-time systems.

Finally, a number of minor areas of possible future work already discussed throughout this thesis include: spatial isolation, adaptive resource reservation, code annotations to reduce the pessimism of GC reconfiguration analysis and memory consumption enforcement

9.4 Concluding Remarks

The contribution of this thesis, RT-OSGi, enables real-time Java applications to undergo software maintenance/evolution and general reconfiguration while remaining available for use and thus constantly providing utility to its users. The chronic disease management application case study is an example of a real-world

real-time application that has high availability requirements but yet requires reconfiguration/evolution. This case study provides evidence that RT-OSGi provides a contribution to real-world applications and is not just of theoretical interest. In its current state, RT-OSGi can be used in a practical setting although providing temporal isolation at the application level does limit the utility of RT-OSGi by restricting the size of an application due to the effect of cost enforcement event handlers on application schedulability. This will undoubtedly be addressed by future versions of the RTSJ which will mandate that implementations of the specification provide cost enforcement thus eradicating the need for RT-OSGi to provide it at the application level.

Finally, because both the RTSJ and OSGi Framework on which RT-OSGi is based are both in widespread use, and therefore result in RT-OSGi being simple for RTSJ and OSGi developers to adopt, it is hoped that RT-OSGi will open up a new market for these technologies.

References

1. Stankovic, J. and K. Ramamritham, *Hard Real-Time Systems Tutorial*, in *IEEE Computer Society Press.*, I.C.S. Press., Editor. 1988.
2. Stankovic, J.A., *Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems.* Computer, 1988. 21(10): p. 10-19.
3. Buttazzo, G.C., *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* 2004: Springer-Verlag TELOS.
4. Kopetz, H., *Real-Time Systems Design Principles for Distributed Embedded Applications*. 1997: KAP.
5. Burns, A. and A.J. Wellings, *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. 2001: Addison-Wesley Longman Publishing.
6. Lehman, M.M., *Laws of Software Evolution Revisited*, in *Proceedings of the 5th European Workshop on Software Process Technology*. 1996, Springer-Verlag.
7. Parnas, D.L., *Software aging*, in *Proceedings of the 16th international conference on Software engineering*. 1994, IEEE Computer Society Press: Sorrento, Italy.
8. ISO. *ISO 14764 Software Engineering — Software Life Cycle Processes — Maintenance* 2006 December 2010 [cited 23rd September 2009]; Available from: http://www.iso.org/iso/.
9. Swanson, E.B., *The dimensions of maintenance*, in *Proceedings of the 2nd international conference on Software engineering*. 1976, IEEE Computer Society Press: San Francisco, California, United States.
10. Liu, L., et al., *Delivering Sustainable Capability on Evolutionary Service-oriented Architecture*, in *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 2009, IEEE Computer Society.
11. Nosek, J.T. and P. Palvia, *Software maintenance management: changes in the last decade.* Journal of Software Maintenance, 1990. 2(3): p. 157-174.
12. Lientz, B.P. and E.B. Swanson, *Software Maintenance Management*. 1980: Addison-Wesley Longman Publishing Co.
13. Hosford, J.E., *Measures of dependability.* Operations Research, 1960. 8(no 1): p. 204-206.
14. Gray, J. and D.P. Siewiorek, *High-Availability Computer Systems.* Computer, 1991. 24(9): p. 39-48.
15. Nguyen, D. and L. Dar-Biau. *Recovery blocks in real-time distributed systems*. in *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*. 1998.

16. Liming, C. and A. Avizienis. *N -Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation*. in *Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on*. 1995.
17. OSGi Alliance. *OSGi Service Platform Core Specification, Release 4*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
18. Bollella, G. and J. Gosling, *The Real-Time Specification for Java.* Computer, 2000. 33(6): p. 47-54.
19. Wilson, P.R., et al., *Dynamic Storage Allocation: A Survey and Critical Review*, in *Proceedings of the International Workshop on Memory Management*. 1995, Springer-Verlag.
20. Wilson, P.R., *Uniprocessor Garbage Collection Techniques*, in *Proceedings of the International Workshop on Memory Management*. 1992, Springer-Verlag.
21. Biron, B. and R. Sciampacone. *Real-time Java, Part 4: Real-time garbage collection*. 2007 [cited March 10th 2008]; Available from: http://www.ibm.com/developerworks/java/library/j-rtj4/index.html.
22. Pizlo, F., et al. *Real-time Java scoped memory: design patterns and semantics*. in *Proceedings of the Seventh IEEE Symposium on Object-Oriented Real-Time Distributed Computing*. 2004.
23. Sun. *The Real-Time Java Platform- A Technical White Paper*. 2004 [cited 26th November 2007]; Available from: http://research.sun.com/projects/mackinac/.
24. Brosgol, B.M., R.J. Hassan, and S. Robbins, *Asynchronous transfer of control in the real-time specification for java\&trade.* Ada Lett., 2002. XXII(4): p. 95-112.
25. Baker, T.P., *Stack-based scheduling for realtime processes.* Real-Time Syst., 1991. 3(1): p. 67-99.
26. Sha, L., R. Rajkumar, and J.P. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization.* IEEE Transactions on Computers, 1990. 39(9).
27. Szyperski, C., *Component Software– Beyond Object Oriented Programming*. 1998: Addison-Wesley.
28. Heineman, G.T. and W.T. Councill, eds. *Component-based software engineering: putting the pieces together*. ed. T.H. George and T.C. William. 2001, Addison-Wesley Longman Publishing Co., Inc. 818.
29. Bachman, F., *Volume II: Technical Concepts of Component-Based Software Engineering*, in *CMU/SEI-2000-TR-008*. 2000, Software Engineering Institute, Carnegie Mellon University.
30. Box, D., *Essential COM*. 1997: Addison-Wesley Longman Publishing Co., Inc. 464.
31. OMG. *CORBA 3.1 Specification*. 2008 [cited March 6th 2008]; Available from: http://www.omg.org/spec/CORBA/3.1/.
32. Sun. *Enterprise JavaBeans Specification, Version 1.1*. 2000 [cited 1st November 2007]; Available from: http://java.sun.com/products/ejb/docs.html
33. Brown, A.W. and K.C. Wallnau, *The Current State of CBSE.* IEEE Softw., 1998. 15(5): p. 37-46.
34. Crnkovic, I., *Building Reliable Component-Based Software Systems*, ed. L. Magnus. 2002: Artech House, Inc. 454.

35. Crnkovic, I. and M. Larsson, *A case study: demands on component-based development*, in *Proceedings of the 22nd international conference on Software engineering*. 2000, ACM: Limerick, Ireland.
36. Garlan, D., R. Allen, and J. Ockerbloom, *Architectural Mismatch: Why Reuse Is So Hard.* IEEE Softw., 1995. 12(6): p. 17-26.
37. Helm, R., I.M. Holland, and D. Gangopadhyay, *Contracts: specifying behavioral compositions in object-oriented systems.* SIGPLAN Not., 1990. 25(10): p. 169-180.
38. Meyer, B., *Applying "Design by Contract".* Computer, 1992. 25(10): p. 40-51.
39. Kramer, R., *iContract - The Java(tm) Design by Contract(tm) Tool*, in *Proceedings of the Technology of Object-Oriented Languages and Systems*. 1998, IEEE Computer Society.
40. Karaorman, M. and P. Abercrombie, *jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation.* Form. Methods Syst. Des., 2005. 27(3): p. 275-312.
41. Hamie, A., et al., *Reflections on the Object Constraint Language*, in *First International Workshop on The Unified Modeling Language*. 1999, Springer-Verlag.
42. Meyer, B., *Eiffel: the language*. 1992: Prentice-Hall, Inc. 594.
43. Jones, C.B., *Development Methods for Computer Programs including a No-*
tion of Interference, PhD Thesis, in *Programming Research Group, Computer Science*. 1981, University of Oxford.
44. Kopetz, H. and N. Suri, *Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces*, in *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*. 2003, IEEE Computer Society.
45. Hissam, S.A., et al., *Packaging Predictable Assembly*, in *Proceedings of the IFIP/ACM Working Conference on Component Deployment*. 2002, Springer-Verlag.
46. Beugnard, A., et al., *Making Components Contract Aware.* Computer, 1999. 32(7): p. 38-45.
47. Franch, X., *Systematic Formulation of Non-Functional Characteristics of Software*, in *Proceedings of the 3rd International Conference on Requirements Engineering: Putting Requirements Engineering to Practice*. 1998, IEEE Computer Society.
48. Turner, M., D. Budgen, and P.l. Brereton, *Turning Software into a Service*, in *Computer*. 2003. p. 38-44.
49. Brown, A.W., et al., *Realizing service-oriented solutions with the IBM rational software development platform.* IBM Syst. J., 2005. 44(4): p. 727-752.
50. Erl, T., *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005: Prentice Hall PTR.
51. Gold, N., et al., *Understanding Service-Oriented Software.* IEEE Softw., 2004. 21(2): p. 71-77.
52. Stojanovic, Z. and A. Dahanayake, *Service-oriented Software System Engineering Challenges And Practices*. 2005: IGI Publishing.
53. Diaz, M., et al., *UM-RTCOM: An analyzable component model for real-time distributed systems.* J. Syst. Softw., 2008. 81(5): p. 709-726.

54. Schmidt, D.C. and F. Kuhns, *An Overview of the Real-Time CORBA Specification.* Computer, 2000. 33(6): p. 56-63.
55. Sandstrom, K., J. Fredriksson, and M. Akerholm. *Introducing a Component Technology for Safety Critical Embedded Real-Time Systems*. in *CBSE 2004 : component-based software engineering International symposium on component-based software engineering*. 2004.
56. Akerholm, M., et al., *Towards a Dependable Component Technology for Embedded System Applications*, in *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. 2005, IEEE Computer Society.
57. Hissam, S., et al., *Pin component technology (V1.0) and its C interface*. 2005, CMU/SEI-2005-TN-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
58. Nierstrasz, O., et al., *A Component Model for Field Devices*, in *Proceedings of the IFIP/ACM Working Conference on Component Deployment*. 2002, Springer-Verlag.
59. Muskens, J., M. Chaudron, and J. Lukkien, *A Component Framework for Consumer Electronics Middleware*. 2005.
60. Etienne, J.-P., J. Cordry, and S. Bouzefrane, *Applying the CBSE paradigm in the real time specification for Java*, in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. 2006, ACM: Paris, France.
61. Hu, J., et al., *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, Springer-Verlag New York, Inc.: Newport Beach, California.
62. Plsek, A., et al., *A component framework for java-based real-time embedded systems*, in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. 2008, Springer-Verlag New York, Inc.: Leuven, Belgium.
63. Tatibana, C.Y., C. Montez, and R.S.d. Oliveira, *Real-Time Dynamic Guarantee in Component-Based Middleware*, in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. 2007, IEEE Computer Society.
64. Tsai, W.T., et al., *Architecture Classification for SOA-Based Applications*, in *Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. 2006, IEEE Computer Society.
65. Panahi, M., W. Nie, and K.-J. Lin, *A Framework for Real-Time Service-Oriented Architecture*, in *Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing*. 2009, IEEE Computer Society.
66. Estévez-Ayres, I., M. García-Valls, and P. Basanta-Val. *An Architecture to Support Dynamic Service Composition in Distributed Real-Time Systems*. in *IEEE ISORC 2007*. 2007.
67. Ayres, I.E., et al., *Solutions for Supporting Composition of Service-Based Real-Time Applications*, in *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. 2008, IEEE Computer Society.

68. Pfeffer, M. and T. Ungerer. *Dynamic Real-Time Reconfiguration on a Multithreaded Java-Microcontroller*. in *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*. 2004.
69. Adler, R., D. Schneider, and M. Trapp, *Engineering Dynamic Adaptation for Achieving Cost-Efficient Resilience in Software-Intensive Embedded Systems*, in *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*. 2010, IEEE Computer Society.
70. Rasche, A. and A. Polze, *Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET*, in *Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2003, IEEE Computer Society.
71. Rasche, A., M. Puhlmann, and A. Polze, *Heterogeneous Adaptive Component-Based Applications with Adaptive.Net*, in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2005, IEEE Computer Society.
72. Bruneton, E., et al., *The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems.* Softw. Pract. Exper., 2006. 36(11-12): p. 1257-1284.
73. Rasche, A. and A. Polze, *Dynamic Reconfiguration of Component-based Real-time Software*, in *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. 2005, IEEE Computer Society.
74. Brinkschulte, U., E. Schneider, and F. Picioroaga, *Dynamic Real-time Reconfiguration in Distributed Systems: Timing Issues and Solutions*, in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2005, IEEE Computer Society.
75. Harbour, M.G. *FRESCOR project*. 2005 [cited 9th November 2009]; Available from: http://www.frescor.org/
76. OSGi Alliance. *About the OSGi Service Platform*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
77. Gosling, J., et al., *Java Language Specification, Second Edition: The Java Series*. 2000: Addison-Wesley Longman Publishing Co., Inc. 544.
78. Lindholm, T. and F. Yellin, *Java Virtual Machine Specification*. 1999: Addison-Wesley Longman Publishing Co., Inc. 473.
79. Liang, S. and G. Bracha, *Dynamic class loading in the Java virtual machine*, in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1998, ACM: Vancouver, British Columbia, Canada.
80. Choonhwa, L., D. Nordstedt, and S. Helal, *Enabling smart spaces with OSGi.* Pervasive Computing, IEEE, 2003. 2(3): p. 89-94.
81. JCP. *JSR 8: Open Services Gateway Specification*. 1999 [cited 5th January 2008]; Available from: http://jcp.org/en/jsr/detail?id=8.
82. Li, X. and W. Zhang, *The Design and Implementation of Home Network System Using OSGi Compliant Middleware*, in *IEEE Transactions on Consumer Electronics*. 2004.
83. JCP. *JSR 291: Dynamic Component Support for Java SE*. 2007 [cited 20th November 2007]; Available from: http://jcp.org/en/jsr/detail?id=291

84. Eclipse Foundation. *Eclipse Project*. 2001 [cited 18th May 2010]; Available from: http://www.eclipse.org
85. ProSyst. *serve@Home*. 2009 [cited May 2009]; Available from: http://www.prosyst.com/success_stories/SuccessStory_Prosyst_BSH.pdf.
86. IBM. *IBM WebSphere*. 2009 [cited May 2009]; Available from: http://www-01.ibm.com/software/websphere/.
87. BMW. *BMW ConnectedDrive*. 2009 [cited May 2009]; Available from: http://www.bmw.com/com/en/insights/technology/connecteddrive/overview.html.
88. Buttazzo, G.C., *Rate monotonic vs. EDF: judgment day.* Real-Time Syst., 2005. 29(1): p. 5-26.
89. Cai, H. and A. Wellings, *Temporal Isolation in Ravenscar-Java*, in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2005, IEEE Computer Society.
90. Gui, N., et al., *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, ACM: Leuven, Belgium.
91. Gui, N., et al., *A hybrid real-time component model for reconfigurable embedded systems*, in *Proceedings of the 2008 ACM symposium on Applied computing*. 2008, ACM: Fortaleza, Ceara, Brazil.
92. Kung, A., et al., *Issues in building an ANRTS platform*, in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*. 2006, ACM: Paris, France.
93. Miettinen, T., D. Pakkala, and M. Hongisto. *A Method for the Resource Monitoring of OSGi-based Software Components*. in *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*. 2008. Parma, Italy.
94. Kaiser, R. *Combining partitioning and virtualization for safety-critical systems*. Technical Report White Paper 2007 [cited; Available from: http://www.sysgo.com/news-events/whitepapers/.
95. Goldberg, A. and G. Horvath. *Software Fault Protection with ARINC 653*. in *Aerospace Conference, 2007 IEEE*. 2007.
96. Strosnider, J.K., J.P. Lehoczky, and L. Sha. *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*. in *IEEE Real-Time Systems Symposium*. 1987.
97. Sprunt, B., L. Sha, and J. Lehoczky, *Aperiodic task scheduling for Hard-Real-Time systems* Journal of Real-Time Systems, 1989.
98. Strosnider, J.K., J.P. Lehoczky, and L. Sha, *The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments.* IEEE Trans. Comput., 1995. 44(1): p. 73-91.
99. Davis, R.I. and A. Burns, *Hierarchical Fixed Priority Pre-Emptive Scheduling*, in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. 2005, IEEE Computer Society.
100. Wellings, A., et al. *Cost enforcement and deadline monitoring in the real-time specification for Java*. in *Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2004. Proceedings*. 2004.

101. Wellings, A., Y. Chang, and T. Richardson. *The Impact of Resource Reservation Contracts on the Real-Time Specification for Java*. in *The 7th International Workshop on Java Technologies for Real-time and Embedded Systems*. 2009. Madrid, Spain.
102. JCP. *JSR 121: Application Isolation API Specification*. 2001 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=121.
103. JCP. *JSR 284: Resource Consumption Management API*. 2005 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=284.
104. Richard-Foy, M., *Partitioning JVM Preliminary Specification*. 2009. p. Personal Communication.
105. Santos, O.M.d. and A. Wellings, *Cost Monitoring and Enforcement in the Real-Time Specification for Java - A Formal Evaluation*, in *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. 2005, IEEE Computer Society.
106. JCP. *JSR 282: RTSJ version 1.1*. 2009 [cited 13th January 2011]; Available from: http://jcp.org/en/jsr/detail?id=282
107. Siebert, F., *JamaicaVM Cost Monitoring*. 2009. p. Personal communication.
108. Wellings, A.J. and M.S. Kim, *Processing group parameters in the real-time specification for Java*, in *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*. 2008, ACM: Santa Clara, California.
109. Welch, I. and R.J. Stroud, *Kava - using byte code rewriting to add behavioural reflection to Java*, in *Proceedings of the 6th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 6*. 2001, USENIX Association: San Antonio, Texas.
110. Masson, D. and S. Midonnet, *RTSJ extensions: event manager and feasibility analyzer*, in *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*. 2008, ACM: Santa Clara, California.
111. Puschner, P. and A. Burns, *Guest Editorial: A Review of Worst-Case Execution-Time Analysis.* Real-Time Systems, 2000. 18(2): p. 115-128.
112. Lindgren, M., H. Hansson, and H. Thane. *Using measurements to derive the worst-case execution time*. in *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*. 2000.
113. Wilhelm, R., et al., *The worst-case execution-time problem: overview of methods and survey of tools.* Trans. on Embedded Computing Sys., 2008. 7(3): p. 1-53.
114. Dianes, J. and M. Diaz. *ServiceDDS*. in *13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing (ISORC)*. 2010. Seville, Spain.
115. Birman, K. and T. Joseph, *Exploiting virtual synchrony in distributed systems*, in *Eleventh ACM Symposium on Operating systems principles (SOSP '87)*. 1987. p. pp. 123-138.
116. Sun. *Java Virtual Machine Profiler Interface (JVMPI)*. 2010 [cited 25th October 2010]; Available from: http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html.
117. Briones, J.F., et al., *Quality of Service Composition and Adaptability of Software Architectures*, in *Proceedings of the 2009 IEEE International*

Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. 2009, IEEE Computer Society.

118. Lima, G., E. Camponogara, and A.C. Sokolonski, *Dynamic Reconfiguration for Adaptive Multiversion Real-Time Systems*, in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*. 2008, IEEE Computer Society.
119. Liu, J.W.S., et al., *Algorithms for Scheduling Imprecise Computations.* Computer, 1991. 24(5): p. 58-68.
120. Zhou, J., et al., *Rule-Base Technique for Component Adaptation to Support QoS-based Reconfiguration*, in *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. 2005, IEEE Computer Society.
121. Davis, R.I. and A. Burns. *An Investigation into Server Parameter Selection for Hierarchical Fixed Priority Pre-emptive Systems*. in *16th International Conference on Real-Time and Network Systems. Rennes, France*. 2008.
122. Liu, J.W.S., *Real-Time Systems*. 2000: Prentice Hall PTR. 624.
123. Zabos, A. and A. Burns. *Towards bandwidth optimal temporal partitioning*. 2009 [cited 27th October 2009]; Available from: http://www.cs.york.ac.uk/ftpdir/reports/2009/YCS/442/YCS-2009-442.pdf.
124. Lipari, G. and E. Bini. *Resource Partitioning among Real-Time Applications*. in *15th Euromicro Conference on Real-Time Systems (ECRTS'03), .* 2003.
125. Almeida, L. and P. Pedreiras, *Scheduling within temporal partitions: response-time analysis and server design*, in *Proceedings of the 4th ACM international conference on Embedded software*. 2004, ACM: Pisa, Italy.
126. Sha, L., et al., *Real Time Scheduling Theory: A Historical Perspective.* Real-Time Systems, 2004. 28(2): p. 101-155.
127. Liu, C.L. and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* J. ACM, 1973. 20(1): p. 46-61.
128. Joseph, M. and P.K. Pandya, *Finding Response Times in a Real-Time System.* The Computer Journal, 1986. vol 29(5).
129. Davis, R.I., A. Zabos, and A. Burns, *Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems.* IEEE Trans. Comput., 2008. 57(9): p. 1261-1276.
130. Bini, E. and S.K. Baruah. *Efficient Computation of Response Time Bounds under Fixed-Priority Scheduling*. in *15th International Conference on Real-Time and Network Systems*. 2007.
131. Stankovic, J.A., K. Ramamritham, and M. Spuri, *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. 1998: Kluwer Academic Publishers. 273.
132. Saewong, S., et al. *Analysis of Hierarchical Fixed-Priority Scheduling*. in *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*. 2002.
133. Chang, Y., *Garbage Collection for Flexible Hard Real-Time Systems, PhD Thesis*, in *Department of Computer Science*. 2007, University of York.
134. Aicas. *JamaicaVM User Documentation*. 2010 [cited 15th January 2010]; Available from: http://www.aicas.com/documentation.html.

135. Siebert, F. *The impact of realtime garbage collection on realtime java programming*. in *ISORC*. 2004.
136. IBM. *IBM WebSphere Real Time for RT Linux Version 2 User Guide*. 2007 [cited 26th November 2008]; Available from: http://publib.boulder.ibm.com/infocenter/realtime/v2r0/index.jsp?topic=/com.ibm.rt.doc/html/realtime/introduction.html.
137. Sun. *Sun Java Real-Time System 2.2 Technical Documentation*. 2009 [cited 15th January 2010]; Available from: http://java.sun.com/javase/technologies/realtime/reference/rts_productdoc_2.2.html.
138. Henriksson, R., *Scheduling Garbage Collection in Embedded Systems, PhD Thesis*. 1998, Lund University.
139. Kim, T., et al., *Scheduling garbage collector for embedded real-time systems.* SIGPLAN Not., 1999. 34(7): p. 55-64.
140. Robertz, S.G. and R. Henriksson, *Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems*, in *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. 2003, ACM: San Diego, California, USA.
141. Nilsen, K., *Using Java for Reusable Embedded Real-Time Component Libraries.* Crosstalk The Journal of Defense Software Engineering, 2004: p. 13--18.
142. Schoeberl, M., *Scheduling of hard real-time garbage collection.* Real-Time Syst. 45(3): p. 176-213.
143. Nilsen, K., *Improving abstraction, encapsulation, and performance within mixed-mode real-time Java applications*, in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. 2007, ACM: Vienna, Austria.
144. Luckham, D., *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. 2002: Addison-Wesley.
145. McKenna, P. *Brainwave-reading headphones need no batteries* 2008 [cited 10th October 2010]; Available from: http://www.newscientist.com/article/dn13738-brainwavereading-headphones-need-no-batteries.html.
146. Nonin. *Avant® 4000 Wireless Tabletop Pulse Oximeter*. 2011 [cited; Available from: http://www.nonin.com/ProductDetail.aspx?ProductID=12.
147. Healthwatch, B. *ECG & Arrhythmia Services* 2011 [cited; Available from: http://www.broomwellhealthwatch.com/index.php.
148. Wellings, A., *Concurrent and Real-Time Programming in Java*. 2004: John Wiley & Sons.
149. Cryer, P.E., *Mechanisms of Hypoglycemia-Associated Autonomic Failure and Its Component Syndromes in Diabetes.* Diabetes, 2005. 54(12): p. 3592-3601.
150. Cryer, P.E., S.N. Davis, and H. Shamoon, *Hypoglycemia in Diabetes.* Diabetes Care, 2003. 26(6): p. 1902-1912.
151. Pramming, S., et al., *Glycaemic threshold for changes in electroencephalograms during hypoglycaemia in patients with insulin dependent diabetes.* British Medical Journal (Clinical research ed.), 1988. 296(6623): p. 665-667.

152. Howorka, K., et al., *Severe hypoglycaemia unawareness is associated with an early decrease in vigilance during hypoglycaemia.* Psychoneuroendocrinology, 1996. 21(3): p. 295-312.
153. Juhl, C.B., et al., *Automated detection of hypoglycemia-induced EEG changes recorded by subcutaneous electrodes in subjects with type 1 diabetes"The brain as a biosensor*, in *Diabetes research and clinical practice*. 2010, Elsevier Scientific Publishers. p. 22-28.
154. Penzel, T. and R. Conradt, *Computer based sleep recording and analysis.* Sleep medicine reviews, 2000. 4(2): p. 131-148.
155. Sha, L., et al., *Mode Change Protocols for Priority-Driven Preemptive Scheduling.* Real-Time Systems, 1988. 1: p. 243-261.