An architecture-based framework for managing adaptive real-time applications

An Architecture-based Framework for Managing Adaptive Real-time Applications

Ning Gui, Vincenzo De Florio, Hong Sun, Chris Blondia
PATS group, University of Antwerp, Belgium,
and IBBT, Ghent-Ledeberg, Belgium
{ning.gui, vincenzo.deflorio, hong.sun, chris.blondia}@ua.ac.be

Abstract—**Real-time systems are increasingly used in dynamic changing environments with variable user needs, hosting real-time applications ranging in number and nature. This paper proposes an architecture-based framework for managing components' dependence and lifecycle in an effective and uniform way. A real-time component runtime service is proposed here to maintain the global view, control the whole lifecycle of the components and keep existing real-time components' promised contracts in the face of run-time changes. This framework is designed to be easily extended with other constraint resolving policies as well as dependence descriptions languages. At end of this paper, the framework is tested by a simulated control application via adaptation and performance aspects.**

Keywords-component; adaptive software; run-time adaptation, architecture-based adapatation

I. INTRODUCTION

Traditional real-time systems such as process control systems typically work in closed and highly predictable environments. As long as those assumptions hold, this approach allows real-time requirements to be met with very high assurance. However, some soft real-time systems, such as smart devices, However, some soft real-time systems, such as smart devices, gains more and more popularity and increasingly operate in changing environments and with diverse user needs. In coping with dynamic and diverse requirements, such devices normally are equipped with open software system that able to host applications ranging in number and nature over time. In such system, the traditional waterflow approach for real-time application development: design, verify, map, and deploy could not work effectively as the system configuration will evolve during the whole system lifecycle.

This open and evolving execution environment requires new approaches in building and manages real-time applications. Thorough off-line system analysis is forbidden by such dynamicity. However these installed applications are, to a certain extent, inter-dependent with one another, as one application may competing with other applications for the CPU, memory, networks or other system resources. Failed to deal with such dependence may lead to breaking applications' real-time contracts which the system designer has supported to satisfy.

Traditional RTOS and real-time schedulers based approach could not effectively cope with such complexity in dependence, as the scheduler itself has no way to get and could not understand that high level application dependences. This dependence can only be interpreted according to current context knowledge which involves many facts in the whole system – hardware, software and human. The RTOS is in a sense too "far away" from the application-level concerns to be able to take the right decisions on which components to select and which resources to dispatch.

In order to support the dynamic changing set of real-time applications, from our experience in designing adaptive real-time systems, any effective solution should have the following three important aspects. First, those resource requirements across the currently hosted applications must be obtained and managed during run-time. Second, those component dependences, including the explicit functional dependencies and the implicit dependencies due to competing for the limited available resources, need to be explicitly modeled into architectural model across application domains. Third, the architectural model should be used to drive the lower-level managements of the corresponding real-time tasks.

Our framework tackles these problems from different perspectives. In this paper, we introduce an architecture-based management framework - the Declarative Real-time Component (DRCom) Framework, spanning through installed application domains. By eliciting each real-time component' distinguishes attributes from its meta-data and its component management interface, an accurate global architecture model for current real-time system can be built, then this model is mapped to adjust low-level real-time task's attributes and states according to plug-able adaptation policies. Benefited from the Service oriented architecture, this framework can be easily extended with other dependence resolving policies to deal with diverse system requirement. The effectiveness of our architecture is demonstrated both from qualitative and a quantitative point of view.

The rest of this paper is structured as follows: in Section 2, a scenario shows the importance for inter real-time application dependence managements. Following section 3 gives a general overview of our architecture. Section 4 describes the component's declarative real-time meta-data format design. Then in section 5, the component's dependence resolving process is discussed. Then, our framework is evaluated from two aspects in Section 6. Section 7 provides a comparison with related works. The future work and our conclusions are introduced at the end of this paper.

II. SCENARIO DESCRIPTIONS

To better illustrate all the complexities in introducing the context knowledge into the application composition process, we introduce an example scenario that will be revisited several times through the course of this paper.

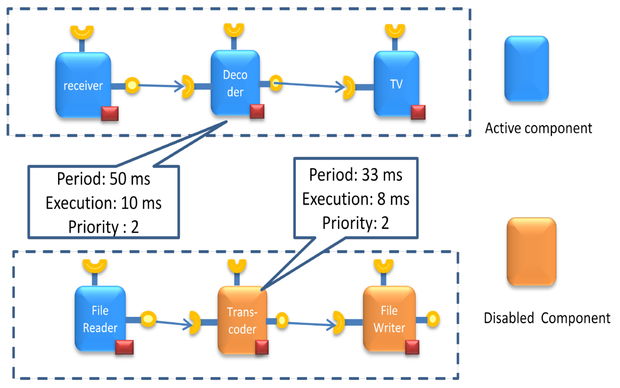

Figure 1. Adaptation across application domains

As a typical multi-task system, if a user starts those two applications, a Set-top may try to execute the two applications simultaneously no matter the Set-top device has enough resources. If not, it may eventually lead to possibly transient timing problems including missing frame, data overflows etc. These time breaches can results in the poor video quality and bad user experience. The typical schedulers in the RTOS itself has no way to determine which actions to be taken as it only support very primitive attributes for scheduling, e.g. Priority or Deadline. Such conflict can only be solved with context knowledge.

III. ARCHITECTURAL FRAMEWORK

The above case study shows that, without the architectural-based adaptation supports, it is very hard to effectively deal with higher level conflicts across application domains. In order to deal with these challenges, a framework based on declarative realtime component is proposed to build the adaptive real-time application. Firstly, the system design goal is analyzed.

A. System design Goal

To deal with highly dynamic environments while providing explicit real-time support, we describe an integrated architecture that provides support for the following.

1) dynamicity support: The dynamicity of real-time component is supported by taking control of all requests for creation, configuration, reconfiguration and un-initialization – that is, the whole of the components' lifecycle – from underlining system. This allows having a complete picture of all the lifecycle requests taking place in the system as a whole, as well as a snapshot of current system configuration.

2) dependence support: Another problem we have to tackle is the intrinsic dependencies that each component has on other components – both in hardware and software. We distinguish two types of dependences, functional (structural) dependencies regarding the business logic and non-functional dependences regarding the QoS requirements.

3) customizable global adaptation support : Given a description of the current system configuration and of the functional and non-functional dependencies between components, rather than using a general adaptation scheme, here we make use these descriptions to support the definition of custom adaptation strategies. Depending upon the dynamic availability of resources and adaptation policy (for instance, performance first, safety first or mission duration first …) different algorithms for constraint resolution may be desired. The following section describes the system architecture design of our solution.

B. System Requirements Analysis

In our model, the distinguishing real-time aspect of DRCom is declared in a meta-data file. Intensive studies have been carried out on how non-functional properties of a component implementation are specified, ranging from simple name-value pair based SPDF (Simple prerequisite Description Format) [17] to modern QoS specification languages such as CQML+ [15] etc. According to specific domains, different ways of specifying real-time components' prerequisites and dependence may be used.

The same is true for the adaptation logic. Due to the fact that real-time application adaptation logics are domain-specific and/or application specific, rather than try to propose a silver bullet general adaptation logic, which is rather unlikely, instead, this architecture is designed to be flexible and easy to be tailored to different contexts.

Our framework is designed in a ways that different kinds of description languages and adaptation logics could be used. Three key modules were designed to fulfill different aspects requirements – the **Description**, **Reasoning** and **Action** modules. Dependencies and configurations are represented in the description modules and reasoned upon in the reasoning module while the corresponding actions are taken in the Action module. Contrary to [15] and most of the state-of-art studies, in which the execution runtime statically integrates these three modules, our framework implemented them as service providers by employing the Service-oriented model. They are loosely coupled by their service interfaces. Future more sophisticated QoS descriptions as well as those adaptation algorithms to be easily plugged into our framework as long as they implemented the same service interface.

C. Architectural Framework

Figure 2 presents a schematic view of the major elements of our architecture. This framework is composed of three main parts corresponding to the modules described in the previous section plus other ancillary services, such as resource management for monitoring system resource changes.

1. A mechanism (Custom language parsing Service, working as Description module) for resource dependence representation lets developers specify component dependencies in their own custom language.
2. A constraint resolving service (working as reasoning module) for QoS constraints resolving policy lets users or administrators specify the domain-specific system adaptation policy.
3. The Declarative Real-time Component Run-time (DRCR) execution environment (action module) works at the core of the framework. It manages the DRCom instances and performs the actions reasoned by resolving service. It also takes the responsibilities to monitoring the change events sent by Resource Management Service and OSGi system. A DRCom instance registry is designed to

keep the installed real-time component instances' information and it is maintained by DRCR.

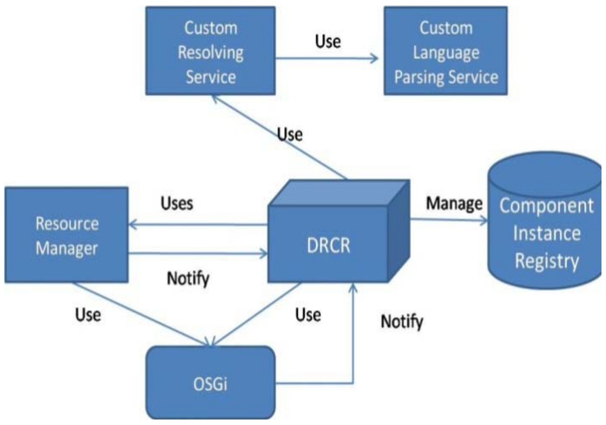

Figure 2. Architectural Framework

Based on the decisions made by the constraint resolving service, DRCR takes full control of the component lifecycle. In this framework, a complete component's lifecycle model and a correspondent real-time component management interface is designed. The adaptation process will be triggered by component state changes and the notification from external event sources such as Resource Manager or a simple periodic time. In the following sections, we provide a more in-depth description of key elements and processes of this architecture.

IV. COMPONENT DESCRIPTION

Real Time Component description is defined in meta-data files attached with component implementation. This document specifies the distinguished characteristics of the specific real-time component. Here we use the hybrid real-time component model introduced in our previous work [9], while an important feature is added to support more complex description languages.

A. *Meta-data description design*

In our implementation of DRCom, a real-time component configuration represents a component instance's functional contracts as well as non-functional requirements. As discussed in section 3, the framework should support different kinds of component prerequisite languages. However, from an implementation point of view, a purely separated approach means little or no pre-knowledge of the component's meta-data format. This will introduce considerable implementation complexity and bring unnecessary performance overhead. Compared to the complex non-functional requirements, component's functional part is comparably well structured and stable. In order to strike a balance between performance and complexity, we use a two phase mode in describing and processing the component's functional and non-functional dependences.

The functional part is defined by XML schema and parsed by DRCR.

For the component's resource prerequisite description (non-functional part), the job of parsing this information is delegated to an external custom parsing service. The custom resolving service will process the parsed data and send results to the DRCR.

Figure 3 shows a fragment of meta-data file which contains the functional and non-functional contract of a smart camera component that can return regions of interests (subsets from a frame image data) on demand.

```
<? xml version="1.0" encoding="UTF-8"?>
<drt:component name="camera" desc="this is a smart camera
controller"          enabled="true">
<implementation                    bincode="ua.pats.demo.
smartcamera.RTComponent"/>
  <outport      name="images"      interface="RTAI.SHM"
type="Byte" size="400"/>
<inport     name="windposition"     interface="RTAI.SHM"
type="Integer" size ="1" optional="false"/>
  <property name="windowsize" type="Integer" value="40" />
<service-specific
language="UA.PATS.Language.SRDF">
    <periodictask cpuid="1" ram="20MB" priority="1"
    executiontime="1000" period="10000"
deadline="2000"/> </service-specific>
</drt:component>                              Non-functional
```

Figure 3. Component sample configuration

B. *Functional Description*

The functional description of component meta-data contains mainly about component functional attributes and requirements. It normally contains, for realtime application, the exposed and required port for data transmission, the component's properties for configuration. The functional description mainly based on our previous work in Hybrid real-time component model. Due to page limit, please refer our previous work [5] for details.

C. *Non-functional Prerequisite Description*

The non- functional prerequisite description is described in the service-specific element. This element has only one attribute – language, which designates which language is employed in describing the resource prerequisites. The DRCR itself will not try to parse the value of this element. It will just parse the language attribute and store the value of service-specific element as a normal string in the component instance registry and delegate it to the custom parser service to parse. To demonstrate the validation of our framework, we designed the Simple Task Description Format (STDF) for describing DRCom real-time task's characteristics.

Figure 3 shows an STDF meta-data for real-time task characteristics in the service-specific element. The Machine-type specifies that this component 's real-time execution parts is a periodic task, running on CPU 1, priority 1 with period 10000ns (at rate 100Hz) and execution time for each round of execution is 1000ns and deadline for overrun is 2000ns.

V. CONSTRAINT RESOLVING PROCESS

One of the key processes in the declarative real-time component framework is the constraint resolving process. It takes responsibility to identify the relationship between the real-time components as well as resources and to reason about possible reactions according to the current system context. Complying with the two phrase

When system configuration changes, e.g. because a new real-time component arrives, or the resource management service prompts that there is a noticeable resource change, DRCR will be activated. Firstly it will check existing DRCom's constraints and dependences. After the functional part was successfully checked and satisfied, the DRCR will determine whether the component's non-functional requirements could be guaranteed without jeopardizing the validity of other components' real-time contracts. If this is indeed the case, this new coming component is now ready to be activated.

Complying with two phase DRCom configuration, two phase constraint resolving was used. One is the Functional dependence resolving phase, which checks the functional dependence for any real-time component when system change occurs. In current prototype, the DRCR integrate the Function dependence resolving modules to achieve better performance and reduce implementation complexity. The other phase is the customized constraints resolving process. As the framework uses the OSGi's service-oriented model, system administrators can change the system resolving behavior by registering their own customized resolving service to fit specific system configurations.

A. *Functional Dependence Resolving*

1) *Circular Check:*

Real-time components in applications can be modeled as a component graph, which normally form a directed acyclic graph. In a task route, a task at a component has a certain set of outputs, which is a set of inputs for the task at next component along the task route. It's possible for a set of component descriptions to create a circular dependency. In this case at least one component configuration should not be satisfied.

2) *Port Compatibility Check*

In order to satisfy the functional dependency, all enabled and activated component's input ports should have corresponding output ports. It mean that the installed components dependence need to be checked when system structure ongoing a change. This guarantees that a component can only be initialized when all its dependences are satisfied. For instance, Figure 3 shows that the smart camera port needs an input port called "windowsposition". In order to satisfy the port compatibility check, one activated component instance with name = "ua.pats.trajectory.windows" and outport defined as follows

```
<outport name="windposition" interface="RTAI.SHM" type="Integer" size
="1"/>
```

B. *Custom Resolving process*

In practical environments, according to dynamic availability of resources and system's specific requirements, different algorithms for the resource allocation resolution may be desired. Thus the Custom Resolving service is designed aligning with service oriented mode to give customable support.

By using different customized resolving service providers, system behavior can be easily altered without the need to change the application main functional logic. This eased the need to recompile and re-load the whole application. The traditional admission service normal enables a component only when all the basic resource prerequisites are fulfilled. However, with context- specific knowledge, more flexible and complex resolving processes can be employed. One example can be: to stop low priority tasks or those tasks with optional port requirements to release their resource for incoming high priority real-time components.

In our system, several custom resolving service providers may co-exist in one system. DRCR will choose one candidate service provider which matches current system context. In current implementation, the system context is a global property managed by administrator. Our ongoing work is to employ the context reasoning scheme to get more accurate and flexible context information so as to make use of a more appropriate custom resolving service.

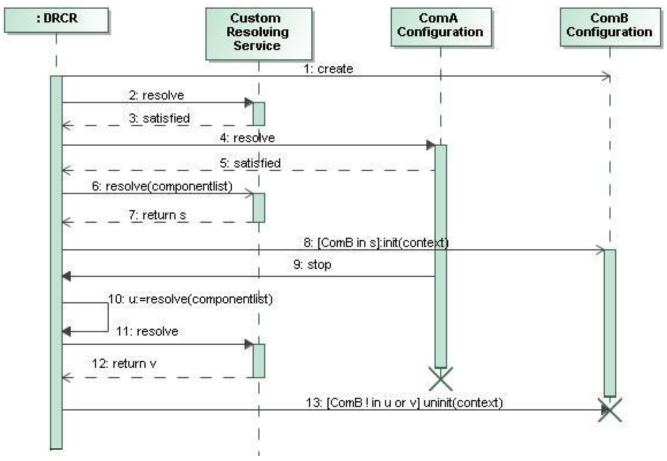

Figure 4. Sequence diagram for adaptation

C. *Adaptation Process*

Figure 4 shows a sequence diagram for adaptation. In this scenario, component B needs component A's output to satisfy its functional constraints. As an instance (ComAInstance) of component A configuration already exists, the DRCR will successfully solve B's functional constraints. Then, the internal resolving service and the external customized service will be consulted for real-time resolving process. When both services return positive results, the DRCR will create and activate the component B's instance (ComBInstance) according to its configuration. If ComAInstance is stopped, the DRCR gets notified about this event and consults its internal and external custom service again to check for newly (un)satisfied component instances. When it gets the results, the DRCR will find the ComBInstance is now unsatisfied and should be disabled.

VI. IMPLEMENTATION & SIMULATION RESULTS

In this section, we will use a scenario-based simulation to evaluate our adaptation approach across different application domains. Our framework has been validated both from a qualitative and a quantitative point of view. Firstly, we will discuss the system implementation.

A. *System implementation*

We use the OSGi framework as our management-layer implementation platform. OSGi technology serves as the platform for universal middleware ranging from embedded devices to server environments. The open standard simplifies the introduction of new capabilities and services. Equinox, is used as our basic development platform. In the real-time component implementation, we use our hybrid model proposed in our previous work [5].

This component model runs component management functions, management and lifecycle, in non-realtime environment and the small, predictable (functional) parts in a real-time environment. Here the real-time environment we used is Real-Time Application Interface (RTAI) –an open source real-time Linux kernel extension.

The DRCom model provides a basic management interface that enables uniform and coherent control for component's lifecycle and attributes. Each component is associated with a meta-data file which enlists its abstractions – for instance, interface for communication, attributes as well as reference constraints. Then, the component's functions are mapped to the RTAi real-time tasks. The real-time parts are controlled /configured via its non realtime peer to keep their state coherent via a clear designed asynchronized control interface [5].

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. A general scheme is very hard to achieve.

B. Simulation configuration

As shown from Figure 1, all six modules' execution parts are implemented as periodic tasks. Decoding component of TV application is implemented with a real-time task with period 33.37 ms with priority 2, execution time about 8 ms and deadline is 12 ms. The execution part of Transcoder module is implemented as with a real-time task with period 50ms with priority 1, execution time about 10 ms and deadline is 30ms. The schedule policy used by underline RTAI system is FIFO. In order to show the interference between these two components, others are implemented with little CPU time. The transmission methods are asynchronized (shared memory) so we can focus on the two coding module's performance.

As the video decoding execution time may vary according the contents of video streams, in order to more clearly demonstrate the mutual-influence among applications, we substituted the decoding function into an calculate function which use comparably constant CPU time for each round of calculation. So the execution influence of inter-influence can be more clearly indentified from the results. In Figure 1, the color of component shows state of each component after adaptation process.

C. Custom resolving policy

As described in section 2, when TV application and transcoding application are executed simultaneous, system may not have enough resources to guarantee their performance. Here, the custom resolving services will be employed to deal with the competing requirements. The strategy employed in our simulation is to protect the TV application as it normally matter user's experience most.

The custom resolving service will track the decoding component's overrun attribute via getproperty() defined in the component management interface. If the overrun increase too fast (by recording overrun values and time of retrieving the values), it will try to disable the component which consume most CPU time (exectutiontime/period), the algorithm 1 describe the adaptation process.

```
Algorithm 1: Guarantee TV application
Requires:  Installed components' CPU usage information
Ensure: Allocate enough CPU time to the TV application
If Decoder. getProperty(overrun)  increase > Max_threshhold
for all cmp in SatisfiedComponents do
    if component not from TV application
     get CPUusage from cmp attributes
      record the cmp with highest CPU usage.
    end if
end for
put cmp to disabled component list(dcl)
return dcl
```

After this custom resolving process, the Transcoder component will be disabled. Then, due to functional dependence constraint violation, the File Writer in the Transcoding application will also be disabled in functional resolving process. The different color in Figure 1 shows the states of installed components after these adaptation processes.

D. Simulation results

We perform 6000 observations for the execution time of Decoding Component in TV application. Table 1 shows the execution time in case of with and without adaptation framework support. In order to eliminate the transitory effects of cold start and adaptation, we collect measurements after the system has started and renders a steady execution. The execution time is express in micro-second (μs).

TABLE I. EXECUTION TIME (WITH/OUT ADAPTATION)

Context/ Execution Results	With Adaptation	No Adaptation
AVERAGE Execution Time (μs)	8127.297	10647.804
AVEDEV (μs)	4.15	3410.9
MIN (μs)	8121.8	8125.3
MAX (μs)	8175.1	18296.7
Overrun (times)	0	1880

In the formal case, the execution time of decoder is largely around 8ms, the biggest execution time is about 8.18 ms when the custom resolving service exist (it will disable transcode module). In contrast, if no such service available, the system will try to run these two applications simultaneously. The jitter of decoder task's execution time is very big. As we can seen from Table 1, the average deviation is 3410.9μs which is significantly worse than 4.15μs when our framework resolving mechanism exists. This may eventually lead to possibly transient timing problems of TV decoding task including missing frame, data overflows etc.

VII. RELATED WORK

Our platform follows the approach of e an architecture-based management framework for real-time system. That is: the real-time guarantee is not directly implemented within components' application code but provided by the container runtime environment according to additional component descriptors.

CIAO [10] builds a QoS-enabled CORBA Component Model (CCM). The project adhered to existing OMG specifications such as RT/CORBA and CCM. In contrast, our focus is on the challenges of simultaneously supporting real-time control and non real-time component adaptation 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 counterproductive to our system goals.

In the real-time component designs, Stewart et al. [12] designed a reconfigurable port-based object framework whose real-time communication scheme is similar to ours. Their pure real-time design philosophy makes this system hard to develop.

In the real-time Java specification, RTSJ [2] introduces the concepts of timeliness, schedule ability, and real-time synchronization to Java-based applications. It aims to enhance the Java platform with capabilities required by real-time applications. However, dynamicity of modules is not addressed, which would prevent an implementation of our concepts on top of this platform.

In order to deal with component dynamicity, Cervantes and Hall [14] propose a service-oriented component based framework for constructing adaptive component-based applications. The key part of the framework is the Service Binder which automatically controls the relationship between components. However, the static dependence description and resolving policy limits its application in normal OSGi applications and could not be applied to the real-time domain.

Aleš, Loiret et al. propose a component framework [15] for Java-based Real-time systems based on RTSJ. It provides continuum between the design and implementation process. By automatically generating an execution infrastructure, it mitigates complexities of RTSJ development and transparently manages the real-time concerns. However, their work lacks an automatic component dynamicity control scheme and their component dependence support is rather static and confined to the RTSJ domain specific knowledge.

Hartig and Zschaler designed and implemented enforceable component-based real-time contracts [17]. It runs large and complex parts in a classic non-realtime environment and only small, predictable parts in a real-time environment. However, although they propose the concept of adaptation manager for parameter adjustment and profile change, there is no formal design for how to deal with the dynamicity of component's availability which is crucial for downtime-free systems.

VIII. Conclusion and Future Work

This paper describes the experience in building a framework that supports run-time adaptation in response to the dynamic evolution and continuous deployments of modern complex real-time systems. In this framework, the real-time contract is specified in the component's meta-data. The component instance is managed by the system for the dependence resolving and real-time contract. Global view of current system configuration is managed by the DRCR service. Such system reasons about the changes in the system configuration and performs certain actions while still guaranteeing the designated real-time components' real-time contracts. It sheds the burden for each real-time component to monitor the system status and maintains the reference to the other dynamically available real-time components. Although our experience was done based on the OSGi middleware, we believe our findings to be general to be used in other architecture-based management systems.

Not all resource requirements can be specified statically. Here, resource usage is highly dependent on the execution context, which is not known at component deployment time. As our framework enables external adaptation polices to be injected into reconfiguration process, while flexibility in nature, how to verify if the context-specific reasoner gave invalid tactics and strategies become a big challenge.

References

[1] OSGi Service Platform Core Specification, http://www.osgi.org,2005

[2] The Real-Time Specification for Java, https://rtsj.dev. java.net /rtsj-V1.0.pdf,2001

[3] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded system design for automotive applications," Computer, vol. 40, no. 10, p. 42-+, Oct.2007.

[4] "RTAI Programming Guide," https://www.rtai.org/

[5] N. Gui, D. Florio, H. Sun, and C. Blondia, "A framework for adaptive real-time applications: the declarative real-time OSGi component model", The 7th Workshop on Adaptive and Reflective Middleware(ARM), 2008

[6] OSGi Alliance, "Declarative Service Specification," 2007.

[7] Sun Java Real-Time System, http://java.sun.com/javase/technologies/realtime/,2008

[8] N. Gui, V. D. Florio, H. Sun, and C. Blondia, "A Hybrid real-time component model for reconfigurable embedded systems," Proceedings of ACM symposium on Applied computing, Fortaleza, Brazil, 2008

[9] D. C. Schmidt and F. Kuhns, "An overview of the real-time CORBA specification," Computer, vol. 33, no. 6, June2000.

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

[11] 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.

[12] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas, "Design, implementation, and performance of an automatic configuration service for distributed component systems," Software-Practice & Experience, vol. 35, June2005.

[13] H. Cervantes and R. S. Hall, "A framework for constructing adaptive component-based applications: Concepts and experiences," Component-Based Software Engineering, vol. 3054, pp. 130-137, 2004.

[14] A. Plšek, F. Loiret, P. Merle, L. Seinturier, "A Component Framework for Java-based Real-Time Embedded Systems," Proceedings of ACM/IFIP/USENIX 9th International Middleware Conference (2008), Leuven, Belgium

[15] H. Hartig, S. Zschaler, M. Pohlack, R. Aigner, S. Gobel, C. Pohl, and S. Rottger, "Enforceable component-based realtime contracts," Real-Time Systems, vol. 35, no. 1, pp. 1-31, Jan.2007.

[16] G. Coates, "Real Time OSGi", http://www.osgi.org/wiki/uploads/VEG/Aonix-RT-OSGI.ppt, 2007

[17] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas, "Design, implementation, and performance of an automatic configuration service for distributed component systems," Software-Practice & Experience, vol. 35, June2005.