

# The Komodo Project: Thread-based Event Handling Supported by a Multithreaded Java Microcontroller

J. Kreuzinger, R. Marston, Th. Ungerer  
Dept. of Computer Design and Fault Tolerance  
University of Karlsruhe  
D-76128 Karlsruhe, Germany  
{kreuzinger, ungerer}@informatik.uni-karlsruhe.de

U. Brinkschulte, C. Krakowski  
Dept. for Process Control, Automation and Robotics  
University of Karlsruhe  
D-76128 Karlsruhe, Germany  
{brinks, krakow}@informatik.uni-karlsruhe.de

## Abstract

*The Komodo project concerns the handling of multiple real-time events by Java threads that are supported by a multithreaded Java microcontroller. The architecture of the processor core and resulting implications are considered. The use of thread-based event handling is introduced and explained in combination with an Automatic Guided Vehicle (AGV) application. The advantages of thread-based event handling over a normal Interrupt Service Routine (ISR) strategy are demonstrated by the development of the AGV example.*

## 1 Introduction

At a first glance, the usability of Java for embedded systems appears unnatural, because Java's current popularity owes much to its widespread use on the internet. It may seem that using Java for embedded systems is simply an attempt to jump on the bandwagon, but this is not the case. Java was originally envisaged as a language for embedded systems, and its usage on the internet is a secondary success. The qualities that have made Java attractive for the WWW are also applicable to embedded systems. In particular, Java has the following advantages:

- Maintenance and reliability benefit from Java's object-oriented nature, which promotes the reuse of already well-tested components.

- Final code produced is compact - Java bytecode has an average instruction length of 1.8 bytes.
- Java bytecode can easily be distributed over networks, as can clearly be seen from Java's use on the internet.
- Java bytecode is platform-independent, reducing development times and costs as well as maintenance and upgrade costs.

Traditional implementations of Java require large amounts of system memory for operation. Owing to the slow performance of interpretative JVMs, it is often necessary to include a JIT (Just-In-Time) compiler, requiring more memory, with performance inferior to that of native code on the same processor. In embedded systems, weight, cost and power considerations are often of the utmost importance, so a requirement for additional memory or a faster processor counts against standard Java implementations. The use of a Java microcontroller can remove these drawbacks, as it runs Java natively, thus eliminating the JIT and providing good Java performance.

The instruction set of SUN's picoJava [1, 2, 3] is based upon the standard Java bytecode. It has been estimated that, for a normal application, 80% of Java bytecode instructions can be handled by picoJava's hardwired core. The remaining instructions must be handled by microcode or by trap routines.

picoJava requires additional instructions that do not exist in standard Java bytecode. Some of these instructions handle memory loads and stores, which are not provided in Java because of security reasons, but are required for the

picoJava to operate with external memory. Other picoJava specific instructions provide functions including cache access and diagnostics.

Embedded applications are often necessarily carried out in real-time and therefore impose strict conditions on the handling of interrupts and other events. The main content of this report is a study of a possible strategy for thread-based event handling in a multithreaded Java-based microcontroller. Consideration will be made of the suitability of Java for real-time computation, the system requirements and abstraction layers for a typical application in the field of Automatic Guided Vehicles (AGV).

The Komodo project comprises a full hardware, middleware and software solution to an AGV problem. In a Java context, this solution comprises the Java-based microcontroller (called Komodo microcontroller), which extends the picoJava core by multithreading using multiple stack register sets, program counters and instruction windows, and by a signal unit that triggers a thread when an interrupt occurs. picoJava's instruction set is enhanced by instructions to provide thread-management, priority-management and real-time support [4]. A JVM written in Java to function on said microcontroller, and a set of classes to implement standard Java (also to be written in Java) and to extend Java for real-time operation and memory access as required for the microcontroller's satisfactory operation is also needed.

A middleware layer, called OSA+, operates as the system platform between the Java API (Application Programming Interface) and the application.

The next section elaborates the real-time requirements for an embedded Java system. Section three shows priority schemes for thread-based event-handling and section four presents our Java microcontroller concepts. An open service-oriented architecture follows in section five and section six clarifies out ideas with an example.

## 2 Real-time requirements

Java is not specifically suited to real-time operation. Features required in real-time systems include good predictability, which can be summarised as a minimal best-case/worst-case interval and rapid, predictable event-handling. There are several features of the picoJava microcontroller that promote or inhibit predictability of real-time event handling. Some of these features are introduced as follows:

- *Garbage collection*

Garbage collection is a particularly sensitive issue in embedded systems, where space and time constraints are critical. It is necessary that a partial garbage collection will perform satisfactorily under foreseeable operating conditions. In non-real-time systems, it is

usual for garbage collection to be complete and non-interruptible. For real-time systems, it is required that the garbage collector be non-disruptive—that is, it can be incrementally called and run for a fixed time while ensuring that all further memory allocation requests will succeed. There exist several garbage collection techniques and algorithms for implementing a viable collector, but each has its own drawbacks. These can include unacceptable processor or memory overheads or low garbage collection efficiency.

In unconstrained systems, it can be appropriate to provide an adaptive or co-operative garbage collector that can assess the current application's attributes and apply the best-suited garbage collection method. However, the amount of ROM required for the implementation precludes the use of such garbage collectors in embedded systems. As embedded systems normally have a single, specific application for their entire lifetime, an alternative is to have a memory allocator that provides connectivity to one of several possible garbage collectors, with the application compiler or other software selecting the most efficient garbage collector from its libraries [5].

- *Caching*

Because caching can not provide 100% of all data requests, cache misses lead to delays in a non-predictable fashion. The requirement of highly-predictable response times renders memory caching unsuitable within a real-time environment. The lower memory requirements of Java bytecode compared to standard RISC code means Java microprocessors are better-suited to an uncached environment than a RISC equivalent. Caches are optional with picoJava, and would not be used in a hard real-time environment.

The development of caches for real-time systems is a research topic. For example, [6] outlines a strategy for increasing the determinism of cache-based systems through the use of preferred pre-emption points. Such research could lead to a Komodo microcontroller that includes a cache. On the other hand, the latency tolerance provided by the multithreading approach in the Komodo microcontroller may render caches unnecessary.

- *Dribbling*

The picoJava core normally implements a 'dribbling' mechanism to maintain the stack, preventing underflow and overflow conditions. It does this by observing stack usage levels, and pushing or popping data from the stack to cache or external memory as required. This functionality impinges on real-time operation through its low predictability. As such, the dribbler's use should be avoided, with responsibility for

avoiding stack errors passing to the compiler or programmer. The picoJava dribbler can be switched off in a control register, affording better predictability.

- *Event-handling*

It is a normal requirement of embedded systems to interact with input/output systems in real-time. This is an example of event-handling, where an external event causes the currently running software to be interrupted while the result of the external event is processed. The traditional manner of handling external events is with interrupts and interrupt service routines (ISRs). It is difficult to guarantee response times and to define priorities for multiple real-time events through the use of ISRs. The cause of these difficulties is that an ISR can be interrupted only by another interrupt of a higher priority level. This means that only top-priority (non-maskable) interrupts can have guaranteed response times. Our alternative event-handling methodology handles events by Interrupt Service Threads (ISTs) instead of ISRs. To handle multiple events simultaneously, this approach requires a multithreaded processor and a priority scheme (see below).

### 3 Prioritisation in thread-based event-handling techniques

Prioritisation is supported by the system platform and application code by virtue of an additional Java class which implements real-time threads. The priorities, or real-time conditions, could be assigned in several manners; the three real-time priority schemes envisaged for the Komodo microcontroller are as follows:

1. *Highest priority thread only.* The thread of highest priority runs using a determinable and specified amount of CPU cycles. Threads of lower priority are suspended.
2. *Highest priority thread preferred.* Similar to highest priority thread only, with the difference that lower-priority threads are able to exploit latencies in the highest-priority thread for execution.
3. *Percental priority thread.* A further alternative is to assign a percentage of processor cycles to which a process would be entitled, the share being assigned in proportion to the priority of the thread. It is then up to the programmer or supporting software to ensure that all processes complete within the available resource percentage, and that the total percentage is not greater than the total available resources, bearing in mind the system's own housekeeping services, for example, garbage collection. In this scheme, no

thread can be blocked completely, so worst-case response and data rates can be guaranteed even for concurrent events.

The first scheme can be implemented on standard microcontrollers such as picoJava. The second scheme requires rapid context-switching, and therefore multithreading capabilities in hardware, including multiple program counters and associated register sets. The third scheme necessitates additional hardware support for management of priorities, which is provided by the priority manager of the Komodo microcontroller.

## 4 Komodo microcontroller concepts

### 4.1 Overview

The Komodo microcontroller has been conceived to perform in real-time with a multithreaded capability. Its microarchitecture is based upon Sun's picoJava core with enhancements that comprise a signal unit, a priority manager, multithreading management, and an additional I/O unit to implement the new instructions and functionality. These additional elements have to be tightly-integrated, as they have some crossover in functionality. It is convenient, however, to consider them as separate entities for an initial description. A diagram of the internal structure is shown below (Fig. 1).

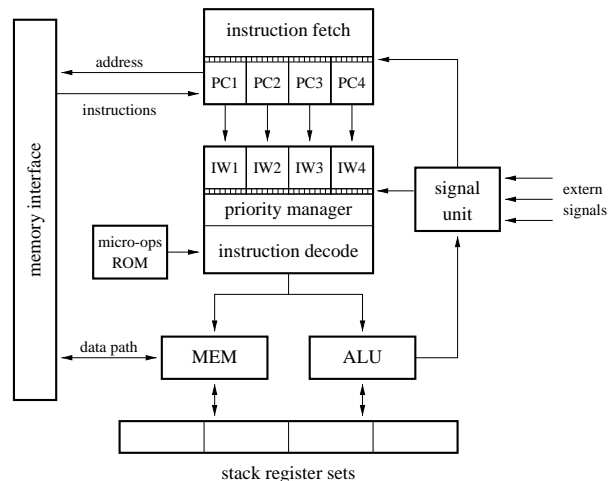


Figure 1. Block diagram of the Komodo microcontroller

### 4.2 Multithreading

The Komodo microcontroller features a multithreaded processor core. In general, a multithreaded processor provides several register sets and program counters on the

processor chip and allows for overlapped execution of instructions of different threads [7]. Our multithreaded Komodo microcontroller comprises four stack register sets in contrast to the single stack register set in the single-threaded picoJava. Each stack register set is attached to a different thread. This provides a rapid context-switch capability between several threads. For control purposes, each stack register set is associated with a status and control register as well as an individual program counter, labeled PC1 to PC4 in Fig. 1, giving the address of the next instruction to be carried out. The control register contains information about the thread: in essence, a priority level, and status information, such as whether the thread is currently active, and counters to control the percental priority scheme (see below) implementation. When the priority manager performs a context-switch, the next thread to be processed is selected. Each thread has its own instruction buffer, or IW (Instruction Window), which are labeled IW1 to IW4 in Fig. 1, these being necessary to prevent delays owing to the absence of caches. The appropriate instruction is fetched from the newly-selected thread's IW and loaded into the pipeline.

In order to achieve real-time processing, only one of the stack register sets is automatically maintained by the dribbler. This stack register set is not capable of real-time operation, but is provided for the operation of a non real-time main program, whereas all other stack register sets are provided for real-time threads.

### 4.3 Signal unit

The signal unit interfaces with external signal lines to determine when an event occurs. It then has to distinguish the priority of the event, and perform a service task as appropriate. It can achieve this by storing an event number and the appropriate IST to be activated when the event is recognised. The signal unit then informs the priority manager, which updates the status and control bits associated with the IW for the thread affected by the current event and any other affected IWs, which may, for example, include the currently-active thread.

### 4.4 Rapid context-switching

In the Komodo microcontroller, context-switching is controlled by the priority manager (see next section). Previous studies [8] have indicated that it is feasible to achieve very fast context-switching within appropriately-designed microprocessors. Throughout the pipeline, each opcode is forwarded along with its thread ID, so that each pipeline stage can contain an instruction from a different thread. The instruction fetch stage loads the IWs with instructions from memory. The next IW to be loaded at any time, and therefore, from which address the next memory read must

be made, is calculated from the fill-level of the individual IWs in combination with the thread status bits (active or suspended) and the priority assignments. The provision of multiple stack register sets and a pipeline able to contain instructions from different threads in its four stages allows context-switching to be very rapid, as there is no need to empty the pipeline and to save the current stack register set.

## 4.5 Priority manager

Priority values can be assigned in varying manners to account for differing application requirements. In the case of above mentioned percental priority scheme, the priority manager has information from each IW, comprising the priority, current status, number of cycles completed in the current time-period and the fill-level of the IW itself. The priority manager strategy decides which instruction will be processed next from the current state of the IWs and status bits according to the following conditions:

- Is the instruction immediately available in the IW of a thread, from highest-priority thread downwards.
- Does a branch or memory access potentially cause a latency in the pipeline, which another thread's instructions could fill.
- How many instructions from each IW (that is, each thread) have executed, and how many more are required to complete the thread processing - this information is stored in a counter in the IW.

From these control information and a mathematical interpretation of the priority strategy, the priority manager can decide from which IW the next instruction is dispatched to the instruction decode stage of the pipeline.

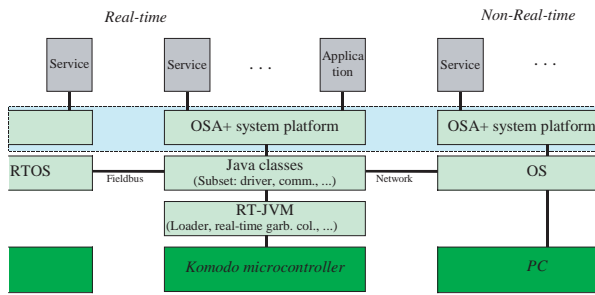
## 5 OSA+ (Open Service-oriented Architecture)

To simplify the development of distributed embedded solutions, a real-time middleware called OSA+ [9] is another part of the Komodo project. OSA+ is an open service-oriented architecture for the implementation of a distributed, scalable, real-time system platform. OSA+ provides access of services to those clients which require them, as well as necessarily allowing for the definition of the services themselves. This follows a normal client-server methodology.

As shown in Fig. 2, OSA+ allows the construction of distributed systems containing real-time and non-real-time parts. It connects the Komodo microcontroller with other heterogeneous controllers and processors. Therefore, Komodo based Java applications can be freely combined with

applications in other languages and on other operating systems. Within OSA+, each microprocessor or microcontroller system has its own local system platform. The local system platforms can communicate with each other over an appropriate communication media, providing a virtual global system platform. This has the following benefits:

- It provides flexibility to an application developer in terms of available resources (services) and applications.
- It is invisible both to services and applications whether or not they reside on the same or different machines. Run-time distribution changes are possible without change to any service or application.
- The level of abstraction provided by the system platform separates service access from the underlying hardware and operating systems, contributing to platform independence.
- Remote test and maintenance is supported.

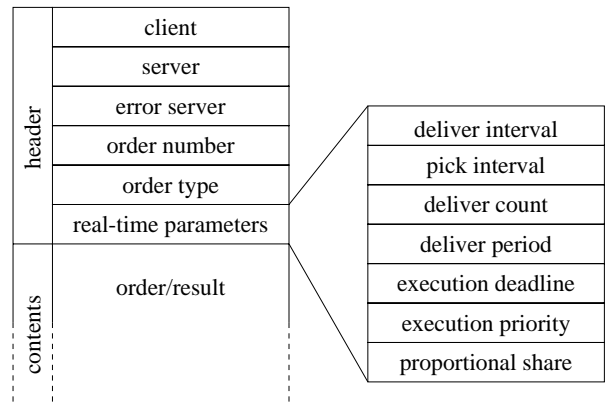


**Figure 2.** A sample Komodo OSA+ architecture

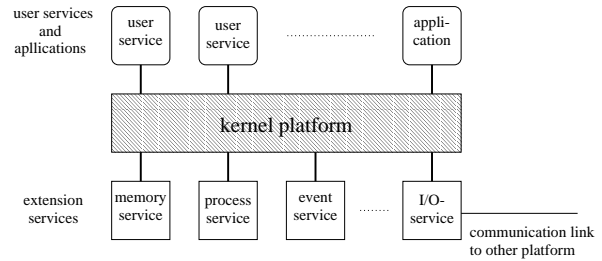
The service access is provided through the interchange of tasks and results. This synchronises communications between the services. There are no other communication or synchronisation methods necessary. All real-time constraints are defined as well by tasks. Therefore, tasks can include optional deliver modes to enable time restrictions to be observed. These deliver modes include information about message delivery time intervals, message pick-up intervals, real-time execution deadlines, priorities and percental CPU resources to be assigned to the task (Fig. 3).

The design concept of the system platform is similar to microkernel operating systems (Fig. 4).

It consist of a small kernel platform, which provides basic functionality, independent of hardware and operating and communication systems. The adaption to specific hardware and software platforms is done by extension services. There is no conceptional difference between user services and extension services. This concept provides the following benefits:



**Figure 3.** The OSA+ task structure



**Figure 4.** OSA+ kernel platform

- a high grade of scalability
- easy adaptability
- easy to extend and to improve
- easy to modify for research purpose

## 6 AGVs (Automatic Guided Vehicles)

This section describes an evaluation example for the Komodo project. AGVs are in increasing demand in production plants, where automation is steadily increasing [10]. It is clearly a benefit to provide low-cost solutions, and simple yet powerful embedded microcontrollers are ideal for fulfilling this goal. A typical AGV system comprises one or more vehicle units, which will often be fitted with a range of sensors; a set of external transponders and sensors; a central control unit with radio links to the vehicles; internal control units in each AGV, and a guide-path, which will be one of a physical, visible, electronic or magnetic path for the vehicles to follow. The task of the AGV's internal control unit is to ensure that the vehicle follows the correct path, meets performance specifications and avoids collisions with other vehicles and between the vehicle and an obstacle.

## 6.1 Desirable functionality for AGV controllers

As each AGV system will be different, owing to differing factory layouts and performance requirements, flexibility in the control system is important. Because, despite their differences, AGV systems have a great deal in common, for example, communication requirements between vehicles and central station, as well as general transponder types, there is a benefit to using modular, adaptable technologies [10]. Java scores well on both of these points, thanks to its object-oriented methodology. On a native Java processor, speed is also an asset, allowing the control of a rapidly-moving AGV.

OSA+ is also well-suited to the AGV environment, having been designed with AGVs in mind, and featuring good scalability, flexibility and modularity. The multithreaded event-handling functions of the Komodo microcontroller are beneficial to AGV systems, as the following example explains.

## 6.2 Consideration of an AGV system

A typical simple AGV system is shown in Fig. 5. Information from track-side transponders, detailing positional information for loading stops or an upcoming track junction, is sent to the vehicle. Further information comes from a scanning video camera, which picks up signals from a reflective guide tape. A laser scanner is used to detect obstructions on the track in front of the vehicle. A radio link transmits information to and from the vehicles. Each of these signal sources has certain time constraints: the track-side transponders only provide information for a finite time-period, which depends upon the speed of the vehicle. Missing a transponder signal causes the vehicle to be falsely positioned. An obstacle in the laser scanner's field of view must be processed in time to prevent collisions, and the available time will again depend upon the vehicle's velocity and also the obstacle's proximity. The tape detector camera provides a scanning view, which is only available for a finite and given time period, so its real-time conditions are the strictest. The radio communications are less critical, but must still be processed within a reasonable period.

If the controllers of the above system's vehicles handled events with normal ISRs, it is difficult to assign priorities in such a way that all the data sources could consistently be handled concurrently. For example, the scanning camera would be given the highest priority interrupt. If any other event, such as a transponder signal, were to be received during the operation of the camera's ISR, the transponder event would be delayed owing to its lower priority. This can cause data miss in the transponder's data stream.

In a similar system to that described above, but with the multithreaded event-handling (ISTs) described in this re-

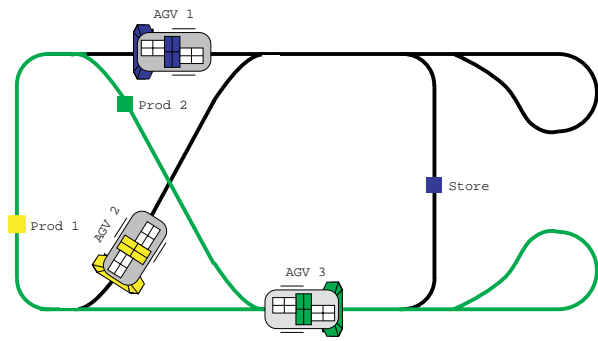


Figure 5. A simple AGV system

port in place of standard ISRs, the scanning camera would have an event number corresponding to the highest-priority thread. This thread would have a specified priority, based on the percentage of processor time it requires for completion. This would normally be a hard real-time condition, with appropriately severe consequences if the processor were unable to deliver enough cycles. The second-highest priority event is raised by the transponders, which could be assigned a lower priority based on the maximum allowable delay to respond to the event (a deadline), corresponding to a maximum tolerance in distance travelled before the vehicle's knowledge of having passed a transponder becomes critical. Knowledge of the processing time required for the transponder signal event to be handled can be used to give a necessary minimum percental priority. The laser scanner would be handled by a similar thread, whose priority would depend on the likelihood of there being an obstacle in the path of the vehicle, and the severity of any collisions (e.g. very slow moving vehicles are unlikely to have a high priority for collisions, unless the obstacles are foreseen as being heavy objects). The radio link might have a thread assigned which has a quite low priority, depending on the degree of autonomy enjoyed by the individual machines. However, it is acceptable that the radio link may have to retransmit any message as required, so it would be reasonable to specify a soft real-time priority of a low percentage of processor cycles, with the radio re-transmitting its request if it does not receive an acknowledge message within a specified time. With these different styles of priority assignments, the AGV designer and the AGV application software must decide how to assign the processor's resources to meet all the conditions.

Because the Komodo microcontroller has good predictability, it should not be overly difficult to achieve this, as the priority requests can all be reduced to an allocation of processor cycles within a specified program structure. Each thread will be allowed a fair access to the processor, and because the context-switch can take place very rapidly, the performance of the complete system should not suffer.

The main tasks of writing the real-time parts of an application are therefore reduced to a model-based analysis of priority assignments, and the programming of each thread so that all of them can complete within the available processor time.

## 7 Conclusions

The use of Java for embedded systems seems to be a sensible idea owing to its strengths in the rapid development of applications, ease of transmission and reusability of code. OSA+'s architecture is also highly appropriate for extended distributed systems owing to its virtual global system platform and ease of modification to other environments.

A multithreaded Java microcontroller seems to be a reasonable idea to combine the platform independence of Java bytecode with high speed execution and real-time support. It seems odd to choose Java as a language in which to write a JVM, but it is principally possible [11], and made easier by the extended Java bytecode of the Java microcontroller. As a method for event handling in AGV systems, thread-based systems provide a clear advantage over standard ISRs owing to their ability to apportion processor resources more equitably. This could result in lower-priced systems, able to use fewer, or lower-powered processors than at present, since exploiting latencies in the highest-priority thread leads to a reduction in the gross processor cycles requirement.

## References

- [1] Sun. PicoJava I Microprocessor Core Architecture *Sun Microsystems, Hardware and Networking Microelectronics Whitepapers WPR-0014-01*, 1997.
- [2] J. O'Conner and M. Tremblay. PicoJava I: The Java Virtual Machine in Hardware *IEEE Micro*, pages 45–53, March/April 1997.
- [3] H. McGhan and M. O'Conner. PicoJava: A Direct Execution Engine for Java Bytecode *IEEE Computer*, pages 22–30, October 1998.
- [4] U. Brinkschulte and T. Ungerer. Thread-basierte Ereignisbehandlung auf Java-Prozessoren in eingebetteten Systemen. *Workshop JAVA und Eingebettete Systeme*, Karlsruhe, pages 45–54, September 1998.
- [5] A. Petit-Bianco. Java Garbage Collection for Real-time Systems *Dr. Dobb's Journal*, October 1998.
- [6] K. A. Smith. Cache Memory Management in Real-time Systems *Computer and Systems Research Laboratory, University of Illinois Urbana-Champaign*, page <http://larcpubs.larc.nasa.gov/randt/1995/SectionB5.fm526.html>, 1995.
- [7] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond*. Springer-Verlag, Heidelberg, 1999.
- [8] J. Kreuzinger and T. Ungerer. Context Switching Techniques for Decoupled Multithreaded Processors *25th EUROMICRO Conference, Milano*, September 1999.
- [9] U. Brinkschulte and H. Vogelsang. A Distributed Scaleable Real-time Add-on for Operating Systems *RTAS '98, 4th IEEE Real-time Technology and Application Symposium, Work in Progress session, Denver*, 1998.
- [10] A. Hoff, H. Vogelsang, U. Brinkschulte, and O. Hammerschmidt. Simulation and Visualisation of Automated Guided Vehicle Systems in a Real Production Environment *ESS'97, 9th European Simulation Symposium and Exhibition - Simulation in Industry, Passau*, 1997.
- [11] A. Taivalsaari. Implementing a Java Virtual Machine in the Java Programming Language *Sun, SMLI TR-98-64*, March 1998.