A Hybrid real-time component model for reconfigurable embedded systems

Ning Gui, Vincenzo De Florio, Hong Sun, Chris Blondia
University of Antwerp
Dept. of Math and Comp. Science, Performance Analysis of Telecom. Systems group,
Middelheimlaan 1, 2020 Antwerp, Belgium,
and Interdisciplinary institute for BroadBand Technology
Ghent-Ledeberg, Belgium
{ning.gui, vincenzo.deflorio,hong.sun, chris.blondia}@ua.ac.be

ABSTRACT

Increasing capabilities of modern microcontrollers greatly increase their applicability to more and more unstable and complex environments. Dynamic reconfiguration provides a powerful mechanism to adapt in such environments. However, the implementation of dynamic reconfiguration is still challenging for embedded real-time control software systems.

In this paper, we present our real-time component framework which simultaneously supports hard real-time control and non-real-time adaption management while keeping the implementation as lean as possible. Our contribution is the hybrid component model in which one part is designed to support the real-time task while its non-real-time counterpart deals with component adaptation and management functions. A detailed analysis of the intra-component management interface was provided. XML was employed to describe and configure real-time task. We also designed an interface between real-time objects to achieve an inter-real-time task communication scheme based on global shared memory. In the non real-time domain, by mapping much of the management functions to the OSGi system service, we realized the components management service. Our framework can achieve complex component management while providing hard real-time assurance.

1. INTRODUCTION

Due to constraints on footprint and performance, development of mission-critical real-time systems has historically lagged far behind mainstream software development methodologies. Real-time systems are often so tightly coupled to their current configuration and operating environment that they cannot adapt readily to technology innovations, or to changes in run-time situational environments. Moreover, these systems often use relatively static methods when allocating scarce or shared resources to system lifecycle, i.e., well before run-time.

For quite some time already, component-based software engineering (CBSE) [1] [2] has been considered to be a good way to cope with this increasing complexity. By dividing software systems into manageable parts, these components can be developed largely independently and reused many times in different application contexts.

However, most real-time component-based software systems are pre-compiled. When updating or maintaining these systems, they need to be shut down for recompilation [10]. Yet, reconfiguring a system on-line is desirable for embedded real-time systems that require continuous hardware and software upgrades during system operation [1][3]. For example, real-time sensor-based control systems must be designed and developed such that software resources, e.g., controllers and device drivers, may change on the fly. Hence, a reconfiguration mechanism, enabling adding, removing, and exchanging components on-line, is needed to ensure that the software is updated without interrupting the execution of the system.

Moreover, in complex real-time systems, the challenges of implementing real-time behavior include not only decoupling and modularizing the real-time behavior, but also the ability to deal with the component deployment, life cycle management, version control, component constrains resolution, etc. The low level languages currently used to program real-time systems are not up to the task of developing large, distributed systems that span a range of complexity from board-level to large enterprise systems. Implementing a system including all these adaptation and management aspects in low level programming and real-time model is very complex and error-prone.

In this paper we present an adaptive framework for dynamic reconfigurable real-time systems. The software framework was developed as part of the ARFLEX Project [4]. It is an offshoot of a project to develop adaptive robots for flexible manufacturing environments. The following goals for a robotics programming environments were initially set forth in ARFLEX:

- Providing hard real-time support.
- Ability to support reconfigurable robots.
- Dynamically changing control logic and Application logic
- Support for different, multiple sensors.
- Ability to support run time component reconfiguration.
- Avoidance of downtimes during system evolution.
- High automation level and quick re-programming capability.

To solve these problems while keeping the system as lean as possible, we suggested a domain-specific software framework

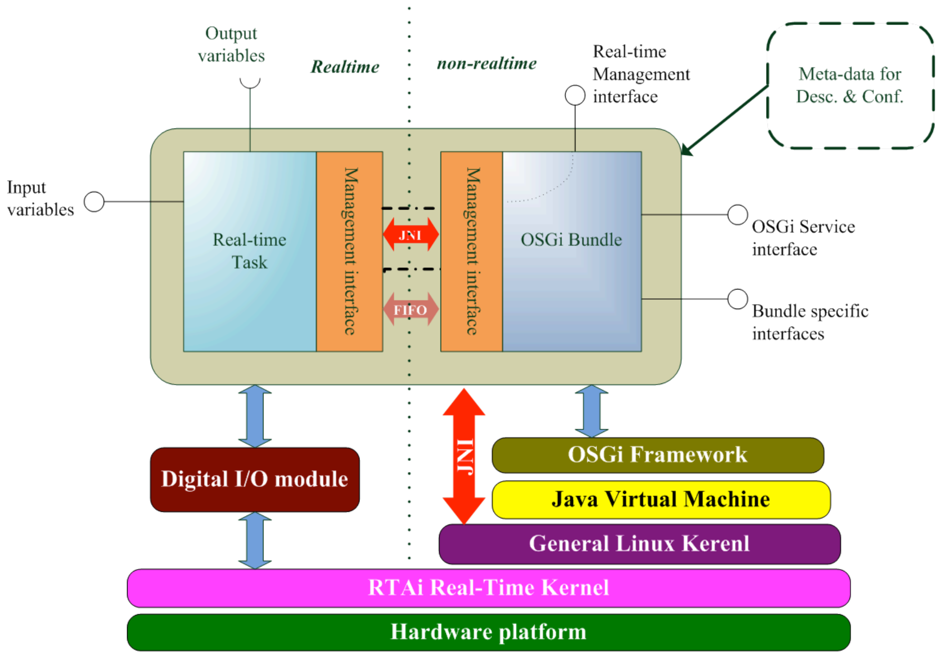

Figure 1: a split container architecture running the Hybrid real-time component

based on OSGi [14] and RTAI [5]. In this framework, a Hybrid Realtime Component (HRC) model was designed. It has two main aspects. One aspect is focused on the support for the real-time requirement. RTAI is used for real-time OS support. Based on the so-called dual kernel technique, RTAI can provide hard real-time support for extended Linux kernels. At the same time, the real-time components are wrapped as OSGi bundles so that we can take full advantage of dynamic configurability, one of the most graceful features of OSGi.

By mapping much management service to the OSGi system service, we provide the HRC's life cycle, version, component constraints management which lacks in most real-time system designs. Reusing current mature specification makes our framework rather simple in design and easy for implementation.

Thanks to the dual kernel design of RTAI, non-real-time tasks such as java tasks cannot interfere with the real-time tasks. Much of the uncertainty introduced by JVM garbage collection and Linux system scheduling scheme can be avoid. Meanwhile, a set of management interfaces were designed to manage the real-time task properties. The XML meta-data are employed to describe and configure the real-time tasks. We also designed the coding methods and provided templates to map the component's local properties, inputs and outputs into the global real-time RTAI shared memory which allows effective communication among the real-time tasks.

The remainder of this paper is structured as follows: Section 2 presents the structure of our hybrid component and describes a split framework supporting the hybrid component. In the same section we also introduce the XML scheme that describes and configures the real-time task. Then, in Section 3, we discuss the inter-real-time communication. Section 4 focuses on the intra-component interface design and briefly reviews the implementation methods. Section 5 then discusses the component runtime adaptation. Advanced techniques of container-managed quality assurance, which support quality adaptation at runtime, are presented in Section 6. Finally, in Section 7, we summarize our achievements and describe our future work in this framework.

2. The Hybrid Adaptation Framework

In the development of project ARFLEX (Adaptive Robots for FLEXible manufacturing systems, European project IST-NMP2 016880), we have designed a hybrid adaptation framework for dynamic reconfiguration of real-time component. ACCORD [6] prescribes that real-time systems should first be decomposed into a set of components followed by decomposition into a set of aspects. Reconfigurability is supported in their real-time component model by weaving the new interface via offline weaving. We share their view of separation of concerns but use a different system approach.

2.1. The Hybrid Real-time Component

In control systems, each component can be mathematically modeled using a transfer function, which computes an output response for any given input [15]. However, with the growing complexity of real-time systems, the real-time component is characterized by more and more non real-time related requirements. These aspects include the deployment process of components, the management of component specifications and implementations, as well as the configuration and reconfiguration phase of components. A reconfigurable real-time component should satisfy two different types of requirements. In this framework, we propose the Hybrid Realtime component model which consists of two main parts – one part runs directly in the real-time OS layer and the other part runs in the OSGi middleware environment.

The real-time part of each HRC is an independent concurrent process, whose functionality is defined by the methods of a standard object. The standard object implements a set of functions that enable it to receive data and commands from the non real time parts while sending status information to its non real-time partner. Each real-time task has certain general properties such as name, description, task type etc, and task specific properties, which are used to (re)configure the real-time component for use with specific hardware or applications. Communication with other real-time modules is restricted to its input and output ports. The real-time task can also connect to sensors or actuators, via the digital I/O module. The details of accessing the hardware are encapsulated within the real-time task.

In the non real-time part, we designed a reconfiguration specific interface containing methods for creating, deleting tasks, setting component parameters or changing the component state and frequency. The non real-time part is implemented as an OSGi service bundle and we use the OSGi framework service to realize the modules deployment, version control, life cycle management etc. Detailed discussion about this can be found in Section 5.

In our implementation, an XML file is used to specify the meta-data describing the real-time task. The example configuration file task.xml is shown in Figure 2. The same real time code may have possibly multiple versions of task.xml file. For instance, we can have multiple types of tasks for different environments by using different configurations of the same functional logic. Currently, we have designed the template file which implements predefined interfaces to set up the reconfigurable real-time task. The component developer merely will have to concentrate on the functional parts of the real-time task, which greatly reduces the complexity of developing such real-time system. A visual UI tool is also under development to generate interface and other configuration details for HRC code. In the real-time domain, the communication between real-time objects is restricted to their input and output ports. A global & local shared memory scheme has been designed to help different tasks exchange information, as detailed in Section 3.

2.2. Split architecture of HRC

In complex and distributed real-time systems, real-time capabilities are often not necessary for the whole large applications, but only for small parts of them. Consequently, we applied this philosophy to the component design and implementation. The result is a split architecture where we have a large non-real-time container, which is based on OSGi. We also have a real-time operating system. Currently, we use RTAI [5] for the underlying hard real-time OS support.

By doing so, we both simplify the development by reusing existing software and minimize the amount of real-time program code. By defining a standard communication interface we connect the two parts and make them appear as one component to the application. This framework provides the mechanisms for quickly realizing each of these modules by using the hybrid model to implement them as real-time reconfigurable OSGi bundles.

The non real-time container implements all functions that do not need to provide real-time guarantees. This includes the deployment process of components, the management of component specifications and implementations, as well as the initialization and startup phase of applications. In the implementation, we design and provide a set of general OSGi service interfaces. These service interfaces can be employed by the bundle itself or can be registered as OSGi services. External bundles can use the internal service oriented architecture to dynamically query all available managed services and change the HRC's properties to suit run-time environment changes.

The HRC's life cycle is controlled by the OSGi framework. The OSGi container also controls the real-time object by invoking the methods in the IRealTimeManagement interface. Currently, a general bundle implementation was designed and it can be used as a general frame to implement HRC OSGi parts.

Real-time container: In the real-time environments, currently, we designed the RTAI real-time task template. This template implements the standard interface and standard functions such as import variables and export results to out port by reading/ writing the global memory. It also provides a general task state control implementation, which executes certain code fragments. For example, in task init stage, the task will wait for the init parameters to be sent from the non real-time part to the task being initialized.

2.3. Meta-data configuration and description

The XML has become a standard method to describe data and the OSGi platform has very good support for it. OSGi provides the XML Parser service for its bundles. Each real-time bundle's real-time interface and constrains are defined in XML. As a consequence, this approach enables a user to search for components suitable to a particular real-time application. Furthermore it enables a component to be configured when instantiated, and facilitates the sharing of components between repositories. We also use the XML to describe the real-time tasks' component-specific input & output variables and properties, which will be used to exchange data between real-time tasks and configure the real-time tasks.

By providing a detailed XML description for a real-time task's binary codes, the HRC user should then be able to parse the XML description, and from its contents, determine the specifics on how to properly interact with the real-time task with appropriate communication methods and data formats. Meta-data should include the task's unique name, description, properties (name, type, default value), ports (inports, outports) including port name (unique), port type and port size. Figure 3 shows a fragment of meta-data file which describes a smart camera that can return regions of interests (subsets from a frame image data) on demand.

```
   <rttask-desc>
        <preface>
            <name> SCam01 </name>
            <desc> It's a type of high speed smart camera that can
selectively transfer only part of the image in which the reference points
reside.    </desc>
           <tasktype>periodic</tasktype>
           </preface>
       <periodic_task >
            <frequence>100</frequence>
            <runoncpu>0</runoncpu>
            <priority>2</priority>
       </periodic_task>
   <properties>
      <property  name="prfr01"  type="int"  desc="frame  rate"
value="100"/>
       <property  name="prex02"  type="int"  desc="exposure  time"
value="20"/>
      </properties>
```

```
  <ports>
    <inport name="ipx001" type="integer" size="1" desc="top x position of frame window"/>
    <inport name=ipy001" type="integer"  size="1" desc="left y position of frame window"/>
    <outport name="ipimg01" type="byte"  size="400" desc="Frame video data in window"/>
    </ports>
</rttask-desc>
```

Figure 2. Sample configuration for smart camera task

In the configuration file task.xml, the name part of task is a global unique name which is used as reference value for the task. In this example, the real-time task will be created with global unique name "SCam01". The properties element contains the parameters that need to be set during the task initialization and task reconfiguration phases. Each property has the unique name that will be used as reference by RTAI. The value field is used when the component is initialized. The inport & outport are the external references for the inport & outport values. They have an important property – size, which is used to allocate memory in the global shared memory. The inputs/outputs and the properties' data are stored in the memory as raw data. The names of ports are used to perform the bindings between input and output.

In our implementation, we use the raw data format. The type field of properties and ports is to be used in later possible data conversions. It can also be employed for validating the compatibility between two tasks. For example, a system configuration could be considered as valid only if for every RTC installed, any data that it requires at its input ports is produced by one of the other HRC as output. The system cannot have two HRC with same output, which may be a conflict as of which output should be used as a given name.

In the current implementation, for simplicity, we directly use many RTAI schemes in the XML file. For example, we prescribe the name field should be 6 ASCII characters. In RTAI, tasks are referenced by using an unsigned long value. It provides function unsigned long nam2num (const char *name) that converts the 6 ASCII character name into unsigned long type. Similarly, the ports names and property names are also used to identify the object in the RTAI system. The ports values and properties have been defined as state variables stored in the global shared memory.

3. INTER REATIME COMMUNICATION

In realizing our HRC, how to map the intra component real-time communication into the RTOS Inter process communication is a very difficult aspect of creating dynamically reconfigurable real-time systems. In order to satisfy the real-time requirements, the communication mechanism needs to comply with the requirements described in what follows.

2.1. Real-time communication requirements

The inter real-time tasks' communication scheme should support the independent process component model [1]. The process reads the input ports at the beginning of each cycle to obtain the most recent data available and writes to the output ports at the end of each cycle. It also needs a simple and straightforward way for binding between components. The communication links should be able to be dynamically reconfigured in bounded time. At the same time, an output should be able be sent to multiple inputs. The communication link should also support processes that may be executing at different frequencies.

As in [1], the global shared memory scheme is used because it has the one main advantage compared to the message based approach: Fanning an output to multiple inputs is difficult in message based scheme because it requires a message to be duplicated for each input. Thus, a more complex mechanism is needed to ensure atomic multicast. Furthermore the overhead of sending multiple messages is comparably higher than writing into a single global shared memory cell.

2.2. Variable-based Communication

As in [1], the communication between real-time tasks is performed via state variables stored in global and local tables. Every input & output port and component properties is represented as a state variable stored in the global table. The table is stored in the shared memory. RTAI provides a set of real-time memory management services which can be shared by RTAI kernel modules and Linux processes.

The variable-based communication has been designed and implemented in our framework. This includes the update of local and global tables. The state variables corresponding to input ports are updated prior to executing each cycle of a periodic function, or before the main event processing function for aperiodic tasks. After the processing for that cycle or event, the task may update the state variable in the global table. The transfer between the local and global tables are block transfers managed by standard C function memcpy(). The integrity of the data is ensured by the data transfers being performed as critical sections. We use one single lock for the entire global variable operation. This approach seems restrictive but even if multiple tasks have separate locks, only one of them can physically access the shared memory at once. Hence this approach is more simple and effective.

4. INTRA-OBJECT COMMUNICATION

Due to the hybrid structure of our HRC, the intra communication interface needs to be standardized to simplify the development of the component. We designed one set of interfaces that need to be implemented by the OSGi parts. An important note about the system is that our framework is designed independently of the functional part of each real-time task. Most of the implementation of these interfaces is identical. We designed classes that implement such interface. The programmer can simply reuse them.

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

Figure 3. The interface definition in OSGi

Firstly, we may need to use deploy() method to load the real-time task code. System.loadlibrary is invoked to install the native code library. After real-time code has been loaded, the task default settings are acquired by parsing the XML file bundled in the OSGi bundles. This information includes: task type, FPU usage, priority and CPU assignment etc. After getting this information, the task is initialized. The initialization process includes creating a task's context, dynamically allocating its memory, creating a local table and translating I/O port symbols into pointers to the global table, and calling the user-defined init method.

When the init procedure is finished, the process then waits in the standby state for a dynamic reconfiguration until the run signal is received, then the local table is updated to reflect the current state of the system, and execution begins. The startRTtask() method was designed to send such signal.

The suspendRTtask and resumeRTtask are used to control the task state. This is very useful when the system needs to suspend/resume certain tasks for a while. It is especially useful when the system experiences errors or when system evolution is needed. When the task is finished or system needs to stop the task, the stopRTtask() can be used to send a signal to the task to perform task finalization function and terminate the task. The get/set properties methods help the programmer to dynamically change the tasks' behavior. In the interface definition, there are no task specific parameters. All related parameters are acquired after parsing the XML configuration file. For example, startRTtask method will get the task name from XML first, and then get task reference by this name; finally it will send a message to this task to signal the task to enter the Run state.

In the OSGi specification, part of the bundle's lifecycle is managed by the framework. In our hybrid approach, the real-time task's life cycle is mapped onto the corresponding non real-time peer. When the OSGi system starts the HRC bundle, it will invoke the deploy() method. When system needs to stop the bundle, it will invoke the stoptask() method to stop and delete real-time task in the bundle stop stage. By doing so, the real-time task's life cycle is synchronized with its non real-time counterpart whose life cycle is controlled by OSGi system.

Each HRC can expose the management interface to external applications as a IRealTimeManagement service provider. General or application specific adaptation managers can monitor the tasks status and adjust the parameter or even change the application structure according to current available resources and system requirements.

Here, we have to point out that currently our framework focuses on providing a general adaptation framework for real-time systems rather than providing the guaranteed real-time reconfiguration. As our adaptation logic mainly operates in the non real-time domain, it cannot guarantee to perform certain adaptation actions in prescribed time constraint. However, guaranteed real-time reconfiguration is often data-dependent, mode-dependent, configuration-dependent and hardware-dependent. It is hard to provide a general scheme. In our implementation, we minimize the uncertainty brought by non real-time JVM by realizing the task management interface by creating native RTAI tasks to perform such actions. For example, we realize the getProperty method by creating a new RTAI task, performing global memory change and sending message via IPC functions to signal the property change. Due to the openness of framework, the programmer can use these services or design his/her own particular interface for application & mission specific real-time adaptation logic, which is out of the scope of our framework.

5. ADAPTATION TO CHANGES

Environment conditions or resource availability may change during application runtime. To deal with such situations, our framework enables adaptation of an application to changing resources and other mutating environment conditions. Adaptation and reservation work together in our architecture, which allows for adaptation on two different levels:

1. Adapting by adjusting parameters
2. Adapting by adjusting parts of an application's structure.

Together with the previous item, this type of adaptation can be achieved by using an application-specific adaptation manager.

A key concept in our approach is that we want to separate the actual adaptation logic as much as possible from the business logic constituting the components' code. In our implementation, we make full use of OSGi bundle management scheme, which greatly reduces the system complexity and development time. We can directly integrate many standard component interfaces that are available for common functions such as HTTP servers, configuration, logging, security, user administration, XML, etc into the target system.

5.1. Property-based adaptation

Our platform targets adaptive robots in flexible manufacturing environments. So the system requirement and control target or control logic may change greatly. For example, our smart camera component returns a window of the grabbed frame data. The window's size and exposure time may change according to new system requirements. This kind of adaptation is what we call parameter-based adaptation, as parameters of the involved components are adapted at runtime. In order to be adaptable to new resource situations each component must implement an adaptation interface. The properties and the values are described in the XML meta-data file. The client component & application use the adaptation interface & meta-data file to correctly communicate with specific real-time task.

Each application can provide the adaptation manager with perceived changes in the amount of available relevant resources and notifies components to be adapted. Our framework provides standard real-time management interface for each real-time bundle so each application can easily adapt the real-time task attribute according to its own application specific requirement. In the future, the OSGi configuration service will be integrated into our system to ease the system configuration process.

5.2. Adaptation by structural modification

In our framework, we use the OSGi system service to simplify the structure modification process. In OSGi, service bundles' basic constraints is managed by the Package Admin Service. In our implementation, we mapped the library constraints of RTAI modules which were implemented as Linux modules into constraints of the OSGi bundles'. As a simple example, the RTAI_hal.ko should be loaded before any other RTAI tasks. Constraints such as this can be expressed and solved in OSGi as follows. We designed a bundle which exports the org.RTAI.hal package. This bundle just takes responsibility to install

RTAI_hal.ko modules. Any our HRC bundle will try import this package when it needs RTAI service. Of course, in the real application, the modules constraints is rather complex. OSGi also provides version, resolution (mandatory, optional) scheme to alleviate this problem. For example, in bundle A's meta-data manifest.mf file, it states:

Bundle A: Import-Package: p;
resolution=optional;
version=1.6

The example shows Bundle A need package p with version equal 1.6 and the resolution is optional which means the bundles A may resolve in starting without the package being wired. While, when resolution is mandatory, it means the package p must be wired for the bundle to resolve. All these resolving works will be done by OSGi. Details can be found in [14].

In OSGi framework, service bundles can be installed, started, stopped and uninstalled without need to reset the system. However, due to our hybrid component structure, we need also consider the real-time task management. In the current approach, we deem that if two components can provide the task description parts in xml file which includes the task name, properties, and ports (inputs & outputs) description part in the XML, then the new component can take the place of the old component without restarting the whole application or system.

In addition to manually start or stop specific bundles, the OSGi specification supports the Start Level Service [14]. Each bundle has an assigned start level and the Framework also has the active system start level. The start level can be dynamically changed during the runtime. By assigning the active start level to different types of bundle or changing the current system start level, we can easily change the application structure. For instance, when an error occurs in the system, we can change the system start level to the safe mode in which only the minimal fully trusted bundles are started.

However, determining when it is safe to perform a dynamic reconfiguration is beyond the scope of the framework. Developing policies that ensure stable execution during a reconfiguration is usually application specific. In our experiments, we used a relatively conservative approach of ensuring that the robot is temporarily at safe state (i.e., velocity and acceleration are both zero before dynamic reconfiguration begins). Further research is required in order to develop more aggressive policies.

6. RELATED WORK

Li M. et al. [7] illustrate the software architecture of a problem solving environment used for the construction of scientific applications from software components. Each legacy component is encapsulated as a CORBA object, with its interface and constraints defined in XML. Their work lacks the consideration of real-time specific requirements.

Eide E. etc [8] illustrate the design and implementation of CPU broker by adjust the tasks CPU reservation. Their research focuses on improving system QoS level and did not consider the real-time component management.

In the area of real-time middleware, there has been significant work in both commercial standards and novel research. For instance, the Object Management Group has designed and continues to evolve standards for RT CORBA [2]. CORBA provides a set of middleware services, which have to be used explicitly by application programmers. However, our platform follows the approach of implicit or descriptive middleware; that is, the use of such services is not directly implemented within components' application code but provided implicitly by the framework support according to additional component descriptors.

CIAO [9;10] builds a QoS-enabled CCM implementation on top of TAO[11]. The project adhered to existing OMG specifications such as RT/CORBA and CORBA Component Model (CCM) [12] and the extension of those. In contrast, our focus is on the challenges of simultaneously supporting hard real-time control and non real-time component adaption management while keeping the implementation of our component model as lean as possible. The considerable overhead of implementing or extending a fully compliant CCM infrastructure would have been counter-productive to our system goal.

Stewart D.B. et al. [1] designed a reconfigurable port-based object framework which has similar real-time communication scheme as ours. But their design used a pure real-time design philosophy which makes this system hard to develop upon.

Hong W. E. et al. [13] used a similar combined approach of OSGi and RT-Linux, but their approach focus on the combined usage of the real-time and non real-time tasks by using the JNI and FIFO. Unlike their approach, our system focuses on the design of management interface and declarative description of real-time tasks. We also reuse the OSGi scheme which solved the version control and software constraints, whose schemes were lacking in most of previous real-time component related research work.

7. CONCLUSIONS

In this paper, we have presented our approach to guaranteeing real-time properties while supporting intricate component non-real-time requirements such as management, reconfiguration, version control etc.

A key concept in our approach is that we want to separate the actual adaptation logic as much as possible from the business logic constituting the components' code while keeping system implementation as lean as possible. In our framework, real-time task's functionality and requirement are described by XML. The non real-time part uses this information to initialize and configure the real-time tasks. A set of communication interfaces has been defined which enables the non real-time bundle control the real-time task. We also introduced inter real-time task communication based on global shared memory. By mapping much management service to the OSGi system service, we achieved the HRC deployment, life cycle, version, and component constraints management.

Our framework and implementation try to reuse the state of art middleware approach while still satisfying the real-time systems' requirements. By mapping the component real-time part directly to the RTAI task, we achieved high real-time performance. While, by employing the OSGi specification, we can enjoy the flexibility of it. We can also enjoy many standard component interfaces that are available for common functions like HTTP servers, configuration, logging, security, user administration, XML, and many more.

At the same time, our design tries to decouple and modularize the real-time behavior. It only deals with functions which have specific real-time requirements. The standard interface for inter/intra component communication designed in our framework makes our system very portable. Our framework can be easily migrated to other real-time OS such as RTLinux or TimeSys Linux while OSGi which is based on JAVA can naturally support various platforms.

8. References

[1] D. B. Stewart, R. A. Volpe, and P. K. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," IEEE Transactions on Software Engineering, vol. 23, no. 12, pp. 759-776, Dec.1997.

[2] K. Sandstrom, J. Fredriksson, and M. Akerholm, "Introducing a component technology for safety critical embedded real-time systems," Component-Based Software Engineering, vol. 3054, pp. 194-208, 2004.

[3] A. Cerpa and D. Estrin, "ASCENT: Adaptive self-configuring sEnsor networks topologies," IEEE Transactions on Mobile Computing, vol. 3, no. 3, pp. 272-285, July2004.

[4] ARFLEX Project, www.arflexproject.eu, 2007

[5] “RTAI Programming Guide,” 2006.

[6] Aleksandra Tesanovic, Dag Nystrom, Jergen Hansson, and Christer Norstrom, "Aspects and Components in Real-Time System Development: Towards Reconfigurable and Reusable Software," Journal of Embedded Computing, pp. 17-37, Jan.2005.

[7] "CORBA Component Model v.4.0," OMG document. formal/04-01-06, 2007.

[8] Maozhen Li, Omer F.Rana, and David W.Walker, "An XML-based component model for wrapping legacy codes as Java/CORBA components," 2000.

[9] Eric Eide, Tim Stack, John Regehr, and Jay Lepreau, "Dynamic CPU Management for Real-Time, Middleware-Based Systems," in Proceedings of the Tenth IEEE Real-Time and Embedded Technology and Applications Symposium(RTAS 2004) Toronto: 2004.

[10] N. B. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring real-time aspects in component middleware," On the Move to Meaningful Internet Systems 2004: Coopls, Doa, and Odbase, Pt 2, Proceedings, vol. 3291, pp. 1520-1537, 2004.

[11] A. Gokhale, D. C. Schmidt, B. Natarajan, and N. B. Wang, "Applying model-integrated computing to component middleware and enterprise applications," Communications of the ACM, vol. 45, no. 10, pp. 65-70, Oct.2002.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," Computer Communications, vol. 21, no. 4, pp. 294-324, Apr.1998.

[13] Won Eui Hong, Bon Jin Ku, Myung Jin Lee, Hong Bae Park, and Soon Ju Kang, "Combined Approach of OSGi and RTLinux Framework for Supporting Software Architecture of Internet Embedded Real-Time System," Toronto, Canada: 2004.

[14] OSGi Service Platform Core Specification v4.0, Aug. 2005 http://www.OSGi.org, 2005

[15] R.C. Dorf, Modern Control Systems, third edition. London: Addison-Wesley, 1980.