Providing Temporal Isolation in the OSGi Framework

T. Richardson, A.J. Wellings
Real-time Systems Research Group
Department of Computer Science
University of York – UK
{tom, andy}@cs.york.ac.uk

J.A. Dianes, M. Díaz
Department of
Computer Science
University of Malaga – Spain
{jdianes, mdr}@lcc.uma.es

ABSTRACT

The OSGi Framework is a run-time environment for deploying service-containing Java components. Dynamically reconfigurable Java applications can be developed through the Framework's powerful capabilities such as installing, uninstalling, updating components at run-time, and allowing the substitution of service implementations at run-time. Coupled with the capability to be remotely managed, the OSGi Framework is proving a success in a variety of application domains. One domain where it is yet to make an impact is real-time systems. Despite the fact that OSGi components and services can be developed using the Real-Time Specification for Java (RTSJ), there are still a variety of problems preventing the use of the Framework to develop real-time systems. One such problem is a lack of temporal isolation. This paper focuses on how temporal isolation can be provided in the OSGi Framework as a first step towards using the Framework to developing real-time systems with the RTSJ.

1. INTRODUCTION

Component-Based Software Engineering (CBSE) [1] and Service-Oriented Architecture (SOA) [2] are becoming effective ways of developing software. An example of the emergence of these development paradigms is the OSGi Alliance's OSGi Framework [3]. The OSGi Alliance was founded in 1999. It is an open standards organization, whose member companies include IBM, Siemens, Intel Corporation, BEA Systems, Nokia, and Sun etc. The Alliance created the OSGi Framework (now also known as JSR-291 Dynamic Component Support for Java SE [4]). OSGi was initially an acronym for "Open Service Gateway initiative", this was to reflect the intended use of the Framework in service gateways (JSR-8: Open Services Gateway Specification [5]). In service gateways [6], there is a gateway device (hosting the Framework) which interfaces an internal network and the Internet. The Framework operator can then download and run service-containing components from the Internet to communicate with the devices attached to the internal network. This means of service provision is (for example) useful for providing services to the home such as home-based healthcare. However, OSGi is no longer considered an acronym since the Framework has found many uses in addition to service gateways.

JTRES'09, September 23-25, 2009, Madrid, Spain

The OSGi Framework is an intra-JVM service-oriented component framework written in Java, which is used to develop highly dynamic Java applications. The reason for the integration of service oriented concepts and component-orientation is that the service-oriented approach introduces dynamism and substitutability [7] into an otherwise static component framework.

Service-orientation introduces dynamism by allowing components to be installed, updated, and uninstalled from the OSGi Framework at run-time. Substitutability is introduced by allowing service implementations to be substituted (replaced) at run-time. Such dynamism and substitutability is achievable because each component uses a separate class loader [8, 9]. For a more detailed discussion of class loaders, see [10].

Using the OSGi Framework to develop a Java application as a number of service-providing and service-requesting components is very advantageous, this can be seen through the wide ranging application domains for the OSGi Framework such as [11]: in Integrated Development Environments (IDEs) [12], in home automation products [13], in enterprise systems [14], and in the automotive industry [15]. However one domain where the power of the OSGi Framework is yet to be exploited is in real-time systems development.

Currently, OSGi applications are developed as a number of Java components and services. Unfortunately, it is generally accepted that standard Java is unsuitable 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) [16] solves these issues by providing extensions to standard Java. As the RTSJ enables real-time systems to be developed in the Java platform, and as the OSGi Framework is Java based, this leads to the question of is it possible to use the RTSJ and OSGi Framework together in order to develop dynamically reconfigurable real-tine Java applications? And if so, what is the motivation for integrating these technologies? Section 2 answers these questions giving both the motivation for and problems of using the OSGi Framework to develop RTSJ applications. Section 3 introduces temporal isolation both at the thread and component levels. In Section 4 we present extensions to the RTSJ to enable temporal isolation to be provided in the OSGI Framework. Section 5 reiterates the need for using the OSGi Framework in RTSJ application development through the use of a nuclear power plant case study. The case study illustrates the uses of dynamism and remote controllability of the OSGi Framework in the real-time systems domain.

Finally, Sections 6 and 7 conclude the paper and give some ideas for future research respectively.

2. OSGi FRAMEWORK AND REAL-TIME

2.1 Review and Motivation

The OSGi Framework allows the development of dynamically reconfigurable Java applications. It would be beneficial to use the OSGi Framework to develop dynamically reconfigurable real-time Java applications [17] using the OSGi Framework and Real-Time Specification for Java (RTSJ) together.

Using the OSGi Framework and RTSJ to develop dynamically reconfigurable real-time systems is useful not only for generally evolving a real-time system at run-time in order to undertake maintenance of software components, but also useful for managing resources during mode changes. For example when it is necessary to change to another mode of operation, the required components can be installed, and unnecessary existing components can be removed. Such dynamic reconfiguration ensures that only necessary components are installed at any one time. Minimizing the number of components (known as Bundles in the OSGi Framework) installed saves memory, which is useful in embedded systems which typically have resource constraints such as small amounts of memory.

Also the OSGi Framework can be controlled remotely i.e. applications can be dynamically reconfigured 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. As an example, consider a nuclear power plant monitoring system. Without remote control of the OSGi Framework, it would be necessary to enter the plant, which would involve being exposed to large amounts of harmful ionizing radiation. Another use of remote controllability of the OSGi Framework is in evolving mass produced embedded systems, such as consumer electronics. 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.

An additional benefit of using the OSGi Framework in real-time systems development is that it enhances the modularity offered by CBSE. Modularity is enhanced by creating separate class loaders for each component, giving component developers the choice of sharing or hiding their Java packages with other components in the system.

Further motivations for using the OSGi Framework to develop real-time systems are those stemming from the fields of Component-Based Software Engineering (CBSE) and Service-Oriented Architecture (SOA). As the Framework uses concepts from both of these fields, the real-time Java applications developed with the Framework will receive the benefits of those development paradigms. The crux of CBSE and SOA is building software as a composition of reusable building blocks (components in the former and services in the latter). One such advantage of using the OSGi Framework then is software reusability. 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.

2.2 Related Work

JSR 121 [18] 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 [19] 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 [20] concerning the integration of 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. In our paper, we are providing temporal isolation at a high level since such partitioned resource reserving JVMs are not mainstream. However such a partitioned resource reserving JVM would be far more efficient and would be the likely target of our real-time OSGi Framework.

In [21], 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. This approach gives a split architecture, real-time tasks are under the control of the real-time operating system, and non-real-time tasks run in the OSGi Framework. This is different from our work. Instead of using native code and a real-time component model, we are using the RTSJ to write real-time components.

In Section 5 of [22], Kung et al describe ideas for providing cost enforcement in the OSGi Framework, but at the VM level. Like our work, they wish to provide resource guarantees to each component. However, unlike the work by Kung et al and Gui et al, we chose to modify the OSGi Framework to provide a suitable real-time environment.

Miettinen et al [23] 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. This work is similar to ours in that they are providing cost monitoring at the component level, however the motivation for our works differ. Miettinen et al are interested in 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. Since they are not using the RTSJ, they do not attempt to provide cost enforcement nor temporal isolation amongst components.

Other, less related works, include dynamically reconfigurable real-time systems both within the context of Java [17] ,and outside of the context of Java [24]. There is a also a European project based on the dynamic reconfiguration of networked embedded systems [25]. There have also been a number of works relating to the use of component-based software engineering in real-time systems development. Whilst the majority of this research has been on providing a component model and a mapping from the model to an underlying OS, in recent years, there has been some research on using component models with the RTSJ [26-28]. For example, in [26], Etienne et al propose a component model which abstracts RTSJ memory management issues, simplifying the development process.

2.3 Challenges

This section discusses a number of reasons why it is not possible to develop dynamically reconfigurable real-time systems by simply writing OSGi services and components using the RTSJ. Instead, the OSGi Framework must be modified and extended to provide a suitable real-time environment for deploying real-time components and services.

2.3.1 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 Analysis or Deadline Monotonic Analysis [29]. 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 1 shows how the priory ordering within C1 is correct and the ordering in 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.

Table 1 Priorities assigned by component developers and by the system

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

2.3.2 Worst Case Execution Time (WCET) Analysis- Unknown WCET

Service requesting components compile to a service interface, the implementation of the service is unknown until the service is acquired at run-time. This is a major problem for building predictable systems. If the implementation is unknown then the WCET of that implementation is also unknown. The service requester requires the WCET of all services it uses for schedulability analysis.

2.3.3 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.

2.3.4 No Temporal Isolation - DoS 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 [30] in the OSGi Framework. Temporal isolation is discussed further in Sections 3 and 4. 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.

2.3.5 OSGi Framework Non-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. Although many issues will come to light after a thorough study of an OSGi Framework implementation, we have already identified some potential issues:

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.

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 important 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.

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.

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. This approach however is not adequate for developing real-time systems. On uninstalling a component, the OSGi Framework needs to safely terminate all of the threads associated with a component, should the component developer forget to.

3. TEMPORAL ISOLATION

3.1 Thread Level Temporal Isolation

Section 2 explained why developing real-time applications with the OSGi Framework and RTSJ is not simply a case of using these two technologies together i.e. it is not possible to develop predictable applications by simply using the RTSJ to develop the individual components. In order to solve those problems, modifications to the OSGi Framework and extensions to the RTSJ are required. In this paper, we focus on the extensions to the RTSJ.

Temporal isolation prevents the timing misbehaviour in one thread from affecting the timing constraints of other independent threads. In a component-based system it is imperative that threads overrunning their CPU budget in one component do not cause the threads in another component to miss their deadlines.

One means of providing temporal isolation is through the RTSJ's cost enforcement [31]. Cost enforcement monitors a thread's CPU usage and deschedules the thread if it overruns its CPU budget. Unfortunately, cost enforcement is an optional feature of the RTSJ, and we know of no implementation of the specification that provides it. The reason for this is most likely because not all hardware architectures/operating systems support it. Another feature of the RTSJ which is also optional but which is implemented by at least one VM is cost monitoring [32]. Cost monitoring is similar to cost enforcement. 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.

Cost enforcement-like functionality and thus temporal isolation can be provided at a higher level than the VM/OS level by using cost monitoring and a cost overrun handler (the code to be executed upon a cost overrun) for each thread in the system. Within the cost overrun handler, cost enforcement-like functionality can be provided by using one of the following approaches[33]:

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 temporal isolation, but there are two major problems:

1. Temporal isolation is provided at the thread level. However, since we are concerned with component-based systems, we would like to work at the component level i.e. we would like to take action on all of the threads in the overrunning component, not just the thread that caused the overrun.
2. The cost enforcement-like functionality must be provided by the component developer. Cost monitoring simply informs the thread that it has overrun, the component develop may chose to ignore this. This may happen for two reasons: firstly, because providing cost enforcement-like functionality requires an extra coding 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 OSGi Framework take the responsibility of providing temporal isolation.

The solutions to the above problems are discussed in Sections 3.2 and 4.

3.2 Component Level Temporal Isolation

To efficiently support component-based real-time systems, the RTSJ requires some semantic changes to for example provide hierarchical schedulers and resource contracts. This would allow components to have priorities and negotiate the resources required to schedule its threads. Such an approach is discussed in [34]. Alternatively, component-based real-time systems can be developed using the current semantics of the RTSJ by providing a mapping between the logical priorities of components and the actual priorities of a component's threads that are used by the default fixed priority preemptive scheduler. Although this approach is inefficient since a complete priority reassignment must take place every time a new component is installed in the OSGi Framework, we have chosen this approach in this paper as it is implementable with the current RTSJ.

The RTSJ has the ProcessingGroupParameters (PGP) [35] class. This class allows multiple threads (Schedulables) to be grouped together, and assigned a group budget per period. With cost monitoring, the PGP acts as an accounting mechanism for the threads in the group, should the threads use more CPU time than the PGP's budget per period, the associated cost overrun handler of the PGP will be executed. It is important to note however that this too is an optional feature of the RTSJ.

The general idea of our approach is to create a PGP object for every component in the system. Each (subclassed) PGP is assigned a logical priority, and a computation time per period in which to execute the threads within the component. When the budget is consumed, the cost overrun handler is released which lowers the priority of the entire component's threads to some background priority. At the beginning of the PGP period, the budget is restored and the component's threads have their priorities raised again to their original values. This provides temporal isolation in that the highest priority component's threads execute first, when the component's PGP's budget is exhausted, the component's thread's priorities are lowered to some background level, and the next highest priority component's threads can execute.

The above approach is essentially the same as using execution time servers (such as Deferrable [36], Sporadic [37] or Periodic [38] Servers). Subject to passing a schedulability test, the system guarantees that a component's threads will meet their deadlines under a given server budget per period, and that any overruns will not cause interference with the timing requirements of threads in other components. See [39] for execution-time server extensions to the RTSJ.

A different approach to resource partitioning is time slicing [40]. We chose to provide temporal isolation via execution- time servers because they are bandwidth preserving and are easily implemented at a high level, although they are arguably less deterministic than using time slicing and scheduling windows.

4. PROVIDING TEMPORAL ISOLATION WITH THE RTSJ

To recap, we are providing temporal isolation in the OSGi Framework as a first step to enabling this Framework to be used in the development of component-based RTSJ applications. As this goal is of a very practical nature, we need the designs to be implementable. For this reason, we have chosen to avoid semantic changes to the RTSJ, and used the default fixed priority pre-emptive scheduler as opposed to a hierarchical scheduling [41] scheme. To provide temporal isolation through the use of execution time servers, and if the system is to provide temporal isolation without burdening component developers, some of the RTSJ classes must be extended. These extensions are discussed in the following section. Naturally the OSGi Framework will require extensions/modification too, but this is out of the scope of this paper.

Subclassing classes implementing Schedulable

It is undesirable to rely on the cooperation of component developers to provide temporal isolation. A more suitable approach is to have threads provide temporal isolation on their construction. Wherever possible, we would like to avoid changes to the RTSJ API, therefore we propose subclasses to the classes implementing Schedulable. The RTSJ has the notion of schedulable objects instead of simply threads. This is a useful abstraction when programming since it makes it clear that for example AsyncEventHandlers are also under the control of the scheduler, although they are implemented at a lower level as threads in any case.

We need to subclass RealtimeThread, NoHeapRealtimeThread, and AsyncEventHandler. The subclasses need to:

- 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.
- 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.

It is worth noting that providing temporal isolation should be as invisible to the programmer as possible hence simply extending the Schedulable interface is not adequate. Therefore in this paper we do not extend the Schedulable interface although it is possible that other features required to use the RTSJ and OSGi Framework together will require extensions to the Schedulable interface.

Figure 1 gives the skeleton class definition of an extension to RealtimeThread. The class will require further extensions to include WCET calculation and priority assignment etc in the OSGi Framework; however they are outside the scope of this paper and have therefore not been included in the code.

```
package uk.ac.york.rtosgi;
import ...

public class OSGiRealtimeThread extends RealtimeThread
{
      private Bundle b;

      public OSGiRealtimeThread(BundleContext bc)
      {
            b = bc.getBundle();
            b.addSchedulable(this);
            setProcessingGroupParameters(b.getPGP());
      }
}
```

Figure 1 OSGiRealtimeThread class, adds a reference of itself to a list maintained by the thread's component, and also sets its PGP to that of its component

The class OSGiRealTimeThread 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 such as to register services, install new components, and to subscribe to events. We use the BundleContext object to retrieve the object representing this component. The component (or Bundle) object maintains a list of its Schedulables, and the constructor adds a self reference to this list. This allows the cost overrun handler to iterate through the list lowering the priorities of all of its Schedulables. Finally, the constructor sets this Schedulable's PGP to be the one belonging to its component. This is so that its resource usage is charged to its component.

The AsyncEventHandler, and NoHeapRealtimeThread classes can be extended in the same way in order to provide temporal isolation on object creation.

Subclassing ProcessingGroupParameters

The ProcessingGroupParameters class also requires extensions. Notably, we want to enforce the temporal isolation by:

- Providing a cost overrun handler to lower the priorities of all of the threads 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 2 shows the RTSJ class definition. Again, the class definition is incomplete and will almost certainly be expanded as we work towards developing a Real-Time OSGi Framework.

The OSGiPGP class requires some explanation. It defines two AEHs, costOverrun and replen. The constructor of this class sets the cost overrun handler and other parameters of the superclass. 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 AEHs then use getBundle to manipulate the associated components threads.

```
package uk.ac.york.rtosgi;
import ...

public class OSGiPGP extends ProcessingGroupParameters
{
      private PeriodicTimer replenTimer;
      private Bundle b;
      private AEH costOverrun = new AEH()
      {
            public void handleAsyncEvent()
            {
                  OSGiPGP.this.getBundle().lowerPriority();
            }

      };

      private AEH replen = new AEH()
      {
            public void handleAsyncEvent()
            {
                  OSGiPGP.this.getBundle().higherPriority();
            }

      };

      public OSGiPGP(RelativeTime period,
            RelativeTime cost, RelativeTime deadline)
      {
            super(0,period,cost,deadline,costOverrun,null);
            replenTimer = new PeriodicTimer(0,period, replen);
            replenTimer.start();
      }

      public Bundle getBundle()
      {
            return this.b;
      }

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

Figure 2 OSGIPGP class, provides and sets AEH to manage its associated threads' priorities.

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.

Our 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 our 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.

5. CASE STUDY- NUCLEAR POWER PLANT MONITORING

The purpose of this section is to reiterate the motivation for using the OSGi Framework to develop dynamically reconfigurable real-time systems. Essentially, the case study shows the benefits of dynamism (through sensor reconfiguration and component installation on alarm conditions) and remote controllability (allowing an external supervisor to slow down the reactivity rate and even shutdown the nuclear reactor completely). In addition, this section also shows how temporal isolation is provided amongst components. However, it is important to note that since nuclear power plants are safety critical systems and their monitoring and control systems have hard real-time requirements, we do not see this domain as a realistic target of the OSGi Framework. Two reasons for this are: firstly, service-oriented

platforms such as the OSGi Framework are highly dynamic, it would be difficult to verify safety critical systems developed on such a platform. Secondly, the RTSJ is currently not suitable for use in safety critical systems, although for these types of systems we envisage using the Level 2 subset of the RTSJ Safety Critical Java proposal JSR302 [42]. Despite nuclear power plant monitoring being an unrealistic target for OSGi, we chose this case study because such a system requires powerful features of both the RTSJ and OSGi, and therefore makes for an interesting example of using these technologies together.

Regarding the development status of the case study, it is currently only designed at a high level, and little time has been spent on its design as an OSGi application. The reason for this is because we have yet to complete the design and implementation of a real-time OSGi Framework, i.e. a suitable platform to execute such an application.

5.1 Monitoring System Requirements

The monitoring system has the following application requirements: firstly, the monitoring system must measure the core temperature, pressure, flow rate, and nuclear reactivity rate. These values must then be output to a local operator and an external supervisor. The sensors are periodic threads that access the sensor hardware through the RTSJ's RawMemoryAccess. Given the nature of the system, we will want to avoid the latencies of garbage collection by avoiding the heap with NoHeapRealtimeThread, ImmortalMemory and ScopedMemory.

Secondly, in alarm situations i.e. when the sensors read values outside of a safe range, the operator firstly needs to take some corrective action to make the system safe again, such as by activating the Emergency Core Cooling System (ECCS). The system then needs to provide software for the operator to interact with a decision support system (DSS) and the external supervisor. Once these components have been loaded, the DSS, operator, and external supervisor can work collectively to diagnose the problem and bring it under control. As an example, a sensor detects that the nuclear reactor is operating at a dangerously high temperature. An AsynchronousEvent is fired to signal an alarm condition, an AsyncEventHandler then installs the alarm diagnosing components using the OSGi Framework's life cycle operations. After installing the DSS and external supervisor communications components, the operator can try and find out, for example, whether the temperature increase was due to a loss of external power supply, partial loss of reactor coolant flow, or an accidental start-up of reactor coolant shutdown system. This approach drastically reduces the number of variables that need to be monitored as we only try to find out the exact alarm condition when necessary, as opposed to constantly monitoring for every possible alarm condition directly.

Also as alarm diagnoses components are only installed when necessary, this reduces memory usage which is vital in embedded systems with resource constraints. Figure 3 shows the steps involved in an alarm diagnosis.

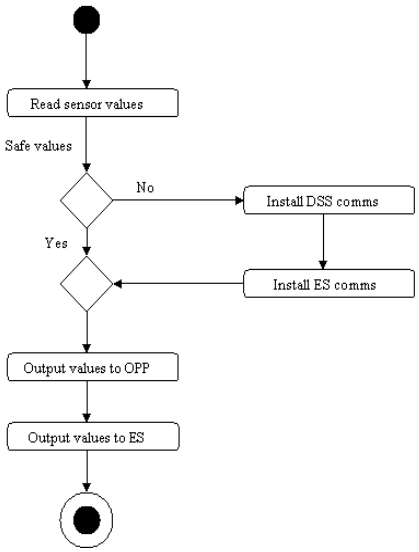

Figure 3 Activity Diagram of the dynamic installation of components under alarm conditions

Thirdly, change sensor configurations. There are three main scenarios for dynamicity regarding sensors. Firstly, it is possible that the RTSJ component interfacing with the sensor hardware contains bugs. The software can be updated at run-time using the OSGi Framework's life cycle operations. Another scenario is to replace faulty sensor hardware. Hardware fails from time to time, we would like to be able to replace the hardware and then be able to replace the software component interfacing with the hardware at run-time. A final scenario is to add new types of sensors as the set of variables to monitor may change. Figure 4 shows the different scenarios for changing sensor configurations.

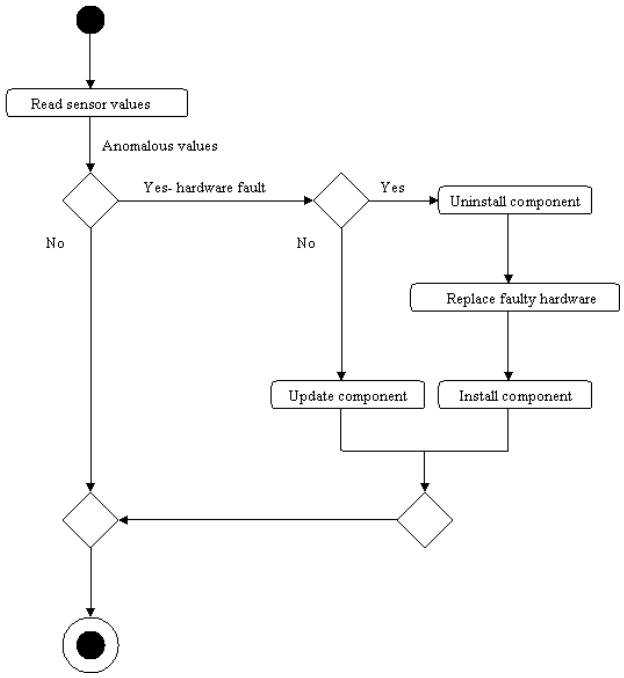

Figure 4 Activity diagram illustrates sensors being dynamically reconfigured for various reasons

The final requirement is to allow the external supervisor to remotely control the nuclear reactivity rate and also to allow the reactor to be shutdown remotely by the external supervisor. There may be emergency situations where the nuclear power plant's operations room is inaccessible e.g. due to fires of terrorist attacks. In such situations, the external supervisor can lower the control rods into the reactor to slow the nuclear reaction down. If absolutely necessary, the external supervisor can also shut the nuclear reactor down completely. Remote control of the OSGi Framework is seen as essential in a real-time version of the Framework. However it will require the use of real-time middleware such as RMI-HRT [43] .

5.2 Monitoring System Architecture and Temporal Isolation

The sensors obtain the values from the actual sensor hardware monitoring the nuclear power plant (NPP). The sensors pass these values to the user interface component for viewing by the operator (OPP). The sensor also passes the values to the Data Vault which acts as a buffer to pass the values onto the gateway which will then communicate the values across a real-time communication line to the external user interface for use by the external supervisor (ES). The monitoring manager is responsible for managing the changing sensor configurations. Finally, in alarm situations, the support comms and DSS comms components are loaded. The support comms component provides some line of communication between the operator and external supervisor so that they can collaborate in controlling the potentially dangerous situation. The DSS comms also helps diagnose the alarm situation by having the decision support system (DSS) contribute in the diagnosis process Figure 5 shows the system architecture.

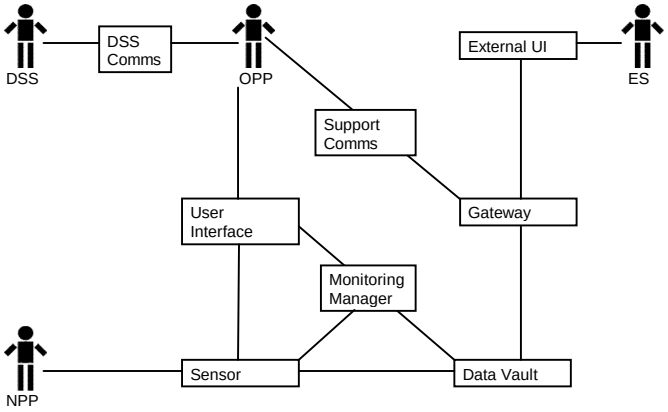

Figure 5 Architecture of the nuclear power plant monitoring system

In terms of temporal isolation, our approach (discussed in Section 3) ensures that independent threads in any of the monitoring system's components do not interfere with one another. To illustrate our approach to providing temporal isolation, consider the following example. A sensor reads an unsafe value and as a result the DSS comms and ES comms components are installed. In alarm conditions, the DSS should be consulted more frequently than the ES, therefore the DSS comms is assigned a higher priority. The DSS comms and ES comms components each have two threads (say) T1, T2, and T3, T4 respectively. All threads are assigned global priorities to reflect their local priority ordering, and to reflect the priority of their component in the system. T1 and T2 are assigned priorities higher than T3 and T4 (because the DSS comms component has a higher priority). As T1 and T2 are the highest priority threads in the system, they execute first. Should they use their component's entire budget, T1 and T2 have their priorities lowered to a background priority, T3 and T4 now have the highest priorities and therefore they will execute. Upon the DSS comms' budget replenishment, T1 and T2 will have their priorities raised to their original values and thus they will become the highest priority threads in the system again. Note that this is bandwidth preserving, should T1vand T2 complete without using their component's entire budget, they will suspend waiting for their next period, and T3 and T4 will execute.

6. 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, coupled with the fact that the OSGi alliance is comprised of many organizations such as IBM, Sun, Nokia, and Mitsubishi, has lead the OSGi Framework to be used in many application domains. The nuclear power plant monitoring system case study showed there is a great motivation for using the OSGi Framework and RTSJ together to develop dynamically reconfigurable real-time systems, for example for remotely controlling the configuration of real-time system such as during mode changes.

Unfortunately, there are number of problems preventing the OSGi Framework and RTSJ from being used together. Some of these problems include: global priority assignment, schedulability analysis, WCET analysis, and lack of temporal isolation etc. In this paper, we discussed two levels of temporal isolation, thread-level, and component-level temporal isolation. Given that the OSGi Framework is used in the development of component-based systems, we are interested at providing temporal isolation at the component-level of abstraction. We have outlined some RTSJ extensions which enable us to provide temporal isolation within the OSGi Framework. Guaranteeing the property of temporal isolation is the first step towards using the OSGi Framework in the development of dynamically reconfigurable real-time systems

7. FUTURE WORK

In this paper we have presented a case for using the OSGi Framework with the RTSJ for the purposes of developing dynamically reconfigurable real-time systems. The focus throughout this paper has been on providing temporal isolation to OSGi applications, however as mentioned in Section 2, temporal isolation is only one of many issues which must be solved before the OSGi Framework and RTSJ can be successfully used together. We intend to work towards a real-time OSGi Framework by carrying out research in the following areas:

7.1 WCET Calculation

Initially, it was thought that the concept of a real-time service would be required. That is a service providing thread would inherit the service requesting threads priority, negotiate real-time constraints, perform its computation subject to those timing constraints, and then return a result to the service requester. However, as the OSGi Framework is an intra JVM service model

and not an inter JVM service model, as with distributed object models such as Java RMI, the service provider does not have an active thread servicing requests, instead, the service requester thread executes the computation for themselves.

Whilst a real-time OSGi does not therefore require the design of real-time services, it does have to closely consider the WCET calculation for threads. A thread's WCET will not only be the time it spends executing its own code, but also the time it executes the code of any services it requires. Essentially, WCET calculation must take service usage into account.

7.2 Server Parameter Selection

In this paper we assumed that the replenishment period and budget of a component's PGP made the component's threads schedulable and at the same time minimised CPU usage so that other components could be accepted into the system and have resources reserved. In reality, generating such parameters is very challenging. There is an optimisation problem in terms of finding values that will make a component's threads schedulable whilst minimising resource usage.

7.3 Local Schedulability Analysis

Given a replenishment period and budget, the analysis checks that those values are sufficient to schedule the threads within a component. In addition to having to do this analysis when a component is installed, it is also necessary when a component is updated. An updated component may have a different thread set, and use services differently (hence threads have different WCET).

7.4 Global Schedulability Analysis

Given some budget and replacement period of a component's PGP which make a component's threads schedulable, can the system actually provide enough resources i.e. is the system able to offer the necessary replenishment period and budget? This analysis needs to take place on a component install. Only if the system can offer the necessary resources does the component get admitted to the system.

7.5 Global Priority Assignment

Threads within each component will have a correct priority ordering. However as components are independently developed, the global ordering will be incorrect. The OSGi Framework (unlike developers) will have a global view and can therefore reassign thread priorities so that the ordering across components is correct.

7.6 Dynamic Reclamation

Once a component is uninstalled from the system, there are resources available which could be distributed among the existing components in the system. This is only worthwhile however for components using "anytime" algorithms and also when the environment is not highly dynamic. If components and installed and uninstalled at a fast rate, the overhead involved in distributing spare capacity may be too large.

7.7 Memory DoS Attacks

Cost enforcement and temporal isolation prevent CPU DoS attacks from occurring, however it is still possible for a component to use excessive amounts of memory, possibly causing OutOfMemoryErrors for threads in other components. In order to prevent this, we need to modify the OSGi Framework for example to allow components to negotiate their memory constraints, and have the OSGi Framework/RTSJ ensure that components do not use more than their memory budget.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

1. Szyperski, C., *Component Software– Beyond Object Oriented Programming*. 1998: Addison-Wesley.
2. Erl, T., *Service-Oriented Architecture: Concepts, Technology, and Design*. 2005: Prentice Hall PTR.
3. OSGi Alliance. *OSGi Service Platform Core Specification, Release 4*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
4. JCP. *JSR 291: Component Support for Java SE*. 2007 [cited 20th November 2007]; Available from: http://jcp.org/en/jsr/detail?id=291
5. JCP. *JSR 8: Open Services Gateway Specification*. 1999 [cited 5th January 2008]; Available from: http://jcp.org/en/jsr/detail?id=8.
6. Li, X. and W. Zhang, *The Design and Implementation of Home Network System Using OSGi Compliant Middleware*, in *IEEE Transactions on Consumer Electronics*. 2004.
7. Hall, R.S. and H. Cervantes, *Challenges in building service-oriented applications for OSGi*, in *IEEE Communications Magazine*. 2004.
8. Gosling, J., et al., *Java Language Specification, Second Edition: The Java Series*. 2000: Addison-Wesley Longman Publishing Co., Inc. 544.
9. Lindholm, T. and F. Yellin, *Java Virtual Machine Specification*. 1999: Addison-Wesley Longman Publishing Co., Inc. 473.
10. 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.
11. OSGi Alliance. *About the OSGi Service Platform*. 2007 [cited 22nd November 2007]; Available from: www.osgi.org.
12. Eclipse Foundation. *Eclipse Project*. 2001 [cited; Available from: http://www.eclipse.org
13. ProSyst. *serve@Home*. 2009 [cited May 2009]; Available from: http://www.prosyst.com/success_stories/SuccessStory_Prosyst_BSH.pdf.
14. IBM. *IBM WebSphere*. 2009 [cited May 2009]; Available from: http://www-01.ibm.com/software/websphere/.
15. BMW. *BMW ConnectedDrive*. 2009 [cited May 2009]; Available from: http://www.bmw.com/com/en/insights/technology/connecteddrive/overview.html.
16. Bollella, G. and J. Gosling, *The Real-Time Specification for Java*. Computer, 2000. **33**(6): p. 47-54.
17. 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.

18. JCP. *JSR 121: Application Isolation API Specification*. 2001 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=121.
19. JCP. *JSR 284: Resource Consumption Management API*. 2005 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=284.
20. Richard-Foy, M., *Partitioning JVM Preliminary Specification*. 2009. p. Personal Communication.
21. 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.
22. 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.
23. 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.
24. Sha, L., R. Rajkumar, and M. Gagliardi. *Evolving Dependable Real-Time Systems*. in *in Proceedings of 1996 IEEE Aerospace Applications Conference, IEEE Inc*. 1996.
25. RUNES Consortium. *Reconfigurable Ubiquitous Networked Embedded Systems (RUNES)*. 2004 [cited 2nd June 2009]; Available from: http://www.ist-runes.org/.
26. 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.
27. 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.
28. Plsek, A., P. Merle, and L. Seinturier, *A Real-Time Java Component Model*, in *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. 2008, IEEE Computer Society.
29. Buttazzo, G.C., *Rate monotonic vs. EDF: judgment day*. Real-Time Syst., 2005. **29**(1): p. 5-26.
30. 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.
31. 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.
32. 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.
33. Siebert, F., *JamaicaVM Cost Monitoring*. 2009. p. Personal communication.
34. Wellings, A., Y. Chang, and T. Richardson. *The Impact of Resource Reservation Contracts on the Real-Time Specification for Java*. in *Submitted to the The 7th International Workshop on Java Technologies for Real-time and Embedded Systems*. 2009. Madrid, Spain.
35. 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.
36. 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.
37. Sprunt, B., L. Sha, and J. Lehoczky, *Aperiodic task scheduling for Hard-Real-Time systems* Journal of Real-Time Systems, 1989.
38. Strosnider, J.K., J.P. Lehoczky, and L. Sha. *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*. in *IEEE Real-Time Systems Symposium*. 1987.
39. 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.
40. 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/.
41. 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.
42. JCP. *JSR 302: Safety Critical JavaTM Technology*. 2006 [cited 1st June 2009]; Available from: http://jcp.org/en/jsr/detail?id=302.
43. Tejera, D., A. Alonso, and M.A.d. Miguel, *RMI-HRT: remote method invocation - hard real time*, in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. 2007, ACM: Vienna, Austria.