An Admission Control Protocol for Real-Time OSGi

Thomas Richardson, Andy Wellings

Real-time Systems Research Group
University of York UK
tom@york.ac.uk
andy@york.ac.uk

***Abstract*— In previous work we motivated the need for using the OSGi Framework with the RTSJ to develop real-time systems. We found a number of issues with using these technologies together. One of the issues we discovered was unbounded dynamism caused by the absence of admission control. Components can be uninstalled, installed and updated without regulation. This means that it is impossible to guarantee resources to components.**

In this paper, we propose a solution to the unbounded dynamism problem by providing an admission control protocol for real-time OSGi. We also provide a priority assignment approach to support temporal isolation. The combination of admission control and temporal isolation ensure that it is safe to update components or install components into the system in terms of guaranteeing resources to components. We show the practicality of our admission control protocol by implementing a prototype and measuring the execution time overhead incurred when performing a component install with admission control.

I. INTRODUCTION

The OSGi Alliance is an open standards organization whose member companies include IBM, NTT, Hitachi, Ericsson, Mitsubishi, Siemens, and Sun. The OSGi Alliance produced the OSGi Framework [1], which is a run-time environment developed in Java. More precisely, it is a component framework with an intra-JVM service model, allowing Java applications to be developed as a number of service requesting and service providing Java components.

As the OSGi Framework has its roots in both Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA), dynamically reconfigurable Java applications can be developed. Components can be installed, removed, and updated at run-time (these operations are known as life cycle operations), and components can acquire and release services at run-time. Such dynamic application reconfiguration can also be controlled remotely.

Due to the powerful capabilities of the OSGi Framework and the diverse range of companies comprising the OSGi Alliance, the OSGi Framework is having success in a number of different domains. For example [2]: Integrated Development Environments (IDEs) [3], in home automation products [4] , in enterprise systems [5], and in the automotive industry [6]. One domain where the OSGi Framework has not been used, but would benefit, is real-time systems. Perhaps the two biggest motivating factors for using the OSGi Framework to develop real-time systems are dynamic reconfigurability and remote controllability. The ability to dynamically reconfigure a real-time application is useful for generally evolving the application at run-time. Components can be added, removed and updated where necessary. For example this is useful for managing resources during mode changes, removing unnecessary components during mode change reduces memory consumption which is vital in embedded systems (which many real-time systems are). The second benefit of using the OSGi Framework to develop real-time applications is remote controllability.

The application configuration can be dynamically changed from a remote location. 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. Another use of remote controllability of the OSGi Framework is in evolving mass produced embedded systems For example, in consumer electronics, where millions 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.

Despite the benefits of using these technologies together, there is one clear problem. OSGi applications are composed of a number of Java components and services, however, 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. The Real-Time Specification for Java (RTSJ) [7] solves these issues by providing extensions to standard Java.

As the RTSJ enables real-time systems to be developed in the Java platform, and the OSGi Framework is Java based, it was thought that developing real-time systems with the OSGi Framework was simply a case of developing components and services with the RTSJ instead of standard Java. However it was shown in [8] that even when using these two technologies together problems remain which prevent real-time systems from being developed, such as unbounded dynamism, runaway threads, and issues with worst case execution time (WCET) calculation.

In this paper we address the issue of unbounded dynamism. Currently, components can be installed, uninstalled, and updated at any time. This means that no component can be guaranteed resources. However in a real-time component-based application, each component requires such resource (CPU time, memory etc) reservations in order to guarantee that its threads will meet their deadlines. Therefore 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.

To guarantee components resources we propose the use of admission control in the OSGi Framework. Admission control [9] is a mechanism which attempts to control the load on the processor by using an acceptance test, in our case, at each component installation. The test checks the schedulability of the system in the presence of the component to be installed. If the component set is schedulable the component can be installed otherwise installation is rejected. In this way, the acceptance test prevents the domino effect of installing a component which would cause other components' threads to miss their deadlines.

In Section 2, we discuss related work, with a particular focus on research relating to the OSGi Framework and temporal isolation. The OSGi life cycle operations are discussed in Section 3. These are the operations which need to be extended with admission control. Our admission control protocol is explained in Section 4, Section 5 runs through an example of the admission control in use, and provides a performance evaluation. Section 6 concludes the paper and Section 7 describes future work

II. Related Work

Our work on admission control in this paper builds on the work carried out in [8]. In reference [8] the use of the OSGi Framework and RTSJ together is explored, giving the motivation for and problems with using these technologies together. The paper's main focus is on temporal isolations, that is, how the threads in one component can be prevented from interfering with the temporal requirements of threads in other independent components in the system. This is imperative in the OSGi Framework as installed components may belong to different independent applications.

The approach taken to provide temporal isolation is to implement execution-time servers [10-12] (which act as a resource reservation mechanism) using the RTSJ's Processing Group Parameters (PGP) and application level implemented cost enforcement. The general idea is to create a PGP for each component, and assign a CPU time budget per period. The component's threads join the processing group (PGP) so that they can use no more than the group CPU budget per period. To enforce this, the JVM monitors the CPU time used by the threads in the processing group, when the budget is used the threads in the group have their priorities lowered to a background priority such that the threads in other components in the system can make progress. When the group budget is replenished at the start of the PGP period, the threads have their priorities raised to their original values. This ensures that threads don't use more CPU time than they specified they would require. The temporal isolation is a form of runtime policing enforcing the resource limits they specified on admission control, which is the topic of this paper.

The temporal isolation approach of [8] requires hierarchical scheduling [13]. In this paper we extend the work in [8] by showing how the necessary hierarchical scheduling for temporal isolation can be provided using the default semantics of the RTSJ. In particular we outline a priority assignment approach which mimics hierarchical scheduling. The priority assignment is discussed in this paper because it is part of admission control, priority ranges are assigned to components on admission, and admission may be rejected on the grounds of insufficient free priorities available for the component.

In addition to extending the temporal isolation work carried out in [8], we also rely on it in that admission control is meaningless without cost enforcement and temporal isolation. It is of little use having admission control allowing components to be installed or updated only when sufficient resources are available, when without cost enforcement and temporal isolation, components are free to use more than the resources they were guaranteed and are able to use the guaranteed resources of other components in the system.

The work carried out in this paper on admission control is similar to the work carried out in the Frescor project [14]. Frescor differs from our work in that it is concerned with scheduling dynamic flexible applications, that is, separate applications are added and removed at run-time, and each application is flexible i.e. applications do not have fixed resource requirements but can tolerate having their resource allocation varied at run-time as resources are used and freed by other applications. The Frescor project is also not in the context of the RTSJ, OSGi, SOA, or CBSE. Other work relating to scheduling dynamic applications include [15-18].

Finally, there has been some work on both resource reservation in the OSGi Framework and on using the OSGi Framework to develop real-time systems. JSR 121 [19] defines an "Isolate" API which allows for multiple isolated computations (Isolates) to execute within a single JVM, with each Isolate having its own logical heap space. JSR 284 [20] 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 [21] concerning the integration of such application isolation and resource management within a JVM. The work in progress investigates the idea of running the OSGi Framework on such a partitioning resource reserving JVM, allowing components to be installed in separate partitions, but only when there are enough resources available. In reference [22], Gui et al looked at using the OSGi Framework in the development of real-time systems. The motivation for their work is the same as ours, to develop dynamically reconfigurable real-time systems. They propose a real-time component model over the OSGi Framework. In their model, XML is used by component developers to declaratively configure real-time tasks. The functionality of real-time tasks is developed in native code, and non-real-time tasks are developed using Java. Unlike our work, neither [21] nor [22] are in the context of the RTSJ.

III. OSGi Life Cycle Operations Review

In the OSGi Framework a component can be installed, started, updated, stopped, and uninstalled, these are termed life cycle operations. Life cycle operations can be performed either by another component in the system or by an OSGi Framework implementation dependent manner such as 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.

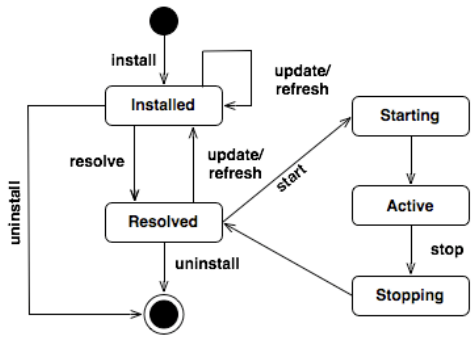

Figure 3.1 OSGi Component State Transitions

Each component state transition is explained below [1]:

- 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.
- Resolving a component – If a component needs to import packages, the Framework checks whether any resolved components have exported those packages, if so the component is resolved and the component is eligible for starting.
- Starting a component – Once a component has been resolved, it can export any packages it chooses. The Framework also creates a BundleContext object for the component, calls the start method of the component's activator, and passes in BundleContext as the argument. While the start method is being executed, the component is temporarily in a starting state. A component moves into active state on successful return of the activator.
- 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.
- Updating a component – one of the most important features of the OSGi Framework is the capability of updating a component at run-time, which allows a new component to take over. This is essential for bug fixes and feature enhancements in deployed components.

IV. Admission Control

In this section we discuss the admission control in detail. The type of analysis that must be performed is dependent on the life cycle operation. For example, when a component is being installed its resource requirements must be gathered, a check needs to be performed to see whether the system is able to guarantee a component its required resources, and finally the component must be assigned a range of priorities for its threads in keeping with our model of temporal isolation. We therefore discuss admission control by explaining the required extensions to each life cycle operation: install, start, stop, update, and uninstall.

Incorporating admission control and temporal isolation provision into the OSGi Framework can be seen as an attempt to have the Framework enforce the temporal characteristics of RTSJ components. This is necessary since without such extensions to the Framework, RTSJ components are likely to experience timing faults. However in order for the OSGi Framework to enforce the temporal behaviour of components there must be some means for the Framework to extract a components temporal specification. For our analysis, we require the computation time (C), period/minimum inter arrival time (T), and deadline (D) of all real-time threads and asynchronous event handlers within a component.

The necessary temporal characteristics are stored in a file called RealTimeDefs. The characteristics defined in this file are what the OSGi framework uses in real-time analysis and attempts to enforce. It is therefore essential that the component developer ensures that the values stored in this file are identical to those defined in their RTSJ source code. A deviation may result in timing faults.

Since admission control only needs to be performed for real-time components, we need some way of differentiating real-time and non-real-time components. Our approach is to assume that components are non-real-time by default, and that components which are real-time can specify this with a header in the component's manifest file. This allows pre-existing non real-time components to be deployed unmodified. This backward compatibility is essential since software reusability is the crux of component based software engineering and thus of the OSGi Framework.

At this point it may be unclear why we need to provide admission control in the OSGi Framework since the RTSJ provides limited admission control in the form of online feasibility analysis. The main problem with this is that the default feasibility analysis is typically inadequate. The RTSJ default analysis is that any threadset not containing aperiodic threads is feasible, although some implementations of the RTSJ provide more useful feasibility analysis (such as utilization based tests). However we prefer to avoid using RTSJ implementation specific features wherever possible as they may be changed in later releases of the implementation.

A. Component Install

Once the OSGi Framework deems a component to be real-time (by retrieving the RealTime manifest header) and reads the RealTimeDefs file, the Framework can then perform some analysis to ensure it is safe to admit (install) the component.

1) Server Parameter Selection: As mentioned we implement temporal partitioning amongst components by using execution-time servers. 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.

In the server parameter selection stage, we must select server parameters which meet the demand of the component's threads' processing requirements whilst the processing time reservation is minimised. In other words, we are trying to reserve enough of the CPU's processing time for a component's threads without being too pessimistic such that the CPU is over allocated to each component thus reducing the total number of components that can be installed in the system. The server parameters selected (C,T) can later be assigned to the component's server.

In terms of the actual server parameter selection process, there are a number of approaches available in the literature such as [23-26]. Approaches to server parameter can be classified as either offline or online. Offline server parameter selection algorithms focus on optimality, finding server parameters which make a component's threads schedulable whilst minimising CPU usage. Such algorithms (as the name implies) execute before the component is deployed. On the other hand, online algorithms execute at runtime selecting pessimistic server parameters (i.e. use more CPU resource than is necessary) but the selection algorithm has a much shorter more predictable computation time. Since we believe that a thread's WCET will be affected by a number of factors unknown until run-time, we choose to use the online approach presented by Almeida [25].

2) *Schedulability Analysis:* The server parameters selected essentially represents a component's resource requirements. Before the server parameters can actually be assigned to the server, the OSGi Framework must check that there is sufficient CPU time available so that the set of servers remain schedulable. Since servers can be treated as periodic tasks for analysis purposes, Response Time Analysis (RTA) [27] can be used to determine exact schedulability. However RTA can take a long time to execute and therefore is most suited to being performed offline. Unfortunately because components can be installed, uninstalled and updated at run-time i.e. we are concerned with dynamically reconfigurable systems, we require an online algorithm.

Davis et al [28] improve the execution performance of RTA such that it can be performed online. This is achieved by introducing new initial values for the RTA such that the algorithm terminates quicker. The schedulability test is online and efficient but it is considered to be a boolean test, that is, the test will determine the exact schedulability of the system but the response times themselves cannot be used in any further analysis. As we are interested in schedulability and not the response times, we have chosen this approach.

To further improve efficiency we use a sufficient schedulability test proposed by Bini and Baruah in [29] known as the response time upper bound (RUB). This test performs faster than the exact RTA therefore this test is performed first on each server. As the test is sufficient, failing the test does not mean that the system is definitely not schedulable therefore servers which fail this test must undergo RTA. If any server fails RTA, because the response time is greater than its deadline, then the system is definitely not schedulable and the component installation is rejected. If on the other hand, all servers have a response time less than their deadline (all servers pass either the RUB or RTA test), the system is schedulable and the component can be installed.

By reducing the number of times RTA (exact analysis) must occur, the time to deem the server set schedulable or not is substantially reduced. This means that the time to determine whether a component can be admitted is reduced.

3) Priority Range Calculation: Upon the system being deemed schedulable in the presence of the component to be installed, the next stage of installation is to calculate priority ranges. Our approach to providing temporal isolation through the use of execution-time servers requires hierarchical scheduling. Currently, the RTSJ does not support this, although proposals have been made to introduce a contract mechanism that would facilitate our admission control protocol [30]. Here we must build hierarchical scheduling using only the default fixed priority preemptive scheduler of the RTSJ. The general idea is to use the component's server's period to influence the priorities assigned to a component's threads such that the threads of components with smaller server periods receive higher priorities than threads belonging to components with longer server periods[1]. This gives the illusion of the system scheduler scheduling components each of which has a local scheduler which schedules its own threads. This is the behaviour we require with execution time servers.

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 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

[1] Typically there is a direct relationship between the periods of the threads within a component and the periods of the execution-time server. Usually the periodic releases of the threads coincide with the replenishment of the server.

installed components along with the priority range that it occupies. This list is sorted based on increasing server period. Table 4.1 shows 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 4.1 PERIOD TO PRIORITY RANGE MAPPING RULES

The priority range mapping rules we use 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.

Subsequent components passing the admission test are added to the list of currently installed components, its position is determined based on the size of its 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.

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, then range reassignments will be necessary.

B. Component Start

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 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 a server created for it. 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.

2) Thread Initialisation: As discussed, 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 code. What we propose is for the component developer to assign relative priorities to threads starting from zero for the thread with the longest period upwards to the number of required unique priorities -1 for the thread with the shortest period. For example if there are four threads the thread with the shortest period will be assigned four and the thread with the longest period will be assigned zero. This ensures that the relative ordering of priorities between threads is correct. We then propose that for each component, the OSGi Framework stores an array of the priorities in the components 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 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 [8]) 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.

C. Component Stop

A general OSGi Framework problem is that it has no control over the threads that are started from installed components. It is entirely possible that a component's threads continue to execute after their component has been stopped. Such "runaway" threads are a resource leak using up CPU time and memory. Worse still, 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 to develop real-time systems. As a note, every component that is in the process of being uninstalled must first be stopped. This is why "runaway" threads are discussed in this section.

A thread termination mechanism is required to ensure that threads cooperate with the life cycle of their component.. In standard Java there are no safe ways of doing this. The RTSJ introduces Asynchronous Transfer of Control (ATC) which allows a thread to cause another thread to abort its normal processing code and transition to some exception processing code. Such an asynchronous transfer of control is safe because certain methods such as synchronized methods defer the ATC until after they have finished executing. This means that before ATC takes place any locks being held are released.

We propose that ATC be used for the asynchronous termination of threads (ATT) when a component is stopped in the OSGi Framework. When a thread is constructed it passes a self reference to its component. When a component is stopped it can iterate through its list of thread references calling `interrupt`. This causes the thread to terminate its `run` method and execute its `interruptAction` method. In order to ensure that threads are cooperative with such a scheme we extend the classes in [8]. For example the class OSGiRTT is extended to implement the `Interruptible` interface and is made abstract such that subclasses must now provide implementations of `run` and `interrupAction` 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` is called which will result in the subclasses' `Interruptible run` method being executed.

As a note, since ATC can be deferred, when a component is stopped, its threads may not terminate immediately. We therefore recommend that component developers avoid using long ATC deferred methods. We also minimise the impact of long running ATC deferred methods executing after the component has been stopped by disabling the replenishment event's timer in the stop method, when the server's budget is used, the thread will continue to execute at background priority until it terminates on return from an ATC deferred execution context.

D. Component Update

When a component is updated, the temporal specification of its threads may change, furthermore, the thread set may change. When such changes take place, to the scheduler, the component appears to be completely new. As a result, the same analysis as that associated with a component install must take place: server parameter selection, schedulability analysis, and priority range selection.

We can improve on the above situation by checking whether the existing server parameters are sufficient to make the updated component schedulable i.e. we can test to see whether the existing guaranteed resources for the component are adequate for the updated component. For this we use the analysis proposed by Saewong [31]. Saewong et al propose an analysis to test whether a given set of threads are schedulable under a given server's parameters. Only if the installed component's existing server parameters fail to make the updated component's threads schedulable do we need to perform the more exhaustive analysis associated with component install. We could of course make the update process more efficient by associating importance with updates: important updates are always carried out, whereas updates of minor importance are only carried out if the component's existing resource guarantees are sufficient for the updated component i.e. components with a minor importance level failing the Saewong analysis have the update request rejected .

As a note, only when the existing threads' temporal specification changes, or when the threadset itself changes do we need to perform any analysis. Therefore such changes can be tested for before performing any of the update and install analysis.

E. Component Uninstall

If we were to provide some form of dynamic reclamation of resources, then when a component is uninstalled we would have the resources used by the component being uninstalled distributed to the other active components in the Framework. However, we have not investigated the use of such a feature. The only activity that takes place during component uninstall is the removal of the component's server.

V. EXAMPLE AND EVALUATION

In this section we will demonstrate the admission control protocol with an example. In the example we assume that a number of components are already installed so as to make the example more interesting. Table 5.1 shows the temporal specifications of the currently installed components.

Server	C (ms)	T (ms)	D (ms)
S1	200	1020	1020
S2	100	3100	3100
S3	150	5000	5000

TABLE 5.1 TEMPORAL SPECIFICATIONS OF INSTALLED COMPONENTS' SERVERS

We now wish to install a new component. The first step of component installation is to read its temporal specification from the component's RealTimeDefs file. This is shown in Table 5.2.

Thread	C (ms)	T (ms)	D (ms)
T1	100	1000	1000
T2	800	4600	4600
T3	1000	6800	6800

TABLE 5.2 COMPONENT'S THREADS' TEMPORAL SPECIFICATION

The next step is to find the component's resource (CPU) requirements from its threads' temporal specification. This is achieved by performing server parameter selection which will generate a budget and period for the component's server, which must be sufficient to ensure that the component's threads meet their deadlines. The server period generated for the taskset in Table 5.2 is 1300 ms, and the server budget generated is 900 ms.

The system must now check that it can guarantee the component sufficient CPU time (900ms every 1300ms). To do this, a sufficient schedulability test (RUB) is performed on each server in the system. Only if the test fails must exact schedulability analysis (response time analysis) be performed on the server. Table 5.3 shows the results of this analysis. All servers (except the highest priority server) fail the RUB test (have a response time greater than their deadline). This is because the RUB test does not perform well when the taskset utilization is high, as in this example. As a result response time analysis is also performed, and as the table shows, the response time of all servers is less than or equal to their deadline and therefore the set of servers is schedulable.

Server	D (ms)	RUB (ms)	Response Time (ms)
S1	1020	200	200
S4- new	1300	1319.5122	1300
S2	3100	4817.568	2500
S3	5000	8625.473	3850

TABLE 5.3 RUB AND RESPONSE TIME OF ALL SERVERS IN THE SYSTEM

In the final phase, the component being installed must have its priority range set. The priority range of each component including the one generated for the component being installed

is shown in Table 5.4. As a note, the installation of S4 causes S2 to have its priority range reassigned.

Server	Start Priority in Range	End Priority in Range
S1	25	26
S4	21	23
S2	15	19
S3	2	6

TABLE 5.4 PRIORITY RANGES TO BE USED BY EACH SERVER'S THREADS

The component has now been successfully installed, and the component can be started.

In terms of evaluating the admission control extensions to the install component life cycle operation, we developed a prototype implementation of the admission control process. The execution time required for a component install without admission control was recorded as an average of 44.5ms. The execution time of each phase of admission control for a component install was also recorded, along with the percentage increase the admission control has on the execution time of component install without admission control. This is shown in Table 5.5

Admission Control Phase	Execution Time Overhead (ms)	Increase in Execution Time of Install (%)
Read temporal specification of component	0.2ms	0.4
Server parameter selection	9.9ms	22.2
Exact RTA (not used in OSGi) Boolean RTA RUB	10.6ms 4.8ms 1.2ms	n/a 10.7 2.6
Priority Range Assignment	5.1ms	11.4
Total	21.2ms	47.6

TABLE 5.5 EXECUTION TIME OVERHEAD OF INSTALLING A COMPONENT WITH ADMISSION CONTROL

The execution time for each phase was calculated for 1000 runs, and the average time taken. These times where recorded from executing the prototype implementation on a laptop with 2GB memory and Intel Core Duo 1.67GHz CPU. The admission control was executed with the scenario discussed in the previous example.

From Table 5.5 it is clear that the overhead of admission control is small regarding the actual execution time of the admission control algorithms (21.2ms). However, the percentage increase in running a component install with admission control compared with no admission control is quite high, a total increase in execution time of 47.6% over the admission control-less component install algorithm. Such an increase was predicted since the component install algorithm is lightweight and performs very little processing. Regardless, we still feel it will be necessary to investigate the effects of such an increase in execution time of a component install on components' threads' temporal behaviour. This is future work.

VI. SUMMARY AND CONCLUSION

Through its use of components and services, the OSGi Framework enables dynamically reconfigurable Java applications to be developed. In addition, the OSGi Framework can be remotely controlled allowing dynamic reconfiguration of software to be controlled from a remote location. Such dynamism and remote controllability are useful features for developing real-time systems and although using the RTSJ to develop OSGi components and services solves the problems associated with using standard Java in the real-time systems domain, there are still a number of problems to be solved.

Admission control is necessary for developing real-time systems with the OSGi Framework, yet the OSGi Framework does not provide it. In this paper we showed how the OSGi Framework life cycle operations can be extended so as to provide admission control. Figure 6.1 gives a summary of the extensions to each life cycle operation.

We showed the effectiveness of the admission control extensions through an example of installing a component with admission control. From measuring the execution time of the admission control, it can be seen that using admission control leads to a fairly steep percentage increase in the execution time of a component install although in terms of actual execution time, the admission control runs quite fast.

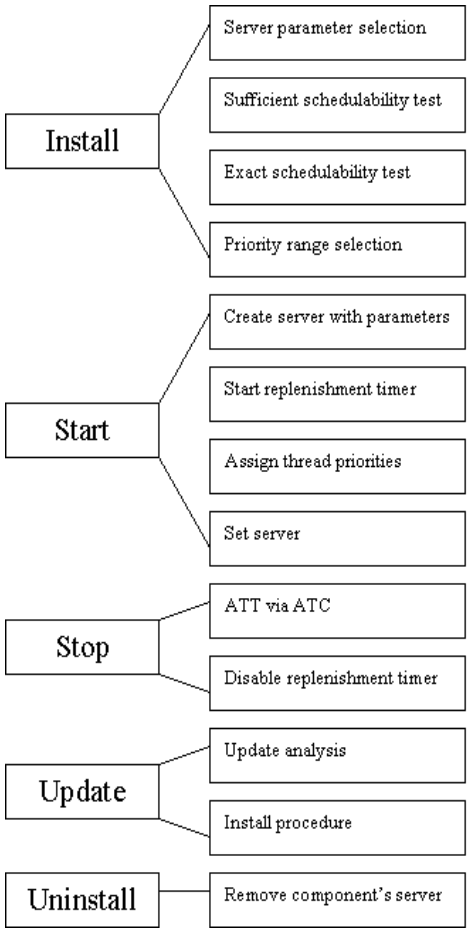

Figure 6.1 Summary of life cycle operation admission control extensions

VII. Future Work

Admission control and temporal isolation are just two of a series of issues which need to be resolved before the OSGi Framework can be used to develop dynamically reconfigurable real-time applications. We envisage the following future work:

- WCET Calculation – a number of factors unknown until run-time affect a thread's WCET, for example a thread's use of services. This means that the server parameters associated with a thread may need to be changed at run-time in order to take into account the thread's changing WCET.
- Admission Control for Memory Resource – the admission control in this paper is concerned only with the CPU resource. However a component requires some method of specifying its memory requirements, and the OSGi Framework must take these memory requirements into account during admission control. Furthermore, the OSGi Framework/VM must enforce the specified memory usage.

Acknowledgement

This work is supported by the EPSRC and IBM through CASE award 0700092X.

References

1. OSGi Alliance. *OSGi Service Platform Core Specification, Release 4*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
2. OSGi Alliance. *About the OSGi Service Platform*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
3. Eclipse Foundation. *Eclipse Project*. 2001 [cited; Available from: http://www.eclipse.org
4. ProSyst. *serve@Home*. 2009 [cited May 2009]; Available from: http://www.prosyst.com/success_stories/SuccessStory_Prosyst_BSH.pdf.
5. IBM. *IBM WebSphere*. 2009 [cited May 2009]; Available from: http://www-01.ibm.com/software/websphere/.
6. BMW. *BMW ConnectedDrive*. 2009 [cited May 2009]; Available from: http://www.bmw.com/com/en/insights/technology/connecteddrive/overview.html.
7. Bollella, G. and J. Gosling, *The Real-Time Specification for Java*. Computer, 2000. **33**(6): p. 47-54.
8. 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.
9. Stankovic, J.A., K. Ramamritham, and M. Spuri, *Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms*. 1998: Kluwer Academic Publishers. 273.
10. Sprunt, B., L. Sha, and J. Lehoczky, *Aperiodic task scheduling for Hard-Real-Time systems* Journal of Real-Time Systems, 1989.
11. 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.
12. Strosnider, J.K., J.P. Lehoczky, and L. Sha. *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*. in *IEEE Real-Time Systems Symposium*. 1987.
13. 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.
14. Harbour, M.G. *FRESCOR project*. 2005 [cited 9th November 2009]; Available from: http://www.frescor.org/
15. Shin, I. and I. Lee. *Compositional Real-Time Scheduling Framework*. in *25th IEEE International Real-Time Systems Symposium*. 2004.
16. Kuo, T.-W. and C.-H. Li, *A Fixed-Priority-Driven Open Environment for Real-Time Applications*, in *Proceedings of the 20th IEEE Real-Time Systems Symposium*. 1999, IEEE Computer Society.
17. Kuo, T.-W. and A.K. Mok, *Incremental Reconfiguration and Load Adjustment in Adaptive Real-Time Systems*. IEEE Trans. Comput., 1997. **46**(12): p. 1313-1324.
18. Deng, Z. and J.W.S. Liu, *Scheduling real-time applications in an open environment*, in *Proceedings of the 18th IEEE Real-Time Systems Symposium*. 1997, IEEE Computer Society.
19. JCP. *JSR 121: Application Isolation API Specification*. 2001 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=121.
20. JCP. *JSR 284: Resource Consumption Management API*. 2005 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=284.
21. Richard-Foy, M., *Partitioning JVM Preliminary Specification*. 2009. p. Personal Communication.
22. 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.
23. Lipari, G. and E. Bini. *Resource Partitioning among Real-Time Applications*. in *15th Euromicro Conference on Real-Time Systems (ECRTS'03),* . 2003.
24. 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 Renne, France*. 2008.
25. 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.
26. 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.
27. Joseph, M. and P.K. Pandya, *Finding Response Times in a Real-Time System.* The Computer Journal, 1986. **vol 29**(5).
28. 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.
29. 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.
30. 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.
31. Saewong, S., et al. *Analysis of Hierarchical Fixed-Priority Scheduling*. in *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*. 2002.