# MAKING JAVA HARD REAL-TIME

*Peter Puschner*

Institut für Technische Informatik
Technische Universität Wien
A-1040 Wien, Austria
Email: peter@vmars.tuwien.ac.at

*Guillem Bernat and Andy Wellings*

Department of Computer Science
University of York
York, YO10 5DD, United Kingdom
Email: {bernat,andy}@cs.york.ac.uk

**ABSTRACT**

Due to its portability and security the Java programming language has become very popular. Standard Java is however not suited for programming hard real-time systems. To overcome this limitation we developed a hard real-time Java profile. This profile enhances standard Java with two important properties: temporal predictability and the portability of timing information. The paper introduces the profile and explains the steps of the portable worst-case timing analysis for the language.

## 1. INTRODUCTION

Only recently the Java programming language [7] has become very popular. This popularity is mainly due to the portability and security of the language. Nowadays, people do not only want to write Web and interface programs in Java, but also want to use the language in other application areas. One of these areas is real-time programming.

Real-time systems are characterized by the importance of time. Programs must not only be functionally correct, but must also deliver results in time. The consequence of missing a deadline depends on the type of application: In soft real-time systems the incorrect timing may impair service quality and cause annoyance (e.g., frames of a video transmission are not displayed in time). In hard real-time systems, in contrast, a single timing failure may lead to significant financial losses or even a catastrophy. Examples of hard real-time systems are the control systems for (nuclear) powerplants and fly/drive-by-wire applications.

This paper describes our endeavour to make Java, a non-real-time programming language, suitable for programming hard real-time systems. By doing so we make the strong features of Java – its security and its portability in the functional domain – available to real-time programming. Further, making this well-known language suitable for real-time programming will help to reduce training costs for real-time programming. Real-time programmers shall no longer have to learn the peculiarities of exotic real-time languages from scratch but shall be able to build on their knowledge of Java.

Making Java suitable for real-time programming is concerned with two essential properties: temporal predictability and portability of timing information. To achieve temporal predictability one needs to restrict the programming language (including its native classes) and to adapt the virtual machine. Extending the portability of Java from the functional domain into the time domain requires to enhance Java class files with portable information about code execution times and to develop concepts for portable worst-case execution-time (WCET) analysis.

The presented work was performed while the first author was on a research stay at the University of York. This stay was supported by the Marie-Curie Fellowship, contract no. HPMF-CT-1999-00184.

In this paper we describe a Java language profile for hard real-time systems. We characterize the restrictions and modifications of the programming language and discuss the necessary adaptions of the execution model of the virtual machine. Further we present a two-step WCET-analysis method that allows Java frameworks to include portable WCET information into Java class files.

Section 2 introduces the hard real-time Java profile and describes mechanisms of the virtual machine that supports the implementation of the profile. Section 3 gives an introduction to WCET analysis and Section 4 presents the two-step WCET analysis that supports the portability of timing information. Section 5 provides a summary.

## 2. THE HARD REAL-TIME JAVA PROFILE

The idea of making Java suitable for real-time applications is not new. In fact, at least two specifications of real-time dialects for Java exist today, [3, 8]. These dialects achieve real-time capabiliy by adding real-time features on top of standard Java. Naturally, these languages are very complex and include many features that are not suited for hard real-time applications.

In contrast to the mentioned approaches the profile presented here has been tailored for hard real-time systems, which need temporal predictability. The profile is based on the Real-Time Specification for Java (RTSJ) [3]. It does however restrict the threading model, thread interaction, and memory management to a very simple and temporally predictable subset of the RTSJ [12].

### 2.1. The Basic Model

As mentioned above, the construction of hard real-time systems requires that both the functional and the temporal behavior of applications are analyzable. This requirement restricts the concurrency of applications and thus influences the semantics that the profile has to provide. We have adopted the following real-time concurrency model that has been proposed in [4]:

- The number of threads, i.e., software activities that coexist and execute concurrently on the computer system, is fixed.
- Each thread has a single invocation event, but has a potentially unbounded number of invocations. There are two trigger types for activities, clocks and signals, the latter coming from the environment or other threads. Threads that are invoked by a clock are called time-triggered threads. Threads invoked by other signals are called event-triggered threads.
- Threads only interact via the use of shared data. Updates to the shared data must be atomic.

Given these assumptions and the knowledge of the scheduling model (e.g., fixed priority scheduling, time-triggered scheduling), the correctness of the behavior of an application can be analyzed (again, see [4]):

- The functional behavior of each thread is verified using techniques appropriate for sequential code. In this analysis, shared data is viewed as data input from or output to the environment. Timing analysis ensures that shared data is appropriately initialized. Worst-case execution time analysis computes bounds for the computation time of sequential code pieces of the threads.
- Given the threads have been assigned their temporal attributes (e.g., computation time, period, deadline) the system-wide timing behavior can be verified using standard techniques, e.g., fixed priority analysis [9, 5] or off-line scheduling [6].

## 2.2. Language-Model Restrictions

Java and the RTSJ comprise a number of threading features and mechanisms for inter-process communication with which higher-level abstractions can be constructed [15]. While these features make the language very powerful and provide valuable programming aids to the developer, they need to be supported by a complex run-time system and make timing analysis very ineffective, if not impossible. These features have thus been omitted from the profile. The following sections introduce the profile. Further details can be found in [12].

### *2.2.1. Threads*

In order to support the temporal requirements of hard real-time applications, the threading model has to be predictable in its functional behaviour, has to be analyzable with respect to its timing, and ideally has low overhead. We achieve this by the following mechanisms.

*Initialization and mission phase.* All applications are structured into two phases, an initialization (startup) phase and a mission phase. During startup an application performs non time-critical initializations to prepare for the real-time operation. It creates and initializes threads, event handlers, events, and memory objects and sets the timing and scheduling parameters (see below). Once all initializations have been completed the application switches to the mission phase. During the mission phase it performs the time-critical operations of the system.

All operations of the initialization phase are coded in `main`. Interference between `main` (i.e., the startup code) and the time-critical threads of the mission phase is avoided by running `main` as an environment thread at the highest thread priority. This way no thread can actually start before main has completed. The completion of the last statement of `main` and of all class initializations (see Section 2.3) marks the end of the initialization. This point also marks the starting point of the mission phase.

*Thread generation in* `main`*.* All threads are created and started during startup. This avoids the thread-creation overhead during the mission phase, makes the application timing predictable, and simplifies timing analysis.

*Static thread priorities.* In order to make timing analysis possible, the execution priorities assigned to threads remain unchanged during their entire life time (except when threads execute `synchronized` methods, [12]).

*Periodic time-triggered activities.* Periodic activities are realized as non-terminating threads which consist of initialization code and an infinite loop (main loop). After the initialization the threads enter the loop and never exit again until the system is shut down. Every path through the main loop has at least one call of the `waitForNextPeriod` method that delays the execution of the thread until the start of its next period.

Periodic real-time activities are implemented using the class `NoHeapRealtimeThread` of the RTSJ. The profile defines `NoHeapRealtimeThread` and its super-class `RealtimeThread` as follows:

```
public abstract class RealtimeThread extends
       java.lang.Thread implements Schedulable
{
  protected RealtimeThread(
            SchedulingParameters scheduling,
            ReleaseParameters release,
            MemoryParameters memory,
            MemoryArea area,
            ProcessingGroupParameters group,
            java.lang.Runnable logic)
            throws IllegalArgumentException;

  public void start();
  public void waitForNextPeriod();

  public static RealtimeThread
                currentRealtimeThread()
         throws ClassCastException;

  // other methods including:
  // getMemoryArea, getSchedulingParameters,
  // getMemoryParameters, getReleaseParameters
}

public class NoHeapRealtimeThread
       extends RealtimeThread
{
  public NoHeapRealtimeThread(
         SchedulingParameters scheduling,
         PeriodicParameters periodic,
         MemoryParameters memory, MemoryArea area,
         ProcessingGroupParameters group,
         java.lang.Runnable logic)
         throws IllegalArgumentException;
}
```

In the profile `RealtimeThread` is an abstract class.

It cannot be instantiated. All periodic real-time threads have to be implemented as instances of `NoHeapRealtimeThread` and do not allocate objects from the heap. This avoids garbage collection, which in turn improves temporal predictability.

The class `NoHeapRealtimeThread` provides only one constructor. This constructor enforces that the programmer defines all parameters required for the proper analysis and realization of periodic real-time threads. Further, the `RealtimeThread` class of the profile defines only a subset of the methods of the RTSJ `RealtimeThread` class. The `set...`-methods of the RTSJ that allow threads to change their own and other threads' timing or scheduling characteristics during the time-critical mission are not supported.

*Sporadic, event-triggered activities.* Non-periodic, sporadic activities are realized as event handlers. In the initialization phase these event handlers are linked to their triggering events. Once set up, the bindings between handlers and events remain unchanged. During the mission phase, each occurrence of an event triggers one execution of its handler.

*No dynamic class loading or initialization during mission phase.* Standard Java allows applications to locate and load classes dynamically. The profile, in contrast, requires that all classes are known before system start and that all classes are completely loaded and initialized during the initialization phase. Knowing all classes is mandatory in order to make sure all code is available for WCET analysis before the application is started. Loading and initializing all code during startup is necessary to obtain the temporal predictability of the code.

### 2.2.2. Concurrency

In a typical real-time application, threads communicate and share data with each other. Also, synchronization constraints exist between the actions performed by different threads. The profile provides mechanisms for access to shared data structures and for thread synchronization. The set of mechanisms is very restrictive to make the implementation of the run-time environment simple and avoid high run-time overheads.

*Synchronized methods.* Synchronized methods provide mutual exclusion to shared resources. Priority ceiling emulation adjusts thread priorities to avoid deadlocks.

*No* `wait`*,* `notify`*, or* `notifyAll`. Due to this restriction, no queues are required. This avoids complex queue management at runtime and avoids difficulties in the analysis of the order and timing of operations.

### 2.2.3. Memory Management and Raw Memory Access

The profile supports a very simple and restricted memory management. This is to facilitate an accurate pre-runtime analysis of the run-time behavior of the memory manager.

*No garbage collection (GC).* The temporal behavior of traditional garbage collectors for Java is unpredictable [14]. This is due to the fact that the points in time when the GC is run and the duration of each activation of the GC depend on the dynamic behavior of both the running Java programs and the virtual machine.

Specific real-time garbage collectors are more predictable [3, 8]. They are activated periodically and provide a guaranteed garbage-collection rate. These GCs in general manipulate complex data structures and the analysis of their characteristics is non-trivial.

Due to the difficulty to predict memory availability and the timing of garbage collectors, the profile does not support GC. The profile does, however, support two other types of memory, *Immortal Memory* and *Scoped Memory* as defined by the RTSJ.

*Immortal memory.* Objects allocated in immortal memory cannot be de-allocated or moved. They live until the system is shut down. The access time to the immortal memory must be known and the time needed for object allocation must be linear in the size of the object.

To avoid that the system runs out of memory, object creation in immortal memory is only allowed during the initialization phase.

*Linear-time scoped memory.* This memory area has a limited life time – the memory area is valid as long as one or more threads have access, i.e., a reference to it. When the last accessing thread removes its reference, finalizers for all objects in the memory area are run and the area is emptied.

The time for the allocation of an object from linear-time scoped memory must be linear in the size of the allocated object. The timing parameters of the memory must be known. The profile imposes restrictions on the allocation of objects in order to avoid a permanent growth of memory needs [12]. In addition, it restricts the use of references to and from scoped memory areas in order to maintain the safety of Java and prohibit dangling references, see [3].

*Object allocation to specific memory areas and raw-memory access.* The allocation scheme can be used to locate selected threads to fast memory in order to meet their timing constraint. Raw memory access allows programs to implement device drivers, memory-mapped I/O, etc.

### 2.2.4. Time and Clock

*Real-time clock and representation of time.* The profile includes class `clock` to represent real time and classes for storing and manipulating absolute points in time and durations. These classes are fundamental for hard real-time systems.

## 2.3. Implementation Issues

### 2.3.1. WCET Analyzability

To guarantee the temporal predictability of a complete processing system, an upper bound for the WCET of each thread instance (i.e., code executing between two calls of `waitForNextPeriod` for periodic threads and the code of event handlers) must be computable. This requires that all (Java and native) methods of the profile are coded such that their execution times can be bounded. The following are minimal requirements for WCET analyzability.

- Recursion must be bounded and loops must have a finite number of iterations. The bound for the maximum number of repetitions of every loop and recursion must be computable at compile time. Computing these bounds can be supported by the *WCETAn* class, as introduced in [2]. Using the *WCETAn* class is useful if control paths depend on input data and deriving loop bounds automatically is difficult.

- Code that executes during the mission phase must not load new classes (see Section 2.2.1).

#### 2.3.2. Virtual-Machine Support

The profile has to be supported by the target virtual machine and its environment. Otherwise real-time response and temporal predictabily are not achievable.

- The virtual machine must be able to manage the transition from the initialization phase to the mission phase and to handle the different requirements for these phases.

- The virtual machine and class loaders must load, link, and initialize all classes of an application during the initialization phase. Errors detected during these operations have to be reported before the initialization completes. This behavior is not in accordance with the Java virtual machine specification [10]. In hard real-time systems, however, error detection before the start of the mission phase is absolutely crucial. Unreported errors that occur during the mission phase might have catastrophic consequences.

- The virtual machine must provide a real-time scheduler to determine the order of thread execution that is necessary to meet all timing constraints.

- The virtual machine has to implement mutual exclusion during execution of synchronized methods.

- The virtual machine must provide a real-time clock and check the correct timing of events and thread execution (execution time, deadline).

- The execution times of all operations of the virtual machine itself have to be boundable.

#### 2.3.3. Tool Support for the Profile

The profile relies on a number of tools to ascertain the correct operation of an application. In fact, wherever possible the enforcement of the profile shall be checked before runtime. Again, although this does not conform to the traditional Java philosophy, it is crucial for hard real-time applications. The tool framework needs to include checks for the rules of the profile with respect to thread creation, memory allocation, and class loading, etc. It has to provide software for schedulability analysis or scheduling, and it has to include a tool for WCET analysis.

Within the following section we will focus on the WCET analysis tool for the profile. While existing WCET tools are customized to one specific hardware target the WCET tool for our Java profile supports the portability of WCET information to different platforms. It thus provides the necessary extension of the portability of Java from the functional domain into the time domain.

## 3. WCET ANALYSIS

Before we describe the portable WCET analysis for our profile we give a short introduction to WCET analysis. Informally, WCET analysis is defined as follows:

> WCET analysis computes upper bounds for the execution times of pieces of code for a given application, where the *execution time of a piece of code* is defined as the time it takes the processor to execute that piece of code.

Note the following points about this informal definition of WCET analysis that are worth mentioning:

- WCET analysis computes *upper bounds* for the WCET, not necessarily the exact WCET.

- The WCET bound computed for a piece of code is *application-dependent*. As the execution paths through the code may differ between different applications, the same piece of code may have different WCETs and thus WCET bounds.

- WCET analysis is *hardware-dependent*.

- Although the above definition does not mention the quality of the results of WCET analysis, WCET analysis is usually required to deliver bounds that are close to exact WCET values.

Current approaches to WCET analysis are complex and focus on specific hardware, see [11]. They are not portable and do not support a simple WCET computation from an intermediate execution-time representation.

The portable WCET analysis of the Java profile is based on the timing-schema approach [13]. The timing-schema approach uses execution-time calculation rules for different constructs (simple, if-then-else, loop, etc.) and construct sequences. It applies these rules recursively, following the syntactic structure of the code to compute WCET bounds, see Table 1.

Table 1: Simple Timing Schema

| $S$ | $t_{max}(S)$ |
|---|---|
| *simple* | WCET of S |
| S1; S2; | $t_{max}(\text{S1}) + t_{max}(\text{S2})$ |
| **if** (E) S1 **else** S2; | $t_{max}(\text{E}) + \max(t_{max}(\text{S1}), t_{max}(\text{S2}))$ |
| **while** (E) S; | $(N+1) \times t_{max}(\text{E}) + N \times t_{max}(\text{S})$ |

While the pure timing-schema approach is easy to use it has a severe problem. It cannot process information about the execution characteristics of a program that spans across the borders of nested constructs. For example, for the code listed in Figure 1 the timing schema approach includes the WCET bound of line 5 $N$ times within the inner loop and $N^2$ times in the whole program fragment shown. Obviously this yields a pessimistic WCET bound. The timing schema does not allow the user to express the number of executions of line 5 relative to the outer loop (i.e., across the borders of the inner loop) which would yield a WCET bound that accounts only $N(N+1)/2$ times for line 5.

```
1  for (i=1; i<=N; i++)        loop count: N
2  {
3    for (j=1; j<=i; j++)      loop bound: N
4    {
5      /* calc. */     executions in outer loop: (N+1)N/2
6    }
7  }
```

Figure 1: Piece of Code and Path Information

A way out of the above-mentioned problem is to add *path information* (i.e., information that characterizes execution paths across construct boundaries) to program code and use more complex methods for WCET analysis. While this works fine for a WCET analysis in which the path information and execution times of program parts are fully defined and for which the complexity of the WCET analysis is of minor concern, it is not suited for a WCET analysis of portable code. The latter analysis has to be kept simple, because one cannot expect that the means for a complex analysis are available at the location where the final WCET computation takes place.

## 4. WCET ANALYSIS FOR PORTABLE CODE

WCET Analysis for portable code differs from traditional WCET analysis in that not all necessary parameters for the analysis (e.g., the timing of instructions on the target hardware) are available at the software developer's site. Therefore the software developer cannot provide a concrete WCET bound with the delivered code. Instead, the developer provides an abstract representation of the execution time that can be easily evaluated when the missing parameters are known. The entire WCET analysis is therefore split into two parts, see Figure 2.

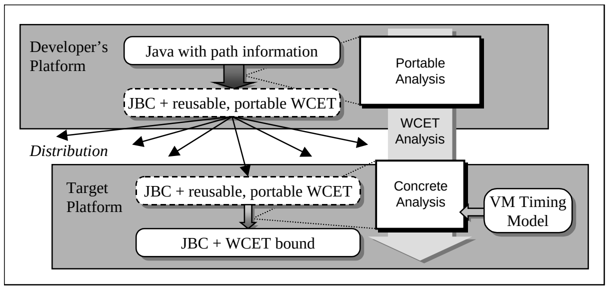

Figure 2: Two-Step WCET Analysis

The first step of the analysis is done at the developer's place. The software developer writes the code to be ported, annotates it with path information, translates it into a distribution format, and generates the abstract representation of the WCET of the code. The latter step is called *abstract WCET analysis*.

Together with the distributable code the abstract WCET information is delivered to the end users (either a human, e.g., a programmer who uses the code, or a computer system). The end user instantiates the abstract WCET information with the missing parameters and computes the final, concrete WCET bound. This step is called *concrete WCET analysis*. As mentioned before, the resources at the end user's site may be very limited and complex analysis tools may not available. The analysis must therefore allow for the concrete analysis to be simple.

### 4.1. Portability

It has been mentioned that portability makes it impossible that the software developer performs the complete WCET analysis. In the following we explain the effects of portability on WCET analysis in more detail.

*Portable code* is ported to and executed on different hardware platforms. For portable code the timing parameters (e.g., processor speed) of the target hardware are not exactly known to the software developer. Therefore, the duration of the actions the software performs is unknown. The WCET analysis of the software developer can therefore only provide *abstract durations* of actions (instructions) of the code. Again, the end user has to instantiate these abstract information to obtain concrete execution-time bounds.

While portability restricts the information about the execution time that can be represented, the static control structure of the code is not affected. The control structure of the code is, of course, a determining factor for the WCET of the code. Portable WCET information therefore consists of

- information about the static control structure of the code and
- abstract execution-time information for the actions/instructions of the code. As the abstract analysis does not have information about the timing of the target hardware, abstract timing information represents knowledge about the number of occurrences (execution frequencies) of certain facts instead of calculated times (see below).

### 4.2. Abstract WCET Analysis

The abstract-analysis tool generates the portable execution-time information. It first reads the portable code and generates a tree data structure that represents the control structure and the abstract timing of the operations of the analyzed program. The tool then traverses the tree structure and generates the abstract execution-time formula: All parts of the tree except scopes are translated into timing-schema expressions (e.g., an *if-then-else* yields an expression $c_1 + \max(c2, c3)$, where $c1$, $c2$, and $c3$ are representations of the abstract execution times of the condition, *then* and *else* branches, respectively).

#### 4.2.1. *Abstract Instruction-Execution Times*

The choice of the format for the abstract instruction-timing information is up to the designer of the class files and virtual machine. Depending on the quality demanded from the WCET analysis one can choose between a very simple and more sophisticated formats. In a simple format the portable execution-time representation for a piece of Java byte codes may be a vector that stores how often each of the byte codes occurs in the code. A more elaborate format may not just represent the number of occurrences of all byte codes but may also count the number of specific byte-code combinations, see [1].

### 4.3. Concrete WCET Analysis

The concrete WCET analysis instantiates the abstract values of the portable WCET information with concrete values and computes the WCET bound. This requires that the concrete analysis provides concrete values for the abstract information characterizing the timing of the basic operations on the target.

Once the concrete information is available the concrete analysis is simple. It replaces all abstract values in the WCET representation by the concrete values and evaluates the resulting formulas to obtain the concrete WCET bound. These formulas are built from the following, very simple operators: assignment, $($, $)$, $+$, $-$, $*$, $\min$, and $\max$.

## 5. SUMMARY AND CONCLUSION

The paper presented a real-time profile for Java that fulfils the needs of hard real-time applications. The profile restricts the real-time specification for Java to a subset which supports temporal predictability and the portability of worst-case execution-time information. To achieve temporal predictability the profile enforces restrictions on thread creation, thread interaction, memory management, class loading, and programming. The portability of WCET information is realized by a two-step WCET analysis. The first step of the analysis is performed by the software developer and generates an abstract execution-time representation that is distributed in Java class files together with the Java code. The second step of the analysis takes place in the user's environment or even on the target. This step instantiates the abstract timing information with the parameters of the target environment to obtain the final WCET value. To support the portability of timing information even to very simple targets the complexity is put into the first step of the analysis and the second step uses only very basic operations.

## 6. REFERENCES

[1] I. Bate, G. Bernat, G. Murphy, and P. Puschner. Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–46, Cheju Island, South Korea, Dec. 2000.

[2] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th Euromicro International Conference on Real-Time Systems*, pages 81–88, Stockholm, Sweden, June 2000.

[3] G. Bollela, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley, 2000.

[4] A. Burns, B. Dobbing, and G. Romanski. The Ravenscar Tasking Profile for High Integrity Real-Time Programs. In L. Asplund, editor, *Proceedings of Ada-Europe 98*, Lecture Notes in Computer Science, Volume 1411, pages 263–275, Berlin Heidelberg, Germany, 1998. Springer-Verlag.

[5] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, third edition, 2001.

[6] G. Fohler. *Flexibility in Statically Scheduled Real-Time Systems*. PhD thesis, Technische Universität Wien, Wien, Austria, 1994.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[8] J Consortium Inc. *International J Consortium Specification; Real-Time Core Extensions*, 2000. http://www.j-consortium.org.

[9] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Harbour. *A Practitioner's Handbook for Real-Time Analysis: A Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[11] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis (Guest Editorial). *Real-Time Systems*, 18(2/3):115–127, May 2000.

[12] P. Puschner and A. Wellings. A Profile for High-Integrity Real-Time Java Programs. In *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 15–22, Magdeburg, Germany, May 2001.

[13] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, SE-15(7):875–889, July 1989.

[14] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, second edition, 1999.

[15] A. Wellings and P. Puschner. Evaluating the Expressive Power of the Real-Time Specification for Java. *Real-Time Systems*, to appear.