Real-Time Java in Control and Automation: A Model Driven Development Approach

Kleanthis Thramboulidis  
Electrical & Computer Engineering  
University of Patras  
26500, Patras, Greece  
thrambo@ee.upatras.gr

Alkiviadis Zoupas  
Electrical & Computer Engineering  
University of Patras  
26500, Patras, Greece  
azoupas@upnet.gr

Abstract

In this paper an approach for the transparent use of real-time Java in control and automation is described. The proposed approach, which is in the context of the Model Integrated Mechatronics paradigm, exploits the Function Block (FB) construct for the design model and the real-time Java specification for the implementation one. Specific interpreters allow for the automatic generation of FB types’ implementation models in terms of real-time Java and the transformation of the application’s FB networks to deployment specifications that are next utilized by a launcher to initialize and execute the control application on the proposed execution environment. The proposed approach favours deployment and re-deployment of distributed control applications and exploits the real-time specification for Java to meet their stringent non functional requirements.

1. Introduction

Java, as an Object-Oriented platform independent language, was not only adopted very soon in the traditional enterprise applications domain, but attracted from the beginning the attention of researchers in the manufacturing domain. Java provides an IT architecture for both process control and enterprise integration and has the potential to run the entire business enterprise as an integrated whole [1]. The Java platform enhances the effectiveness of the object-oriented development in terms of re-use of control technology both in software and hardware terms. Many positive experiences of using Java in control and automation have already been reported so far [2]. In spite of the many positive results [3]-[5], the wide use of Java in the control and automation domain is prohibited due to the following two main reasons: the first being Java’s limitations to address stringent real-time constraints and the second the fact that control engineers are not familiar with current software engineering practices and technologies.

However, language support is no longer a problem. Java’s limitations that make it inappropriate for the development of these counterparts of the manufacturing IT system that impose stringent real-time constraints, i.e. the way that memory is dynamically managed, the inability to access the underlying hardware and the unspecified semantics in its concurrent programming model, are successfully addressed by the evolving specification for real-time Java (RTSJ) [6][7]. Moreover, regarding the second reason, an approach accompanied with the proper tools to overcome this limitation is proposed in this paper.

The proposed approach exploits the new IEC61499 standard [8] that is based on the Function Block (FB) concept, and integrates it with current software engineering practices, such as model driven and component based development to allow the control engineer to work in the design level using the already widely accepted (in control and automation) concept of FB and transparently use real-time Java for the implementation model of the system. Archimedes ESS, an IEC-compliant Engineering Support System (ESS) that is in the context of the Model Integrated Mechatronics (MIM) paradigm [9], is utilized for the construction of the FB-based design model of the control application. However, any other IEC61499-compliant ESS can be utilized since model interchange is allowed through the use of XML. This paper extends the work presented in [9] by focusing on the implementation space of the MIM architecture. Its main contributions are:

1. The definition of a flexible FB implementation model for FB-based design models,
2. The definition of the mapping of FB design-space constructs to RTSJ implementation-space constructs,
3. The definition of an execution environment that is required for the deployment and predictable execution of the proposed FB implementation model, and
4. The development of the RTSJ-AXE package (as part of the Archimedes ESS) that utilizes the above infrastructure to allow the control engineer to exploit MDA in the domain of control and automation.

Other researchers are also working to provide
implementation environments for IEC61499 based DCSs. FBRT (http://www.holobloc.com), which is the first attempt to provide an execution environment for FB design models, utilizes Java but does not support timeliness neither run-time re-configurability. Brennan et al. [10] propose a FB-based model to support configuration and reconfiguration of DCSs and discuss its implementation on real-time Java. Tangermann et al. [11] examine the possibility of implementing FB design models using RTSJ. However, except that no proof of concept neither a prototype implementation is provided for the above approaches, it is almost impossible for control engineers that are accustomed with IEC models and languages, to utilize the proposed implementation environments and manually construct the implementation models. We are not aware of any other work that provides implementation models for IEC61499 compliant FB design models that meet real-time constraints and support run-time re-configurability.

The remainder of this paper is organized as follows: In the next section, the background of this work is presented. In section 3, the RTSJ-based Archimedes execution environment (RTSJ-AXE) package is presented. The implementation model framework that allows the automatic generation of RTSJ code from FB design diagrams is described in section 4. The architecture of the proposed execution environment is described in section 5. A re-configuration scenario is also described and performance results are given. Finally the paper is concluded in the last section.

2. Background work

2.1. The IEC61499 Function Block

The IEC61499 FB consists of a head and a body, where the head is connected to the event flows and the body to the data flows. The functionality of the function block is provided by means of algorithms, which process inputs and internal data and generate output data. The sequencing of algorithm invocations is defined in the FB type specification using a variant of statecharts called Execution Control Chart (ECC).

Control applications are defined as aggregations of FB instances that are interconnected through event and data connections. Figure 1 presents the FB network diagram of the Counter application that is used as a running example in this paper to illustrate the proposed approach. The Counter application is a rather simple application that abstractly represents the part of a real system that has to monitor and count the appearance of external events and respond to them. The system counts external events starting from a certain initial value, using given step and moving upwards and downwards between two given values. Counter and Corder are the basic FB types utilized by the application. E_INIT is used to initialize the application; E_CYCLE and HMI_FB are used to simulate the event producer and provide the data inputs required by the Counter FB, respectively.

2.2. The Real-Time Specification for Java

Java since its inception in 1995 has greatly grown in popularity. Characteristics such as safety, exception handling, multithreading and garbage collection, made the language popular and successful, but the large size, the nondeterministic behaviour and the poor performance have led to Java being unsuitable for the real-time and embedded domains. However, despite these drawbacks a wide variety of researchers recognized Java’s potential in real-time applications that resulted in the definition of the RTSJ. RTSJ in order to make Java more real-time defines: a) some modifications to the semantics of the JVM, and b) an additional package, namely javax.real-time that defines extra classes and interfaces. However, it should be noted that even though RTSJ is backwards compatible with existing non-real-time Java programs, it embodies a somewhat altered version of the Java principal that is Write Once, Run Anywhere (WORA) and leans strongly to Write Once Carefully, Run Anywhere Conditionally (WOCRAC) [6].

RTSJ received a high acceptance from vendors that resulted in a variety of real-time Java implementations and products. RI, the first reference implementation was developed by TimeSys (http://www.timesys.com) and runs on all Linux versions. However, its real-time characteristics, such as priority inheritance, can be fully exploited when the implementation runs under TimeSys’ real-time Linux which is released as a commercial version. Jamaica, a different implementation provided by Aicas GmBH (http://www.aicas.com) runs on a wide variety of Real Time Operating Systems. Jamaica provides the programmer with a real-time garbage collector, making memory programming a more relaxed process compared to the RTSJ one [12]. However, this inherits the drawback to the developed application, of not being compatible with other RTSJ VMs. Other JVM implementations compliant with RTSJ are: Aero JVM (http://www.aero-project.org/), jRate (http://jrate.sourceforge.net/) and the JVM developed by ajile Systems (http://www.ajile.com). The later running directly on hardware opens new horizons on the application of RTSJ in embedded devices.

Figure 1. The Counter example FB-based control application.
2.3. Archimedes system platform

The Archimedes system platform, a prototype implementation based on the Model Integrated Mechatronics (MIM) paradigm [9], is composed of a methodology, a framework and an ESS. The Archimedes ESS that currently supports the design and deployment phases in the application and partially in the resource layers of the MIM architecture was developed utilizing the General Modelling Environment (GME). GME is a configurable toolset with generic functionality for graphical development that supports the easy creation of domain-specific modelling and program synthesis environments [13]. The FB-type editor, the ECC editor, and the FB-network editor are the basic components of Archimedes ESS which can be downloaded from http://seg.ee.upatras.gr/ mim/download.htm.

Archimedes ESS can be used to define new FB types and FB networks or import these artefacts from IEC-compliant XML specifications that have been produced by other IEC-compliant ESSs. These FB design models are further refined and enhanced to capture the real-time constraints of the control application. The so constructed platform independent models, utilizing the approach proposed in this paper, are transformed by specific interpreters to the proposed RTSJ-based FB platform-specific implementation model that can be executed on the proposed execution environment running on RTSJ compliant implementations.

3. The RTSJ-based Archimedes execution environment package

Archimedes supports many execution environments. Each execution environment is supported by a specific package called Archimedes eXecution Environment (AXE) package. RTSJ-AXE is the package that allows Archimedes to exploit RTSJ for the execution of FB-based DCSs.

The RTSJ-AXE package extends the functionality of Archimedes system platform so as to exploit RTSJ in the model driven development process of distributed control applications. It is composed of:

a) An FB implementation model framework, i.e. a set of classes that enable the re-use of all these design decisions that have been done for the proper use of RTSJ constructs in mapping FB based design specifications of control applications to executable real-time Java implementations.

b) An execution environment that is required for the deployment and execution of the proposed FB implementation model. This environment provides the infrastructure required to meet deployment and redeployment needs, as well as stringent non-functional requirements such as maximum permissible response times, minimum throughputs and deadlines usually imposed by the nature of DCSs.

c) A set of interpreters to automatically generate the implementation model from the FB design model.

d) A tool (RTSJ launcher) to support the preparation and launching of the application on the target environment.

For the definition of the RTSJ-AXE package the following enhancements proposed by RTSJ to the Java specification were mainly exploited.

RealtimeThread and NoHeapRealtimeThread classes, introduced by RTSJ as new schedulable objects extending the Thread class, are utilized to implement both event and time triggered FBs since they can be periodic or aperiodic in their execution. NoHeapRealtimeThread allows selected FB implementations to be independent from the GC, since it utilizes memory areas not handled by the GC. These memory areas known as “immortal” and “scoped” memory are defined and represented by specific classes of RTSJ. However, even though the ability to use other memory areas than the Heap is a quite powerful feature, it is very hazardous and demands an experienced programmer to be used correctly [14]. This later disadvantage is bypassed to a large extend by the proposed model driven approach.

The explicitly supported by RTSJ priority scheduler that supports priority inheritance provides up to 28 priority levels, to map the FB instance priorities of the FB network. FBs’ execution times and their CPU costs and deadlines can be affected utilizing the specific parameters that are defined to influence the scheduling of real time threads. The new mechanism for Asynchronous Transfer of Control that was defined and integrated to the Java exception handling mechanism can be utilized for an FB to terminate its execution in a timely and safe manner.

Event connections are implemented by means of the Asynchronous Event Handling Mechanism that was expanded in RTSJ by generalizing the traditional Java event handlers and allowing them to become schedulable entities. The so defined handlers act as real time threads and inherit all the scheduling characteristics of threads.

Special queues, that have been introduced, allow the implementation of interconnections between FBs, represented as real time threads, and execution environment entities or FBs, represented as non real-time threads. This provides the infrastructure for a secure non blocking communication between threads of different kinds and help to avoid any unpredictable interaction with GC. Timers can be created and defined to be fired at certain points of time, while time can be calculated in RTSJ with nanosecond accuracy due to the high resolution time types that were introduced. Finally, the possibility provided by RTSJ to access physical and raw memory is expected to enhance the way that mechanical-process-interface FBs would be implemented in the RTSJ-AXE package.
4. The implementation model framework

This section describes the most important design decisions in the proposed mapping of the IEC-compliant FB-based design specifications to RTSJ compliant Java code. These design decisions have been captured in the implementation model framework which is composed of a set of classes that are utilized by Archimedes interpreters to automatically produce an RTSJ compliant implementation model from FB based design specifications.

A part of the class diagram of the proposed framework is given in fig. 2. Basic classes include: FB, FBType, BasicFBType, ECCType, ECState, ECAction, ECTransition, OutputEvent, InputEventMonitor, and IECDataType. BasicFBType is a key class of the framework that inherits the FBType class and implements the FB interface both members of the same framework as shown in figure 2. It contains static data members for EC state names as well as input and output event and data. Every FB type of the FB design diagram is mapped to a class that inherits BasicFBType. This class contains an initializer which initializes the static variables to setup the class for the execution environment. It also creates the implementation constructs for the FB type’s ECC. Methods of the class include:

a) a constructor for the construction of the implementation artefact that corresponds to FB type’s instance,

b) a number of instance methods that correspond to the algorithms of the implemented FB type, and

c) a number of instance methods that correspond to the transitions of the FB type’s ECC.

Methods resulting from the FB type’s algorithms are generated utilizing the algorithmImplementation field of the corresponding Archimedes GME-atom, which should be defined during the FB type’s definition time by the control engineer. Methods resulting from ECC transitions are generated utilizing the information captured at design time in the Event and Condition fields of the corresponding Archimedes GME-connections. Whenever an ECTransition instance is created, the corresponding condition’s method is passed as parameter to the constructor. The same technique is applied for the construction of ECAction instances where the corresponding algorithm’s method is passed as parameter.

This mapping results to a completely reusable structure which utilizes Java’s reflection API and simplifies the translation process of FB design diagrams to RTSJ compliant code. It also provides a flexible implementation model that exhibits the following behaviour during run-time. Whenever the FB instance is activated, either from the presence of an external event (event triggered) or from the progression of time (time triggered), the transitions leaving an EC state are examined by sending the checkTransitions message to the ECState instance that corresponds to its current state. This method sends the check message to the ECTransition instances associated with the given State. As a response to the check message the transition instance sends the invoke message to the corresponding condition that is an instance of the Method class. Java’s reflection mechanism finds the appropriate method in the corresponding FB type class and executes it. The value of the transition condition is calculated and returned to the ECTransition instance and subsequently to the ECState instance. A slightly different but simpler process is used for the execution of algorithms.

Figure 3 shows part of the specification of the Counter class that was produced automatically by the appropriate Archimedes ESS FBType2RTSJ interpreter using as input the IEC61499 specification of the Counter FB type. The static initializer of the class acts as factory of the corresponding FB’s ECC. The constructor of the Counter class is utilized by the Deployment
Management Entity of the execution environment to create the required Counter FB instances. Methods, such as the countdown, that implement the algorithms of the FB type and the ones that implement the transitions of its ECC, such as the start2countDown, complement the specification of the class.

```java
public class CounterFB extends BasicFBType{
    //EVENT OUTPUT NAMES
    public static final int CNF=1;
    //STATE NAMES
    public static final int START=0;
    public static final int CountDOWN=1;
    public static final int CountUP=2;
    
    static{ //ECC BUILDING
        //ACTION DECLARATION
        itsActions[CountDOWN]=new ECAction(Class.forName("CounterFB").getMethod("CountDOWN",null),CNF);
        //STATE DECLARATION
        itsStates[CountDOWN]=new ECState(itsActions[CountDOWN]);
        //TRANSITION DECLARATION
        itsTransitions[CountDOWN]=new ECTransition({Class.forName("CounterFB").getMethod("CountDOWN2START",null),itsStates[START]});
        //STATE TRANSITION ASSOCIATION
        itsStates[CountDOWN].updateTransitions(itsTransitions[CountDOWN]);
    }
    public CounterFB(){
        //INPUT EVENT INITIALIZATION
        itsInputEventMonitor=new InputEventMonitor numOfEventInputs,this);
        //ECC INSTANTIATION
        itsEcc=new ECC(sch,null,this,itsStates[START],false);}
    //ALGORITHM
    public void CountDOWN(){
        TEMP.setValue(TEMP.getValue()-
        STEP.getValue());
        COUNT.setValue(WIN.getValue());
        if((TEMP.getValue()==MIN.getValue())){
            CEILING.setValue(false);
            FLOOR.setValue(true);
        }else{
            CEILING.setValue(false);
            FLOOR.setValue(false);
        }
        //TRANSITIONS
        public boolean START2CountDOWN(){
            if((itsEventInputs[REQ]==true)&&((DOWN.getValue()==true)&&(UP.getValue()==false)){
                clearInputEvent(REQ);
                return true;
            }else{return false;}}
    }
}
```

**Figure 3. Part of the automatically produced java specification for the Counter FB type.**

One of the actions performed by the constructor of the java class that represents the FB type is the construction of an instance of the class ECC and the assignment of its reference to the corresponding static variable (itsEcc). The ECC class belongs to the Archimedes RTSJ framework and allows the ECC to inherit the NoHeapRealtimeThread class and to be constructed and executed in special memory areas such as ImmortalMemory and ScopedMemory, in order to avoid the overhead that can be added to its execution by an unexpected invocation of the GC. This special handling of the real-time threads is not required when using JamaicAVM, since it replaces the traditional GC of the JVM with a real-time GC, which is not allowed to preempt real-time threads. In this case ECC inherits the RealtimeThread class.

Another important action of the constructor is the initialization of the ItsInputEventMonitor static data member. ItsInputEventMonitor is a monitor defined to implement the semantics of the FB execution related to event flows as defined by the IEC61499 standard.

For the implementation of event flows of the FB network the asynchronous event handling mechanism of RTSJ was utilized. This mechanism allows the bounding of Events to hardware or software happenings, such as interrupts or software implemented timers.

Whenever a happening occurs the mechanism is able to create a new thread and execute the piece of code that is defined by the programmer. Each output event of the FB type is represented as an instance of the OutputEvent class, which extends the AsyncEvent RTSJ API class. Every OutputEvent instance is bound to an instance of the EventHandler class, which inherits the AsyncEventHandler or the BoundAsyncEventHandler class. The BoundAsyncEventHandler class is permanently associated with a server real-time thread instead of the AsyncEventHandler class that adopts the dynamic binding of the handler to a server real-time thread. The first alternative has the drawback of adding an extra thread to our application, while the second one provides a way to avoid the extra overhead that is introduced to the implementation model by the dynamic binding of the handler to a server real-time thread. Extra information has to be annotated to the FB network model to allow the tool to automatically select between the two alternatives.

The complexity of the java code that is required to express the semantics of the FB type and the FB network made evident that not only it is not possible for a control engineer to implement a design specification and execute it on a RTSJ compliant JVM, but it is even a difficult task for an experienced Java programmer. The only solution to this problem is the automatic generation of the implementation code from the appropriate design space specifications. Specific interpreters have been developed as plug-ins of the General Modelling Environment to enable the control engineer to effortlessly and transparently exploit the benefits of real-time Java and all the RTSJ compliant products and tools that are expected to appear soon in the market.

Two interpreters have been developed to automate the transformation process of FB design specs to RTSJ specs. The FBTType2RTJava interpreter translates the applications FB types to corresponding real-time Java code. For the transformation of the applications FB
networks a two phase process was adopted. Each FB network is translated by the FBNet2XML interpreter to an XML based deployment specification. A specific tool (RTSJ launcher) has been developed to realise the so produced deployment specification on the target execution environment.

5. The execution environment

The execution environment is composed of a set of classes that provide the infrastructure required by an RTSJ compliant execution environment to support deployment and predictable execution of FB based design specifications as well as run-time reconfiguration. Between the most important classes of this category are: DeploymentManagerEntity (DME), DataConnectionManager (DCM), EventConnectionManager (ECM) and EventHandler.

During the establishment of an event connection between two FB instances, the DME subscribes the consumer FB to the EventHandler of the OutputEvent that represents the corresponding output event, by calling the subscribe() method on the proper instance of the EventHandler class. The DME can subscribe an FB instance to more than one EventHandler instances that allows an FB to receive notifications from multiple event producers (FB instances).

Run-time re-configuration is based on the fact that DME is allowed to subscribe and unsubscribe FB instances from handlers by calling the unSubscribe() method even during run-time. The re-configuration process that takes place during run-time is carried out in two phases. The 1st half runs in low priority; it includes actions that can be considered as preparation steps and can be interrupted by higher priority threads, for the control application to be executed without missing deadlines. Such actions include downloadFBType, createFBInstance and createDataConnection. The 2nd half runs in high priority; it contains all those actions that have to be executed with run-to-termination semantics. At this phase consistency issues for re-configuration have not been considered yet.

During run-time when an FB instance reaches the point in its ECC where it should signal its output events, the fire() method, defined in the AsyncEvent class, is called on the OutputEvent instances associated with this action. The handler instances that are associated with the output events are released and the overridden handleAsyncEvent() method of each instance is executed as shown in the sequence diagram of figure 4. The handlers are charged with the task to inform the consumer FB instance InputEventsMonitor of the newly generated input event and to awaken the FB instance, in the case that the FB is event-triggered. When the consumer FB instance is awaken it reads from the InputEventMonitor the input events and enters its ECC in order to check if any of its transition conditions is satisfied.

![Figure 4. Sequence diagram for event notification.](image)

The Counter example application presented in figure 1 is used in the experiment, which is described in the rest of this section, to demonstrate the applicability of the proposed approach regarding re-configuration. The functionality of the 1st configuration scenario of the Counter application has as follows. The hmi FB instance receives a signal from the user and generates the Con event output to initialize the remaining FB instances with the provided by the hmi output data values. After initialization the eCycle FB instance generates events that are counted by the counter FB instance which is controlled regarding the counting direction by the corder FB instance.

To analyze the timing behaviour of our prototype execution environment the Counter example application was developed, deployed and re-deployed. As target device was used a PC with Intel Pentium 4 CPU at 2,40 Ghz with 512Mb RAM. The software platforms used to test the features of RTSJ-AXE package are:

1. Fedora Core 2 Kernel 2.6.5 running JamaicaVM 2.6 Release 2 (Build 702) of Aicas Gmbh.
2. TimeSys Linux/RT (GPL version) 4.1 Kernel 2.4.21 with TimeSys Reference Implementation (RI) ver 1.0-547.1. RI implements all the mandatory features of RTSJ being a fully compliant implementation.

Archimedes editors were used to construct the application’s FB network and define the corresponding FB types. The RTSJ-AXE interpreters were used to automatically produce the real-time Java code of the application that corresponds to FB-based design. The resulting implementation model of the control application was deployed on the proposed execution environment using the Archimedes launcher. The whole development process is shown in fig. 5.

Table I presents the timing characteristics of the deployment process. It must be noted that the 5 instances were all of different FB types so the load time for 5 different classes is responsible for the long total instantiation time. The 1\(^{st}\) half of the deployment task was executed with a priority 20, while the 2\(^{nd}\) half with very high priority (35).
Table I. Deployment timing characteristics of Counter application.

<table>
<thead>
<tr>
<th>Action</th>
<th># of times performed</th>
<th>Total time</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Jamaica VM</td>
<td>RI</td>
<td></td>
</tr>
<tr>
<td>FB instantiation</td>
<td>5</td>
<td>105,953 ms</td>
<td>24,251 ms</td>
<td></td>
</tr>
<tr>
<td>Data connection</td>
<td>10</td>
<td>848 us</td>
<td>686 us</td>
<td></td>
</tr>
<tr>
<td>Event connection</td>
<td>8</td>
<td>6,889 ms</td>
<td>19,752 ms</td>
<td></td>
</tr>
<tr>
<td>Start FB instance</td>
<td>5</td>
<td>2,361 ms</td>
<td>4,810</td>
<td></td>
</tr>
</tbody>
</table>

Table II presents timing characteristics regarding FB execution time for an FB with algorithms having zero execution time and one event signalled per EC action. The average execution time was calculated over 1000 measured values. FB execution time was measured for 1, 2 and 3 sequential transitions and return to the EC initial state. Table III presents the event connection latency giving average and maximum values. Much better results are expected for Jamaica executed on an RTOS.

After the deployment of the 1st version a re-deployment scenario was executed. Two instances, one of E_SPLIT type and one of PrintInt type, were appended to get the final version that is the one shown in figure 6. This version has exactly the same behavior as the 1st one except that the current value of the counter is printed on the display by the printInt FB instance.

Table II. Timing characteristics of FB execution.

<table>
<thead>
<tr>
<th># of transitions</th>
<th>Average execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Jamaica VM</td>
</tr>
<tr>
<td></td>
<td>avg</td>
</tr>
<tr>
<td>1</td>
<td>374 us</td>
</tr>
<tr>
<td>2</td>
<td>513 us</td>
</tr>
<tr>
<td>3</td>
<td>650 us</td>
</tr>
</tbody>
</table>

Table III. Event connection latency in us.

<table>
<thead>
<tr>
<th>Event</th>
<th>Jamaica VM</th>
<th>RI</th>
<th>avg</th>
<th>max</th>
<th>avg</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle to Counter REQ</td>
<td>142</td>
<td>147</td>
<td>42</td>
<td>49</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle to Cycle Start</td>
<td>348</td>
<td>361</td>
<td>52</td>
<td>59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counter to Corder REQ</td>
<td>137</td>
<td>139</td>
<td>80</td>
<td>90</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first half of the executed re-deployment scenario, which includes 2 FB instantiations of new FB types, requires 26,528 ms (8,147 ms for RI) while the 2nd half, which includes 2 delete-event actions, 5 create-event actions and 2 startFB actions, requires 5,111 ms (16,125 ms for RI). Table IV presents detailed timing characteristics regarding redeployment.

Figure 5. The proposed development process
6. Conclusions

The IEC61499 FB-based model is a promising technology that can guarantee the always increasing requirements for agile manufacturing. Even though the standard was recently accepted, the absence of methodologies, environments and tools that should enable its adoption in industry is evident. The presented in this paper model driven development approach integrates the FB model with RTSJ to provide an important enabling technology for reusing both the architecture and the functionality of both models. It allows the control engineer to ignore all the complexities inherent in field devices and execution environments and concentrate on the actual problem to be solved working with the already known FB notation.

The proposed approach exploits already existing skills on FB issues and allows control engineers to exploit in a transparent way current trends in software engineering. It utilizes the FB construct as the main building block for constructing the software counterpart of the mechatronic component that is the basic component for the construction of the next generation mechatronic systems.

The RTSJ-AXE prototype implementation demonstrates the applicability of the proposed approach. Even though RTSJ implementations are in their early steps and significant improvements in performance are expected the test results regarding performance characteristics were very promising for their acceptance in the control and automation domain.

Acknowledgments

Thanks are due to Aicas Gmbh for the support in using Jamaica. We gratefully thank G. Doukas for his contribution in executing the experiment and N. Papakonstantinou for the prototype implementation of Archimedes RTSJ interpreters.

References