XRTJ*: An Extensible Distributed High-Integrity Real-Time Java Environment

Erik Yu-Shing Hu, Jagun Kwon and Andy Wellings

Real-Time Systems Research Group
Department of Computer Science
University of York, York, YO105DD, UK
{erik,jagun,andy}@cs.york.ac.uk

Abstract. Despite Java's initial promise of providing a reliable and cost-effective platform-independent environment, the language appears to be unfavourable in the area of high-integrity systems and real-time systems. To encourage the use of Java in the development of distributed high-integrity real-time systems, the language environment must provide not only a well-defined specification or subset, but also a complete environment with appropriate analysis tools. We propose an extensible distributed high-integrity real-time Java environment, called XRTJ, that supports three attributes, i.e., predictable programming model, dependable static analysis environment, and reliable distributed run-time environment. The goal of this paper is to present an overview of our on-going project and report on its current status. We also raise some important issues in the area of distributed high-integrity systems, and present how we can deal with them by defining two distributed run-time models where safe and timely operations will be supported.

Keywords : Real-Time Java, High-Integrity Real-Time Systems, Static Analysis Environment, Distributed Run-Time Environment

1 Introduction

There is a trend towards using object-oriented programming languages, such as Java and C++, to develop high-integrity real-time systems because the use of such languages has several advantages, for instance reusability, data accessibility and maintainability. Typically, high-integrity systems, where failure can cause loss of life, environmental harm, or significant financial penalties, have high development and maintenance costs due to the customised nature of their components. Therefore, the use of object-oriented programming in such systems may offer a number of benefits including increased flexibility in design and implementation, reduced production cost, and enhanced management of complexity in application areas.

* This work has been funded by the EPSRC under award number GR/M94113.

The Java technology with its significant characteristics, including cost-effective platform-independent environment, relatively familiar linguistic semantics, and support for concurrency, has many features for developing real-time and embedded systems. It also provides well-defined *Remote Method Invocation* (RMI) features which support distributed applications on the Java architecture.

However, despite Java's initial promise, the language appears to be unfavourable in the area of high-integrity systems [22] and real-time systems [7]. Its combination of object-oriented programming features, its automatic garbage collection, and its poor support for real-time multi-threading are all seen as particular impediments.

The success of high-integrity real-time systems undoubtedly relies upon their capability of producing functionally correct results within defined timing constraints. In order to support a predictable and expressive real-time Java environment, two major international efforts have attempted to provide real-time extensions to Java: the Real-Time Specification for Java (RTSJ) [5] and the Real-Time Core extensions to Java [9]. These specifications have addressed the issues related to using Java in a real-time context, including scheduling support, memory management issues, interaction between non-real-time and real-time Java programs, and device handling, among others.

However, the expressive power of all these features, along with the regular Java semantics, means that very complex programming models can be created, necessitating complexity in the supporting real-time virtual machine and tools. Consequently, Java, with the real-time extensions as they stand, seems too complex for confident use in high-integrity systems. Furthermore, in addition to the difficulties with analysing applications developed in these frameworks with all the complex features, there is no satisfactory static analysis approach that can evaluate whether the system will produce both functionally and temporally correct results in line with the design at run-time.

For the above reasons, to encourage the use of Java in the development of high-integrity real-time systems, the language environment must provide not only a well-defined specification or subset, but also a complete environment with appropriate analysis tools. Hence, we propose an extensible distributed high-integrity real-time Java environment, called XRTJ, that supports the following attributes:

- Predictable programming model
- Dependable static analysis environment
- Reliable distributed run-time environment

The XRTJ environment has been developed with the whole software development process in mind: from the design phase to run-time phase. The XRTJ environment includes: the Ravenscar-Java profile [23], a high-integrity subset of RTSJ; a novel Extensible Annotations Class (XAC) format that stores additional information that cannot be expressed in Java class files [18]; a static analysis environment that evaluates functional and temporal correctness of applications, called XRTJ-Analyser [18]; an annotation-aware compiler, called XRTJ-

Compiler; a modified real-time Java virtual machine, called XRTJ-Virtual Machine that supports a highly reliable run-time environment.

The aim of the paper is to present an overview of our on-going project and report on its current status. The rest of the paper is organised as follows. Section 2 presents an overview of the XRTJ environment. Further details of the static analysis environment and distributed run-time environment are provided in Section 3 and 4 respectively. Section 5 shows a simple example that demonstrates how our approach can be used in a practical application. Section 6 gives a brief review of related work while Section 7 presents the current status of the project. Finally, conclusions and future work are presented in Section 8.

2 XRTJ Environment Overview

The major goal of our project is to provide a predictable and portable programming environment to develop distributed high-integrity real-time systems. The XRTJ environment is targeted at cluster-based distributed high-integrity real-time Java systems, such as consumer electronics and embedded devices, industrial automation, space shuttles, nuclear power plants and medical instruments.

To encourage the use of real-time Java in high-integrity systems, we have introduced the Ravenscar-Java profile [23]. The profile or restricted programming model excludes language features with high overheads and complex semantics, on which it is hard to perform temporal and functional analyses. Further details of the profile are given in Section 2.1.

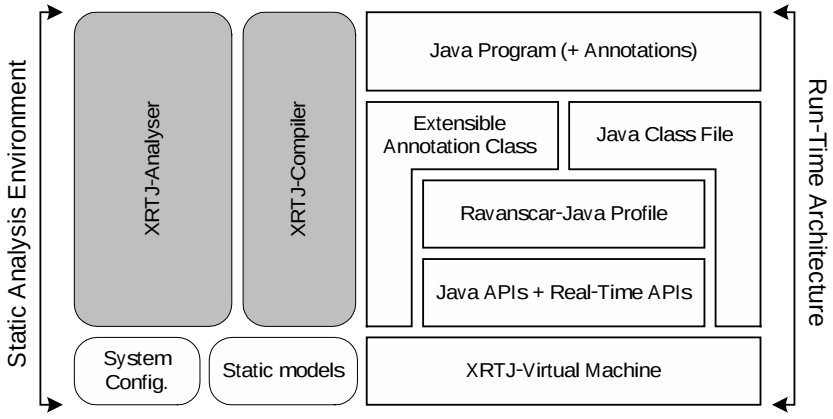


Fig. 1. A basic block model of the XRTJ environment

Based on the Ravenscar-Java profile, we propose a highly dependable and predictable programming environment to develop distributed high-integrity real-time applications. As shown in Figure 1, the XRTJ environment can be divided

into two main parts: a *Static Analysis Environment*, which offers a number of tools that conduct various static analyses including program safety and timing analysis; a *Distributed Run-Time Environment*, in which highly predictable and dependable distributed capabilities are provided.

Before a detailed discussion of each environment, two major components of the XRTJ environment will be introduced. In our environment, to facilitate the various static analysis approaches and provide information that cannot be expressed in either Java source programs or Java bytecode, an extensible and portable annotation[1] class format called Extensible Annotations Class (XAC) file is proposed [18]. To generate XAC files, an annotation-aware compiler, named XRTJ-Compiler, which can derive additional information from either manual annotations or source programs, or both, is also introduced. Taking advantage of the knowledge accumulated with the compiler, different analysis tools may be integrated into the XRTJ-Compiler to carry out various verifications or validations either on source programs or Java bytecode.

Essentially, the *static analysis environment* supports various analysis techniques by means of the XRTJ-Analyser where program safety analysis and timing analysis can be statically carried out. In the XRTJ environment, Java programs extended with specific annotations, such as timing annotations or model checking annotations[2], are compiled into Java class files and XAC files by either a simple XAC translator and a traditional Java compiler or the XRTJ-Compiler. A conformance test that verifies whether the applications obey the rules defined in the Ravenscar-Java profile or whether the manual annotations are correct can also be conducted during the compilation. The XAC files, together with the Java class files, are used by the XRTJ-Analyser to perform various static analyses. As shown in Figure 1, various *static models*, such as a *Virtual Machine Timing Model* (VMTM)[3], can be provided to perform different static analysis approaches on the XRTJ-Analyser. Further aspects of the static analysis environment are discussed in Section 3.

The *distributed run-time environment* provides mechanisms for underlying systems to facilitate both functionally and temporally correct execution of applications. This infrastructure is targeted at cluster-based distributed infrastructure where remote objects are statically allocated during the design phase. In order to accommodate a diverse set of the implementations on the underlying platforms or virtual machines, two run-time environments with different levels of distribution are supported in the XRTJ run-time environment. This will be explored further in Section 4.

[1] The term *annotations*, in this paper, means both manual annotations and annotations generated by the XRTJ-Compiler automatically.

[2] Model-checkers, such as JPF2[6], which requires special annotations, may be employed in our architecture to facilitate safety checks of concurrent programs.

[3] VMTM is a timing model for the target virtual machine including a list of the worst-case execution time of native methods and Java bytecode instructions.

2.1 Ravenscar-Java Profile

We have presented a Java profile for the development of software-intensive high-integrity real-time systems in [23]. The restricted programming model removes language features with high overheads and complex semantics, on which it is hard to perform timing and functional analyses. The profile fits within the J2ME framework [31], fullfils the NIST Real-Time Java profile requirements [7] and is consistent with well-known guidelines for high-integrity software development, such as those defined by the U.S. Nuclear Regulatory Commission [16].

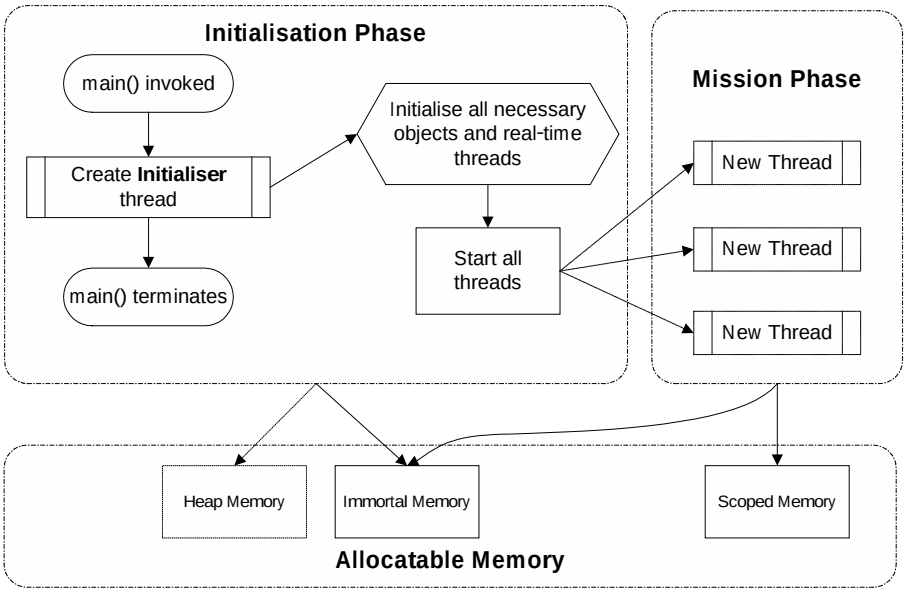


Fig. 2. Two execution phases of Ravenscar Virtual Machine

Its computational model defines two execution phases, i.e. initialisation and mission, as shown in Figure 2. In the initialisation phase of an application, all necessary threads and memory objects are created by an `Initializer` thread, whereas in the mission phase the application is executed and multithreading is allowed based on the imposed scheduling policy. There are several new classes that should ultimately enable safer construction of Java programs (for example, `Initializer, PeriodicThread,` and `SporadicEventHandler`), and the use of some existing classes is restricted or simplified due to their problematic features in static analysis. For instance, the use of any class loader is not permitted in the mission phase, and the size of a scoped memory area, once set, cannot be changed.

Further restrictions include (see [23] for a full list)

– No nested scoped memory areas are allowed,

- Priority Ceiling Emulation must be used for all shared objects between real-time threads,
- Processing groups, overrun and deadline-miss handlers are not supported,
- Asynchronous Transfer of Control is not allowed, and
- Object queues are not allowed (i.e. no `wait, notify,` and `notifyAll` operations).

Restrictions are also imposed on the use of the Java language itself, for example

- `continue` and `break` statements in loops are not permitted, and
- Expressions with possible side effects must be eliminated.

Most subsets of Java or the RTSJ (e.g. [3, 28]) overlook some important elements of the language, for example, multithreading and the object-oriented programming model. Thus many of the advantages of Java are lost. However, the Ravenscar-Java profile attempts to cover the whole language issues, as well as the run-time model. The profile is expressive enough to accommodate today's demanding requirements for a powerful programming model, yet concise enough to facilitate the implementation of underlying platforms of virtual machines.

3 Static Analysis Environment

The static analysis environment consists of two components: program safety analysis and timing analysis. The former highlights program safety in terms of functional correctness and concurrency issues, such as safety and liveness, whereas the latter emphasises the analysis of timing issues in terms of temporal correctness. For the most part, these static analysis approaches may be carried out individually or combinatorially. A block diagram of the XRTJ architecture for the static analysis environment is given in Figure 3 and further details of each major component are discussed in subsequent sections.

3.1 XAC (Extensible Annotation Class) File

One of the key components in the XRTJ architecture is the XAC format that provides information for the various analysis tools that cannot be stored in Java class files without making them incompatible with the traditional Java architecture [18]. The XAC format has been designed with two main goals in mind: *portability*, to support both platform independence and language independence, and *extensibility*, to hold extra information needed for other analysis tools. Therefore, the XAC files are easy to extend for various purposes or apply in annotation-aware tools or JVMs.

Each XAC file is generated for a specific Java class file, and so the relationship between a Java class file and an XAC file is one to one. Essentially, the offset numbers of bytecode in a method are stored with the associated annotations in the XAC file. Therefore, the corresponding bytecode and annotation may easily

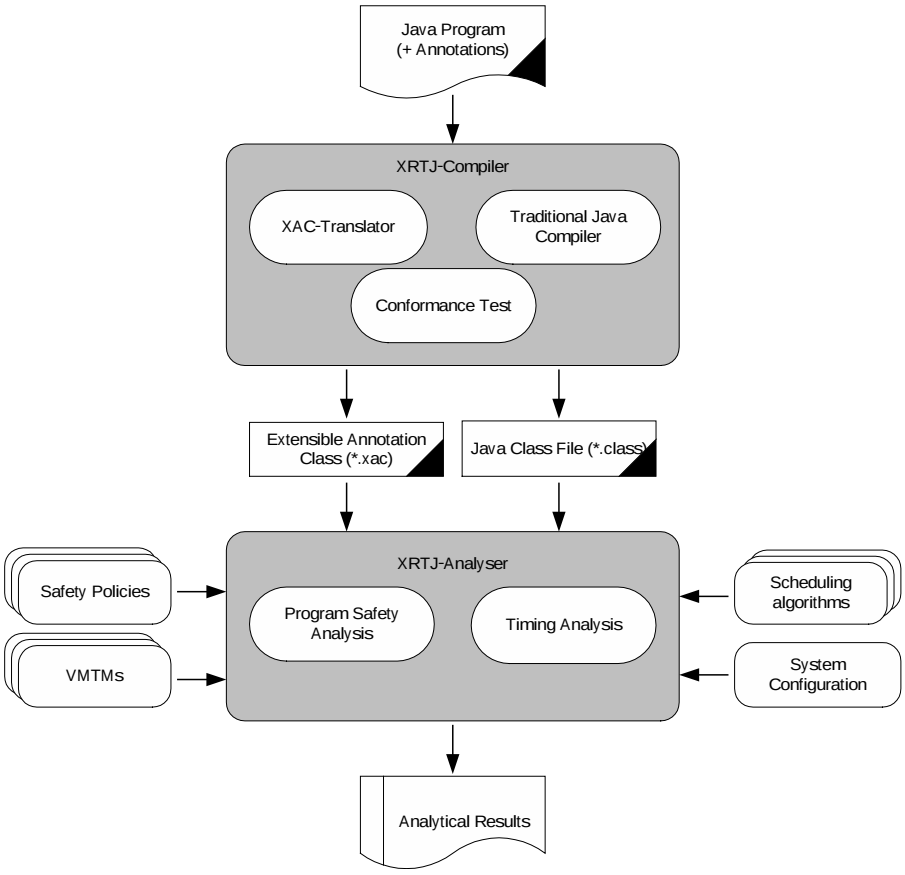


Fig. 3. A block diagram of the XRTJ architecture for static analysis environment

be reconstructed in analysis tools. A checksum is also provided in XAC files to facilitate analysis tools or JVMs to verify the consistency between the Java class file and the XAC file. Further details of the XAC file are discussed in [18].

In addition, using XAC files has benefits for distributed systems as XAC files do not increase the size of traditional Java class files. Therefore, if the XAC files are not required at run-time, they do not need to be either loaded into the target JVM or transferred among distributed machines.

3.2 XRTJ-Compiler

Compiler techniques have been applied to analysis approaches, such as worst-case execution time analysis and program safety analysis, in order to achieve more accurate results. For example, Vrchoticky [35] has suggested compilation support for fine-grained execution time analysis, and Engblom et al. [13] have proposed a WCET tool called *Co-transformation*, integrated with compilation support,

to achieve safer and tighter estimation of timing analysis approaches. These approaches show that compilation support can not only address the optimisation issues introduced by compilers, but also provide additional information that may accumulate from the source code level for particular analysis tools.

In the XRTJ environment, an annotation-aware compiler (XRTJ-Compiler) is introduced in order to both manipulate annotations and validate that the contexts of source program code obey those rules defined in the Ravenscar-Java profile. On the whole, the XRTJ-Compiler extracts both manual annotations introduced for timing analysis and specific annotations that can be derived from source code level for particular purposes. For instance, the XRTJ-compiler derives *Abstract Syntax Trees*(AST) and *Worst-Case Execution Frequency* (WCEF)[4] vectors of specific applications to facilitate the WCET analysis (Section 3.4). Furthermore, the requirements of other static analysis tools, such as information needed for model checkers and other safety analysis tools, may also be produced by the XRTJ-Compiler and can be stored in associated XAC files.

It can be observed that the XRTJ-Compiler may provide valuable information not only to achieve more precise and reliable results from analysis tools, but also to facilitate the implementation of various static analysis tools on the XRTJ infrastructure.

3.3 Program Safety Analysis

The inherent complexity in the verification of non-trivial software means that unsafe programs could be produced and used under critical situations. This is increasingly the case as today's programming models become more complex. Our Ravenscar-Java profile [23] has been developed with such concerns in mind, so that programs become easier to analyse, and the run-time platform will also be simpler to implement.

By program safety, we mean that a program will behave according to its functional (and temporal) specification, and not exhibit any erroneous actions throughout its lifetime. Erroneous actions include data races, deadlocks, and memory overflows. Also, in the context of real-time Java and the Ravenscar profile, we also need to ensure that the rules defined in the profile and RTSJ are observed. These rules are checked when programs are compiled and tested for conformance to the profile. This conformance test alone will remove many possible errors in the program. For example, deadlocks, and side effects in expressions can be prevented. The following subsections address some issues that are not directly addressed by the profile, but which are still important in validating the safety of a Java program.

[4] WCEF vectors represent execution-frequency information about basic blocks and more complex code structures that have been collapsed during the first part of the portable WCET analysis.

Verification of the Java Memory Model's effect As reported in [26, 29], the Java memory model (JMM) in [14] is a weaker model of execution than those supporting *sequential consistency*. It allows more behaviours than simple interleaving of the operations of the individual threads. Therefore, verification tools that simply examine Java source code or even bytecode are prone to producing false results [29]. Because the semantics of the JMM can lead to different implementations, some virtual machines may support sequential consistency, while others may not for performance reasons. This does not match the Java's *write once, run anywhere*[5] philosophy.

However, we can develop restricted fragments of Java programs for which the JMM guarantees sequential consistency (as opposed to the approach in [29]), given that there is a means to efficiently analyse Java bytecode to locate only necessary synchronizations. Libraries will still be considered because such an analysis tool will operate at the bytecode level. The point-to and escape analysis [8, 30] can be used to trace escaping and possibly shared objects, as well as improving overall performance by allocating non-escaping objects in the stack of a method. This approach, in fact, is how our analysis algorithm has been designed to uncover data races.

The underlying assumption of our algorithm is that any reads and writes on a shared object in a method must be enclosed within the same synchronized block (or method) in order not to have any data races. In other words, any syntactical gap between a read and write that are not covered by a single synchronized block will cause possible data races in a multithreaded environment because either a read or write action can be lost. This is true even when a shared object is indirectly read and updated using a local object. For example, an interleaving of another thread that may update the shared object can occur in between the indirect read and a (synchronized) write in the method, resulting in a lost write. Thus, any indirect reads and writes should also be treated in a similar manner to direct ones on a shared object.

Another similar case is the following: even when both a read and write are synchronized, there can still be data races if the two blocks are guarded by two different synchronized blocks and can be interleaved by other threads in between. Our algorithm is capable of analysing all such conditions, thus detecting problematic data races by tracing all shared objects and checking whether they are properly guarded by synchronized blocks or methods [21].

Memory Usage Analysis Shortage of memory space at run-time can be devastating in high integrity systems, but at the same time, oversupply of it will be costly. Considering the new memory areas introduced in the RTSJ, we may need a different means of estimating the worst-case memory space that a program requires at run-time, so that only the required amount of memory for each area will be allocated. For this purpose the RTSJ defines the `SizeEstimator` class, but the `getEstimate()` method does not return the actual amount of memory that an object of a class and its methods dynamically use, but simply the total

[5] Programs may still run anywhere, but possibly with different or unsafe behaviours.

size of the class's static fields. In this sense, the class is not readily usable in estimating the required memory size for an RTSJ application.

However, the Ravenscar-Java profile places some restrictions on the use of RTSJ's memory areas; for example, access to scoped memory areas must not be nested and such memory areas cannot be shared between `Schedulable` objects [23]. These restrictions greatly ease the development of an algorithm that will inspect each thread's logic to discover all classes it instantiates. After that, by making use of control and data flow information extracted from the code and the XAC file (such as loop bounds), the algorithm will be able to tell how many instances of each class are created by a thread. This information can then be used to produce a tight upper bound of the amount of memory that a thread utilises at run-time by applying `reserve()` and `getEstimate()` methods of the `SizeEstimator` class at the target platform before system despatching. This thread-oriented memory usage analysis algorithm is currently being developed.

Other Pre-runtime Analyses In addition to the ones introduced above, our static analyser (XRTJ-Analyser) is also intended to do the following analyses:

- Exception propagation analysis, and
- Dynamic memory access check analysis.

The first analysis stems from the fear that the propagation of any unchecked exceptions at run-time can be hazardous, while the latter is concerned with eliminating unpredictable runtime overheads caused by dynamic checks of the virtual machine. Memory access checks can be prevented by means of the point-to and escape analysis [8, 30], which will be integrated in our XRTJ analyser together with an efficient exception propagation analysis technique.

3.4 Timing Analysis

Timing analysis is crucial in real-time systems to guarantee that all hard real-time threads will meet their deadlines in line with the design. In order to ensure this, appropriate scheduling algorithms and schedulability analysis are required. Typically, most scheduling algorithms assume that the *Worst-Case Execution Time* (WCET) estimation of each thread has to be known prior to conducting the schedulability analysis. Therefore, estimating WCET bounds of real-time threads is of vital importance. In addition, having accurate timing estimations enables the developer to allocate resources more precisely to the system during the design phase.

On the whole, most WCET approaches [13, 35, 27] are tied to either a particular language or target architecture. Moreover, RTSJ has kept silent on how the WCET estimations can be carried out on the highly portable Java architecture. Consequently, it is unlikely to achieve Java's promise of "write once, run anywhere" or perhaps more appropriately for real-time "write once carefully, run anywhere conditionally" [5].

Hence, in order to offer a predictable and reliable environment for high-integrity real-time applications, a number of timing analysis issues need to be addressed, for example:

- How the WCET analysis can be carried out on a highly portable real-time Java architecture,
- How the run-time characteristics of Java, such as high frequency of method invoking and dynamic dispatching, can be addressed,
- How schedulability analysis can be conducted statically, and
- What techniques need to be provided to take account of the supporting distributed run-time environment.

The subsequent sections explore how these issues can be addressed in the static analysis environment of the XRTJ infrastructure to be able to ensure that real-time threads will meet their time constraints.

Portable WCET Analysis A portable WCET analysis approach based on the Java architecture has been proposed by Bernat et al. [4], and extended by Bate et al. [2] to address low-level analysis issues. This section presents how the portable WCET analysis can be adapted for our environment to be able to perform the WCET analysis statically [18].

The portable WCET analysis uses a three-step approach: high-level analysis (i.e. analysing the annotated Java class files and computing the portable WCET information in the form of Worst-Case Execution Frequency (WCEF) vectors [2, 4]), low-level analysis (i.e. producing a Virtual Machine Time Mode (VMTM) for the target platform by performing platform-dependent analysis on Java byte code instructions implemented for the particular platform), and conducting the combination of the high-level analysis with the low-level analysis to compute the actual WCET bound of the analysed code sections.

In our environment, the XRTJ-Compiler analyses the annotated Java programs and extracts the WCEF vectors during the compilation. The WCET vectors and WCET annotations are stored in the XAC file by the XRTJ-Compiler automatically. Therefore, after compilation, the class files and XAC files are ready for WCET analysis tools. To be able to build VMTMs of various platforms for real-time and embedded Java-based systems in an efficient way, we are developing a timing analysis benchmark that can build a VMTM of a target platform automatically simply by providing a native method that can access the machine cycle of the target platform. A WCET analysis tool in the XRTJ-Analyser, then, performs the combination of the high-level analysis with the low level VMTM to compute the actual WCET bound of the analysed code sections.

WCET Annotations Dynamic dispatching issues have been considered in compiler techniques for a number of years [1, 11, 12]. Unfortunately, these approaches cannot be directly applied to WCET analysis since they are solely optimising dynamic binding and do not guarantee that all dynamic binding will be

resolved before run-time. However, in WCET analysis for hard real-time systems, the execution time of every single method has to be known prior to executing it. Therefore, most approaches in the WCET analysis field have simply assumed that dynamic dispatching features should be prohibited. It is possible that these restrictions could make applications very limited and unrealistic because they might eliminate the major advantages of object-oriented programming [17].

In [17], we have explored the ways in which dynamic dispatching can be addressed in object-oriented hard real-time systems with the use of appropriate annotations. Our approach shows that allowing the use of dynamic dispatching can not only provide a more flexible way to develop object-oriented hard real-time applications, but it also does not necessarily result in unpredictable timing analysis. Moreover, it demonstrates how to achieve tighter and safer WCET estimations.

It is an open question for most annotation-based approaches as to how to verify if the provided annotations are correct. Combining optimisation techniques, such as Class Hierarchy Analysis (CHA) [11] or Rapid Type Analysis (RTA) [1], with our approach allows the annotations to be verified, if there is no dynamic linking at run-time. For example, applying the CHA approach, we can easily get the maximum bound of the class hierarchies information from the Java bytecode.

Schedulability Analysis This section demonstrates how schedulability can be carried out for our real-time Java architecture in line with the portable WCET analysis. In [18], we have illustrated how real-time parameters, including priority and dispatching parameters, for the set of threads and WCET estimates can be produced from the Java class files and XAC files. Given the WCET estimates and real-time parameters, the schedulability analysis can be conducted easily. In the XRTJ-Analyser, only the system configuration information is needed. Following the system configuration, the XRTJ-Analyser loads the scheduling algorithm and carries out the schedulability analysis. Scheduling algorithms must provide scheduling characteristics, algorithms which can calculate other scheduling parameters, such as release-jitter, blocking time, response-time, and resource access protocols which are provided to manage the priority inversion problems. The XRTJ-Analyser produces the result of the analysis of the system. The output file provides not only the result of the analysis, but also includes timing and scheduling information, such as response time, release-jitter, and blocking time.

Support for Distributed Features It should be noted that analysing the WCET bound of real-time threads in a distributed run-time environment differs from a standalone run-time environment. In particular, there are a number of issues that need to be clarified to achieve safe and tight WCET estimation and schedulability analysis of real-time threads containing remote method invocations. In the XRTJ infrastructure, we assume that one compatible virtual machine resides on each node in the cluster network and no recursive remote method invocations are allowed. In accordance with these assumptions, the WCET estimation and schedulability can be carried out as follows.

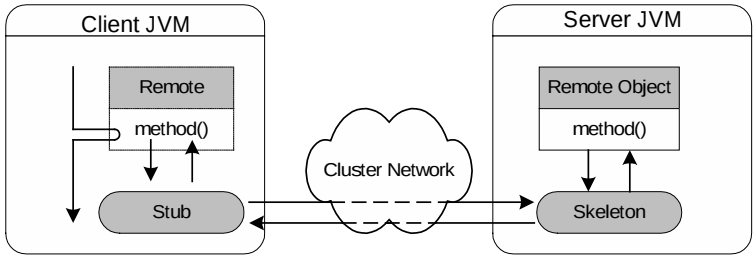


Fig. 4. The Java's RMI architecture [19]

Based on Java's RMI architecture shown in Figure 4, a *stub*[6] needs to be provided on the local virtual machine, whereas a *skeleton*[7] resides on the remote virtual machine [19]. In line with this architecture, holistic schedulability analysis can be performed [33, 25]; the response time estimations of all remote methods and the skeleton on the server node have to be analysed as sporadic threads during the schedulability analysis.

As to the client node, the WCET estimation of a real-time thread that holds remote method invocations differs from those that only comprise local method invocations. One should note that the WCET estimation of a remote method on the client node should not take into account the execution time of the remote method because a remote method is translated by the stub that resides on the local virtual machine and is executed on remote virtual machines. The WCET bound of a remote method invocation, therefore, should only take account of the execution time of the stub.

4 Distributed Run-Time Environment

This section is mainly concerned with the distributed run-time environment of the XRTJ infrastructure, which is targeted at cluster-based distributed high-integrity real-time systems. Moving from a centralised environment to a distributed environment requires the following issues to be addressed:

- How objects are allocated to nodes in the cluster,
- What form of communication is supported between distributed objects,
- How the model of communication can be integrated into Ravenscar-Java, and
- What impact the model has on the XRTJ environment.

For high-integrity environments, objects should be statically allocated to each node in the cluster. Therefore, the term *distributed* in this paper means

[6] A *stub* is a class that automatically translates remote method calls into network communication setup and parameter passing.

[7] A *skeleton* is a corresponding class that accepts these network connections and translates them into actual method calls on the actual object.

statically distributed whereby remote objects are allocated to nodes during the design phase. Although there have been many different communication models proposed for distributed Java programs (tuplespaces, distributed events, etc) most are based on top of the Java's RMI mechanism. XRTJ assumes the existence of a real-time RMI facility [36], such as that proposed by Miguel [10].

To accommodate existing practice, which is a stated goal of the project, two static distributed run-time environments are introduced, including *Initialisation Distributed Environment*, in which RMI is only allowed for use in the initialisation phase of an application, and *Mission Distributed Environment*, where a restricted real-time RMI model [36] can be used during the mission phase. The following subsections give further details on each of these and show how those issues mentioned previously can be addressed.

4.1 Initialisation Distributed Environment

The Ravenscar-Java profile does not support any remote interfaces on its main classes. Neither are they serialisable. Consequently, no remote operation can be applied to periodic threads or sporadic event handlers. This implies that they cannot be passed over the network during the mission phase of the RVM.

However, in order to provide not only high predictability and reliability, but also some degrees of support for distributed applications, which may reduce the development and maintenance costs of overall systems, the *initialisation distributed environment* is introduced. The motivation of providing this environment can be observed by a simple example given in Section 5. In such systems, communications between a server and each node, including loading data and reporting status, is essential and this can be achieved easily if the run-time environment provides distributed features in the initialisation phase.

In line with the framework proposed for integrating the RTSJ and Java's RMI [36], the standard RTSJ may offer a distributed environment with a minimal distribution level, defined as Level 0 integration by Wellings et. al. [36]. Following this approach, the initialisation distributed environment can be applied to either a standard Real-Time Java Virtual Machine (RTJVM) or a Ravenscar Virtual Machine (RVM). In such a run-time environment, both RTJVMs and RVMs can support a distributed environment defined as Level 0 distribution in [36] before all real-time threads are started (i.e. the initialisation phase of Ravenscar-Java).

In the mission phase of the RVM or after executing the real-time threads in a standard RTJVM, no remote method invocation is allowed. However, if the Ravenscar-Java profile supports aperiodic or non real-time threads, it is possible to use RMI in such threads with lower priority than real-time threads. Obviously, there is no modification required for standard RTJVMs or RVMs to support distributed high-integrity real-time Java-based applications in this environment.

4.2 Mission Distributed Environment

Supporting distributed features in the mission phase makes it necessary to address more issues, such as how to guarantee statically that all hard real-time threads will meet their deadlines, when distributed virtual machines can enter the mission phase and when real-time RMI can be used without rendering hard real-time tasks unsafe.

To offer a more flexible way to develop distributed high-integrity applications in the XRTJ environment without loss of predicability and dependability, the *mission distributed environment* is introduced. To support this distributed environment, three execution phases are proposed in the XRTJ-Virtual Machine (XRTJ-VM), including *initialisation* phase, *pre-mission* phase and *mission* phase.

In the mission distributed environment, all remote objects are allocated during the design phase and the XRTJ-VM supports Level 1 (i.e. real-time RMI) distribution defined by Wellings et. al. [36]. The program safety and timing analysis can be carried out with static analysis tools as mentioned in Section 3.4 during the static analysis phase. Note that the response time of all remote objects and threads, and the skeleton on the server node can be analysed as sporadic threads during the schedulability analysis, since they are allocated during the design phase.

The initialisation phase of the XRTJ-VM can be assumed to be the same as the initialisation the RVM mentioned previously. However, it should be noted that allocations, registrations, reference collections of all remote objects that are allowed for use in the mission phase have to be done during the initialisation phase.

Since the invocations of real-time RMI [36] are allowed in the mission phase of the XRTJ-VM, one should note that a virtual machine executing in its mission phase must not attempt to invoke a remote method on another virtual machine that is not running under the mission phase. The use of such invocations may result in unpredictable and unanalysable real-time threads running in the mission phase. To address this issue, synchronisation needs to be provided to decide when distributed virtual machines can enter into the mission phase at the same time. In line with the synchronization, all XRTJ-VMs in the same cluster network can be in the waiting stage after initialising. This phase is named the *pre-mission* phase of the XRTJ-VM.

The only difference between the mission phase of the RVM and the mission phase of the XRTJ-VM is that the invocations of pre-instantiated remote objects are allowed during the mission phase of XRTJ-VM. Furthermore, the XRTJ-VM supports the notion of real-time remote objects, real-time RMI, and simple distributed real-time threads [36] to enable the development of high-integrity real-time systems with greater flexibility.

5 Example

In this section, we present a simple example, which we hope is realistic enough to illustrate the application of our approach. Assume that there is an automated industrial production line where a number of multi-purpose robots and their controllers are employed. Each robot station (i.e. a robot and its controller) is linked over a network to the main server that will provide them with tailor-made instructions or tasks, depending on the models of products[8]. Once robot stations are set up with particular tasks, they will remain unchanged until new tasks are required to manufacture different products.

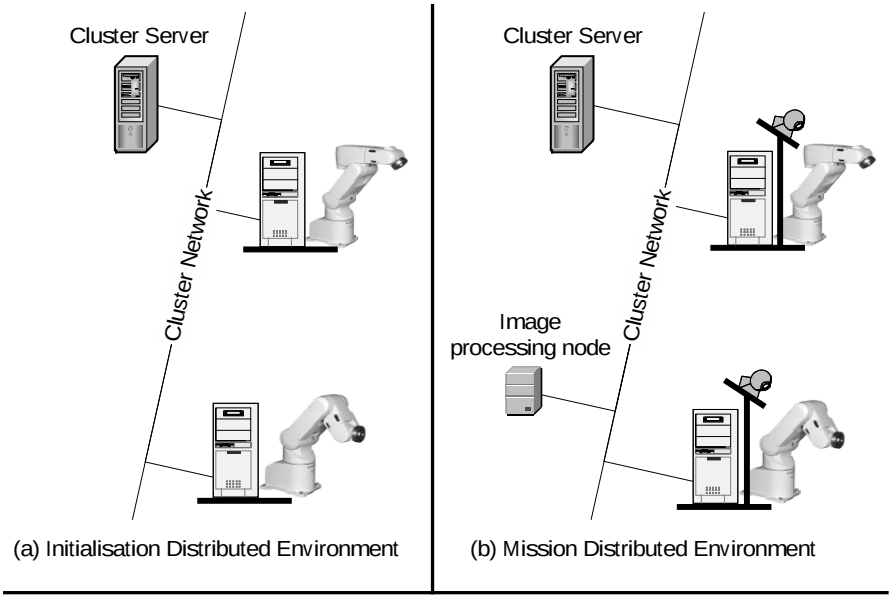
Our first distribution model, the Initialisation Distributed Environment described in Section 4.1, can be utilized in this situation, minimizing complexity in program analysis and in the implementation of underlying systems. In this manner, dependable software can be developed using our restricted programming model (i.e. the Ravenscar-Java profile), and static program safety and timing analysis techniques integrated in the XRTJ-Analyser. In the initialisation phase of all the robot stations, they will be given specific tasks by the main server by means of RMI. Having passed the initial phase, all the robots can begin their assigned operations, but are not allowed to invoke remote methods any more. A brief list of pseudo code for the robot controller is shown in Figure 5.

However, there are many other situations where robot controllers need to communicate with the server while in operation. For instance, a robot may inspect products using an overhead camera, send images to the server and require real-time feedback, assuming that the server has more powerful processors and resources to process images and distinguish faulty goods. In such cases, our second distribution model, the Mission Distributed Environment (see Section 4.2) is a valid approach. As with the code given in Figure 5, robot stations may invoke remote methods in the initialisation phase, as well as in the mission phase to cooperate with the server in a real-time manner as explained in Section 4.2. The pre-mission phase may be required to synchronize operations of the robots. However, in this more tolerant model of system distribution, static timing and schedulability analysis become more challenging, thus as we discussed briefly in Section 3.4 a holistic schedulability analysis should be performed to obtain response times of real-time threads communicating across a network.

6 Related Work

A consortium of European companies and research institutes have been working on a high-integrity distributed deterministic Java environment called HIDOORS [34]. The targeted applications of HIDOORS are similar to ours, but the project is mainly based on the Real-Time Core Extension specification [9], whereas our project is in line with the Real-Time Specification for Java [5]. However, there is a limited amount of information available on the HIDOORS project, and it is

[8] Robots need to be able to handle different models or versions of products manufactured in volume.




```
import ravenscar.*;
...
public class RobotController extends Initializer {
  public void run() {                          // Initialisation routine
    // Get Server's instructions/tasks via RMI
     ...
    // Set up real-time threads and sporadic event handlers
    // with appropriate parameters. For example,
    PeriodicThread robotRoutine1 = new PeriodicThread (
      new PriorityParameters(10),        // Priority:10
      new PeriodicParameters(
        new AbsoluteTime(0, 0),          // Start time
        new RelativeTime(5333, 0)        // Period
      ),
      new Runnable() {                   // Application logic
        public void run() {
          // Logic for the robot controller
          // Here, real-time RMI may be used in the mission distributed environment
           ...
          // Events may be fired
        };
      }
    );
    robotRoutine1.start();               // Start of Mission Phase!
  }

  public static void main (String [] args) {
    RobotController init = new RobotController();
    init.start();
  }
}
...
```

Fig. 5. An industrial automation environment

not clear how program safety analysis and timing analysis can be carried out in their preliminary report [34]. It should be noted that the HIDOORS project has attempted to provide a predictable implementation of the full Java langauge, whereas our project relies on the Ravenscar-Java profile.

Moreover, there has been considerable work in the area of formal verification of Java programs and bytecode, and Hartel and Moreau [15] systematically review most of this. Of particular interest to us are the verification techniques for Java Card applications based on the J2ME architecture [31], and Leroy [24], who recently developed an efficient on-card bytecode verifier. Leroy's approach is superior to other existing work in that it requires much less memory at run-time, and it handles additional features of the Java language (e.g. subroutines). Although our work does not directly deal with formal verification techniques at the moment, we feel encouraged by such developments, and may be able to incorporate them into our XRTJ-Analyser in the future.

7 Current Status

Currently we are modifying the Kopi Java compiler [20] to facilitate development of the XRTJ-Compiler. Our prototype XRTJ-Compiler can extract annotations from the source code and produces XAC files during compilation. The implementation of our prototype involved modifications to abstract syntax trees in order to map the annotation to the associated Java bytecodes. The prototype shows the feasibility of providing extra information that cannot be expressed in both Java programs and Java bytecode for static analysis tools. We are also working on the XRTJ-Compiler in order to provide a virtual machine timing model of a particular virtual machine automatically for the portable WCET analysis.

In addition, program safety and timing analysis tools are under development and will be integrated into the XRTJ-Analyser. The goal of the XRTJ-Analyser is to provide a user friendly graphic interface for the static analysis environment in future. We are also working on the reference implementation of RTSJ (RTSJ-RI), which is released by TimeSys [32], on Linux platform. A number of modifications will be conducted on the RTSJ-RI to be able to support mechanisms enforced both functionally and temporally correct results of applications in the distributed run-time system.

We have also created a website (`http://www.xrtj.org`) on which the most up-to-date information on this project can be found.

8 Conclusion and Future Work

In this paper, we have presented an overview of the XRTJ environment that is expected to facilitate the development of distributed high-integrity real-time systems based on Java technology. The three main aims of the XRTJ are to develop a predictable programming model, a sophisticated static analysis environment, and a reliable distributed run-time architecture.

Bearing these aims in mind, we have addressed several of the problematical features of the Java language, its run-time architecture, and the Real-Time Specification for Java. Our novel approaches include the Ravenscar-Java profile, program-safety and timing analysis techniques, and a distributed run-time environment. However, the profile may be supported by different architectures, and the analysis techniques are versatile enough to apply to other programming models. We have also raised some important issues in the area of distributed high-integrity systems, and presented how we can deal with them by defining two distributed run-time models, i.e. Initialisation Distributed Environment and Mission Distributed Environment, where safe and timely operations will be supported.

There are also some open issues, including design methodologies and tools; these should facilitate formal verification of systems at design stage. We intend to work towards these issues in the course of our implementation. We consequently feel confident that the XRTJ environment will provide a logical and practical base for future high-integrity real-time systems.

Acknowledgements

The authors would like to thank Dr. Guillem Bernat and Dr. Steve King for their contribution to many of the ideas expressed in this paper.

References

1. D. Bacon and P. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. *Proceedings of the ACM Conference on Obejct-oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, October 1996. San Jose, California.
2. I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. *In 6th IEEE Real-Time Computing Systems and Applications RTCSA-2000*, pages 39–48, December 2000.
3. S. Bentley. The Utilisation of the Java Language in Safety Critical System Development. MSc dissertation, Department of Computer Science, University of York, 1999.
4. G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. *In proc. 6th Euromicro conference on Real-Time Systems*, pages 81–88, June 2000.
5. G. Bollella, J. Gosling, B. M. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull. *Real-Time Specification for Java*. Addison Wesley, 2000.
6. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder- Second generation of a Java model checker. *In Proc. of Post-CAV Workshop on Advances in Verification*, 2000.
7. L. Carnahan and M. Ruark, (eds.). Requriements for Real-Time Extensions for the Java Platform. NIST Special publications 500-243, National Institute of Standard and Technology, http://www.nist.gov/rt-java, September 1999.
8. J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA*, pages 1–19, 1999.

9. J. Consortium. Real-Time Core Extensions for Java platform. *International J Consortium Specification*, Revision 1.0.14, September 2000. http://www.j-consortium.org/rtjwg/.
10. M. de Miguel. Solutions to Make Java-RMI Time Predictable. *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2001*, pages 379–386, 2001.
11. J. Dean, D. Grove, and C. Chambers. Optimisation of Object-Oriented programs using Static Class Hierarchy Analysis. *ECOOP' 95 Conference Proceedings*, Springer Verlag LNCS 952:77–101, 1995.
12. D. Detlefs and O. Agesen. Inlining of Virtual Methods. *ECOOP' 99 Conference Proceedings*, Springer Verlag LNCS 1628:258–277, 1999.
13. J. Engblom, A. Ermedahl, and P. Altenbernd. Facilitating Worst-Case Execution Times Analysis for Optimized Code. *In Proc. of the 10th Euromicro Real-Time Systems Workshop*, June 1998.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 2nd. edition, 2000.
15. P. H. Hartel and L. Moreau. Formalizing the Safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–588, 2001.
16. H. Hetcht, M. Hecht, and S. Graff. Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems. NUREG/CR- 6463, U.S. Nuclear Regulatory Commission, http://fermi.sohar.com/J1030/index.htm, 1997.
17. E. Y.-S. Hu, G. Bernat, and A. J. Wellings. Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 109–116, April 2002.
18. E. Y.-S. Hu, G. Bernat, and A. J. Wellings. A Static Timing Analysis Environment Using Java Architecture for Safety Critical Real-Time Systems. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 77–84, January 2002.
19. M. Hughes, M. Shoffner, and D. Hamner. *Java Network Programming*. Manning, 2nd. edition, October 1999.
20. Kopi. The Kopi Project. DMS Decision Management Systems Gmb. Hhttp://www.dms.at/kopi/.
21. J. Kwon, A. Wellings, and S. King. A Safe Mobile Code Representation and Runtime Architecture for High-Integrity Real-Time Java Programs. *Work-in-Progress proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 37–40, 2001.
22. J. Kwon, A. Wellings, and S. King. Assessment of the Java Programming Language for Use in High Integrity Systems. Technical Report YCS 341, Department of Computer Science, University of York, http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-341.pdf, 2002.
23. J. Kwon, A. Wellings, and S. King. Ravenscar-Java: A High Integrity Profile for Real-Time Java. *Proceedings of Java Grande-ISCOPE 2002*, pages 131–140, November 2002.
24. X. Leroy. On-Card Bytecode Verification for Java Card. *Springer-Verlag LNCS*, 2140:150–164, 2001.
25. J. C. Palencia and M. G. Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. *In Proc. of the 20st IEEE Real-Time Systems symposium (RTSS)*, pages 328–339, 1999.
26. W. Pugh. Fixing the Java Memory Model. *Proceedings of Java Grande Conference 1999*, pages 89–98, 1999.

27. P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.
28. P. Puschner and A. Wellings. A Profile for High-Integrity Real-Time Java Programs. *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2001*, pages 15–22, 2001.
29. A. Roychoudhury and T. Mitra. Specifying Multithreaded Java Semantics for Program Verification. *Proceedings of the International Conference on Softwre Engineering - ICSE*, pages 489–499, 2002.
30. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.
31. Sun Microsystems. Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices. White paper, Sun Microsystems, http://java.sun.com/j2me/docs/, 2002.
32. TimeSys. Real-Time Java. TimeSys. http://www.timesys.com/prodserv/java/.
33. K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.
34. J. Ventura, F. Siebert, A. Walter, and J. Hunt. HIDOORS - A high integrity distributed deterministic Java environment. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems WORDS-2002*, pages 113–118, January 2002.
35. A. Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. *In Proc. of the ACM SIGPLAN Wrokshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.
36. A. Wellings, R. Clark, D. Jensen, and D. Wells. A Framework for Integrating the Real-Time Specification for Java and Java's Remote Method Invocation. *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC-2002*, pages 13–22, April 2002.