

A Case Study on a Component-based System and its Configuration

Hiroo Ishikawa and Tatsuo Nakajima

Department of Information and Computer Science
Waseda University
3-4-1 Okubo, Shinjuku, Tokyo, 169-8555, Japan
{ishikawa,tatsuo}@dcl.info.waseda.ac.jp

Abstract. Ubiquitous computing proliferates complexity and heterogeneity of software. Component software provides better productivity and configurability by assembling software from several components. The purpose of this paper is to investigate system configurations on a component-based system and the side effects of the configurations. We have implemented a component-based Java virtual machine named Earl Gray, by modifying an existing Java virtual machine. The case study revealed several problems to use the current component framework when configuring software. We report three experiments of those problems and present a future direction to solve the problem.

1 Introduction

We can find many consumer electronic devices that contain computers. In ubiquitous computing environments, computers are embedded in various objects and environments, such as furniture, dishes, and cloths[11]. These embedded computers are networked and cooperated to provide various services satisfying a user's requirements. For example, a computer embedded into a chair cooperates with an air conditioning system in the same room and adjusts the room's temperature according to a person's preference or posture.

The vision of ubiquitous computing promises calm computing and integrated spaces between real world and virtual world. Consequently, system software such as operating systems and middleware are required to be more heterogeneous and complex, because various types of networked embedded systems will be available to build the integrated spaces.

Embedded systems are required various kinds of configurations according to requirements and resource constraints. Despite increasing the number of configurations, the current embedded systems are evolved by modifying their source code directly. This causes the reduction of maintainability and configurability of the systems.

For building such heterogeneous and complex software, a component-based approach provides better productivity and configurability by assembling software from several components. A component software allows embedded systems to be modified systematically. A component software can be modified by replacing the

component in it. So we don't manage the entire software, but each component of software. This advantage of component software encourages reusability and configurability of embedded systems.

This paper presents Earl Gray that is a component-based Java virtual machine. Earl Gray has been implemented by modifying an existing virtual machine. Since component-based design makes the system structure clearer than before, the system becomes more configurable. For instance, we can replace several components according to a platform's characteristics.

We also present three configurations on Earl Gray. Our experimental results show several problems to use the current component frameworks when configuring software. We report three case studies that show the problems and we present a direction to solve the problem.

The remainder of this paper is organized as follows. Section 2 describes the design and implementation of Earl Gray. In Section 3, we compare the performance of Earl Gray and that of the original JVM. In Section 4, three experiments are presented. They are (1) replacing the thread scheduler and associated components, (2) replacing the bytecode verifier component with one on a remote machine, and (3) extending Earl Gray to be able to handle the scoped memory functionality. The result and problems of each case are also described. In Section 5, we present related work, and we conclude the paper in Section 6.

2 A Component-based Java Virtual Machine: Earl Gray

We have developed a component-based Java virtual machine, named Earl Gray, by modifying the existing JVM, Wonka[15], and in a component description language, Knit[7]. On the process of decomposition of the system, we had several decision about component granularity, and component interface definitions. Despite describing in components, the performance of Earl Gray is changed little from Wonka. This section presents the off-the-shelf virtual machine and composition tool, and then describes the design and implementation of Earl Gray.

2.1 Off-The-Shelf System and Tool

Wonka Virtual Machine: Wonka is an open source virtual machine and supports the Java Virtual Machine specification provided by Sun Microsystems, Java 1.2 APIs with AWT, and several I/O devices such as RS232C ports.

Knit Component Description Language: We have adopted Knit to describe the components of Earl Gray. Knit is a component description language developed by the Flux research group at the University of Utah for describing components in OSKit[4].

A component in Knit consists of a set of typed input ports and output ports. The advantage of this model is that a connection between two components is explicitly described outside the components. Each port bundles some interfaces, and the interfaces are implemented by a set of functions written in C. The input

ports of a component specify the services that the component requires, while the output ports specify the services that the component will provide. An interface type consists of a set of methods, named constants, and the other interface types. A component in Knit is a black box component. The internal implementation of a component is hidden from clients.

There are two types of components in Knit as shown in Fig. 1 and 2. An atomic component is the smallest unit to compose programs, while a compound component consists of atomic components and/or other compound components. A system is structured by composing these two types of components.

```

bundletype Collector_T = {
    gc_collect,
    gc_create,
    ...
}

unit Collector = {
    imports [ heap : Memory_T ];
    exports [ gc : Collector_T ];
    depends { exports needs imports; };
    files { "src/heap/collector.c" }
}

```

Fig. 1. An example of an atomic component. `bundletype` defines an interface of a component in which function names in C are described. The `depends` block indicates dependencies between interfaces in `imports` and that in `exports`. The `files` block indicates an implementation of the component.

A components in Knit is a compile-time component. Components are statically combined into one executable binary after the compilation. Unlike CORBA and COM, component binding at run-time is not supported by Knit. The advantage of Knit is to keep the system small without communication overhead among components, discovery and binding mechanism. The compilation of Knit is executed in the following way: (1) Knit compiler checks syntax and dependencies between ports. (2) The compiler creates a rename table according to the `link` description in the compound components. (3) It compiles each component to a binary file by using `gcc`. (4) It renames entries in the symbol table in each object file according to the rename table created in phase (2). This is because Knit allows more than one components to be implemented the same interface. The compiler distinguishes components with the same interface by referring the renaming table. (5) The `ld` linker program links all object files into one executable program. The implementation of an atomic component in Knit is written in C or assembly languages. The atomic component consists of more than one C and/or assembly source files.

```

unit RuntimeMemoryArea = {
  imports [ thread : Thread_T,
            exception : Exception_T,
            ... ];
  exports [ gc : Collector_T,
            method : Method_T,
            ... ];
  link { [ method ] <- MethodArea <- [ thread, malloc, ... ];
        [ gc ] <- Heap <- [ exception, thread, malloc, ... ];
        [ malloc ] <- Malloc <- [];
  }
}

```

Fig. 2. A compound component example. A compound component includes the `link` block that explicitly connects atomic component and other compound components. `MethodArea` and `Heap` component is connected with `thread` interface from outside of `Collector` component. `Malloc` is an internal component of `RuntimeMemoryArea` component and it connects to `Heap` and `MethodArea` components.

2.2 Overall Architecture

Earl Gray is designed based on the original architecture as much as possible, because component software allows source code to explicitly reflect the image of the architecture. For instance, in Knit, connections among components are described in compound components. Those descriptions realize a clear architectural view in the source code.

Earl Gray consists of three large components, *Kernel*, *Middleware*, and *VM*, as shown in Fig.3. The kernel component provides low level services such as thread management and memory management. The middleware component provides common services for the VM component, such as string operations and network or serial port drivers. The VM component contains basic functionalities to implement Java virtual machine such as class loader, a runtime memory area, an execution engine, and native interfaces bridging to JavaAPI[10].

2.3 Component Granularity

We have implemented the functionality contained in each file as an atomic component. Wonka is a well-structured Java virtual machine. Each source file of Wonka usually contains one functionality. Since the component contains one functionality, each atomic component is usually small.

The granularity of compound components varies depending on their functionalities. For example, the native library component is the largest component of Earl Gray, because it includes many components implementing Java API. On the other hand, the Class Loader component contains only four atomic components.

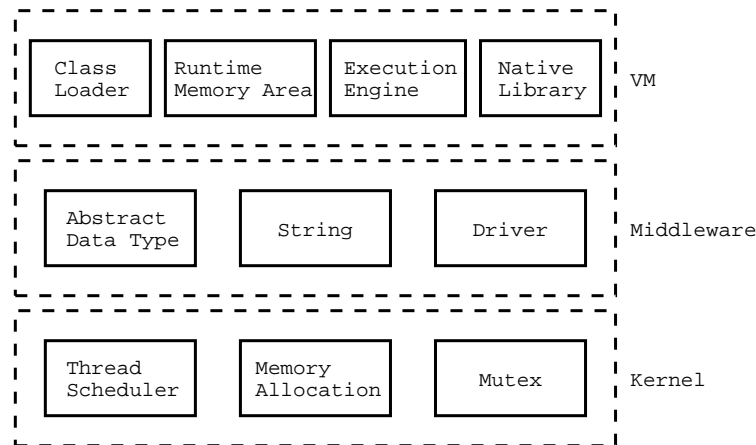


Fig. 3. Earl Gray Architecture

2.4 Component Interface

A component interface is a definition of an end point which other components connect to and communicate with. Port is an instance of the component interface. The number of links among ports depends on how many ports each component provides. The ports are classified into two types, input ports and output ports. Components are explicitly composed by connecting an input port and an output port by a connector. This approach makes the system architecture clearer than the original source code. For example, it is difficult to understand the relationship among the functions without examining all source code files of usual C programs. However, it is much easier to understand the relationship among components by examining component description files.

In our design, an atomic component offers only one interface to make an atomic component as simple as possible in order to clearly separate the roles of atomic components and compound components. If a component needs to offer two interfaces, we decompose the component into two atomic components, and create a compound component from the two atomic components. For example, `Runtime Memory Area` component consists of two atomic components, the heap component and the method area component. The heap and method area components provide `Collector.T` and `MethodArea.T` interfaces respectively.

2.5 Implementation

Earl Gray is a component-based Java virtual machine that is built by modifying the Wonka virtual machine, which is developed for embedded systems. Earl Gray supports Java Virtual Machine Specification provided by Sun Microsystems, Java 1.2 APIs, and several I/O devices such as RS232C ports. It does not support

JIT (just-in-time) compilation. The current version of Earl Gray runs on Linux for the Intel x86 family processor.

In the current implementation, the kernel component contains 16 atomic components and 1 compound component when the default scheduler is selected. The middleware component contains 25 atomic components and 3 compound components. Lastly, the VM component contains 108 atomic components and 8 compound components. All the atomic components are described in Knit and implemented in C.

3 Evaluation

This section compares Earl Gray with Wonka which is the original JVM of Earl Gray in terms of program size and performance. Despite using Knit, Earl Gray is as almost same size and performance as Wonka. Each JVM is compiled by gcc version 2.95 with `-O6` option without any debugging options, and doesn't include JIT compiler nor AWT support.

3.1 Program Size

The size of each JVM without symbols is almost same (Table 1). The component descriptions are dealt with in order to check the connections among components and rename the symbol tables. Thus, the descriptions are not compiled into the binary file. Earl Gray is 128byte bigger than Wonka. This is because Knit generates additional files in order to initialize and finalize the program.

Table 1. Size comparison between Earl Gray and Wonka

Program	Size (byte)
Earl Gray	567496
Wonka	567368

3.2 Performance

In order to measure the performance of Earl Gray, we have executed the Richards and DeltaBlue benchmarks[14] on Earl Gray and Wonka. The Richards is a medium-sized language benchmark that simulates the task dispatcher in the kernel of an operating system. The DeltaBlue is a constraint solver benchmark.

Table 2 shows the results of the benchmarks on Earl Gray and Wonka. All benchmarks were measured on a 1.2GHz Pentium 3 with 1024MB of RAM running Linux version 2.4.20. Earl Gray was compiled with gcc version 2.95.4 at

Table 2. Performance Evaluation (Execution Time)

Benchmarks	Wonka	Earl Gray
richards_gibbons	198ms	198ms
richards_gibbons_final	195ms	195ms
richards_gibbons_no_switch	231ms	231ms
richards_deutsch_no_acc	322ms	321ms
richards_deutsch_acc_final	700ms	697ms
richards_deutsch_acc_virtual	700ms	700ms
richards_deutsch_acc_interface	755ms	753ms
DeltaBlue	87ms	88ms

optimization level - O6. The results were reported by using the benchmark programs themselves. Therefore, they does not include any JVM initializations.

The performance of Earl Gray is as almost same as that of Wonka. Each result is the average of 100 times benchmarking. There are a few differences between Earl Gray and Wonka. This is because locations of functions in an executable file compiled by Knit are different from the one in the original executable file compiled by just gcc. In the Knit version, two set of functions in two atomic components respectively are placed closely in the executable file, if the components are compounded into one.

4 Case Studies on Component-based Configuration

This case study shows three configurations by replacing or adding components and the side effect of the configurations. In each case, we found problems of a component-based system. Although component interfaces indicate inter-component dependencies, there are other inter-component dependencies that component interfaces can not indicate explicitly. The case studies described in this section show the implicit inter-component dependencies appeared when configuring a system. We have examined the following three cases:

- 1. Replacing Thread Scheduler.** Replacing the default scheduler with a scheduler provided by a host operating system.
- 2. Modifying Bytecode Verifier.** Replacing the default bytecode verifier with a bytecode verifier executed in a remote machine.
- 3. Adding a Real-time Feature.** Adding scoped memory, that is one of the features described in the Real-time Specification for Java[1], to Earl Gray.

4.1 Scenario

In ubiquitous computing environments, a system is expected to be automatically modified and extended, because the software has to adapt to the current situation[2]. This case study is examined based on this kind of situation. In case

1 and 3, appropriate components are downloaded and activated according to the requirements. In case 2, due to the memory constraints, the system connects to a component on a remote machine instead of downloading it.

4.2 Using a Platform Functionality

This experiment aims to change a system to use alternative functionalities provided by a platform, instead of ones included originally. This change is realized by replacing components.

This experiment changes a scheduler component and investigates the effect of the change to the entire virtual machine. Because the thread scheduler is one of the core mechanisms of the Java virtual machine, the effect of the replacement must be examined.

Implementation: We replace the original thread scheduler with a scheduler that maps a thread in the virtual machine to a thread provided by the Linux kernel directly. The original thread scheduler's implementation includes a thread dispatcher mechanism and the threads are multiplexed on a single Linux thread. This replacement takes a scheduler mechanism away from Earl Gray, and the Linux kernel schedules the threads.

When implementing a new scheduler component, monitor and mutex components in the kernel component are also replaced because the kernel component includes monitor and mutex which are needed to synchronize threads.

As a result of direct mapping to the scheduler provided by the host operating system, the number of components in the kernel component was decreased. The components in the kernel component originally consist of 17 core components and 4 sub components. 8 components in 17 core components are used only inside of the kernel component. The 8 components contain mechanisms for thread management such as interrupt handling, timer, generating random number, and so on. The direct mapping implementation does not need these actual implementations. The remaining 9 components are still used when the new scheduler component is selected.

Since the kernel component is completely separated from other components, the new implementation does not affect other components in terms of explicit dependencies among components.

Implicit Dependency on Scheduling Policy: When a scheduler component is replaced, we found that the system was stopped unexpectedly. A race condition occurs in the function to uncompress a zip file where *push* and *pop* functions are invoked (Fig. 4). The functions were not considered that thread switch timing is different due to a different scheduling policy.

The original implementation assumes that the scheduler is not preemptive. Therefore, the queue structure in the uncompress component does not need to be protected from concurrent accesses while accessing it. However, Linux kernel threads are preemptive, thus we need to use mutex variables to protect the

queue. Moreover, adding critical sections requires the initialization of the mutex variables, and this requires to modify the initialization component.

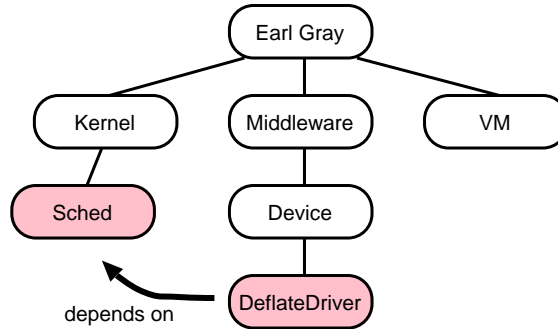


Fig. 4. Implicit dependency on the thread scheduler

4.3 Using a Remotely Available Resource

The second experiment changes a system to use components on a remote machine, instead of ones on the local machine. We investigated the difference between a local component and a remote component, and the effect of such a replacement. A bytecode verifier may invoke exceptions when it detects an invalid bytecode sequence. In the case of a remote bytecode verifier, the exceptions have to be invoked not only by invalid bytecode sequence, but also by network errors. The remote bytecode verifier requires a virtual machine to manage the exceptions raised by network errors in addition to the default exceptions.

Implementation: The remote bytecode verifier consists of two components, a stub component and a remote verifier component. Figure 5 depicts the verifier setting. The VM component requires a component providing the service with the `Verifier.T` interface. `Verifier` (local or stub) components provide the service with `Verifier.T` interface.

The stub component provides the same interface as the local bytecode verifier component. Therefore, the default verifier can be replaced by the remote bytecode verifier without modifying the other codes in the virtual machine.

The remote bytecode verifier communicates with the stub component by using the remote procedure call (RPC). We have adopted ORBit[13], which is one of the CORBA[12] implementations, as an RPC mechanism.

Implicit Component Behavior: Since the verifier component is located on a remote machine, we have to consider the effect of the network connection between

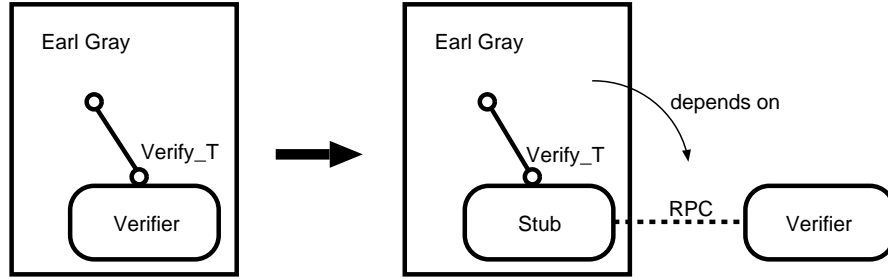


Fig. 5. (a)The verifier component is locally composed in Earl Gray. (b)The behavior of Earl Gray depends upon the network condition between the stub and the verifier component.

Earl Gray and the verifier component. The original verifier component is located on the local machine and composed with in Earl Gray statically, thus it returns the result immediately after finishing verification and the behavior of the verifier component is defined as verifying bytecode sequences. In the case of using the remote bytecode verifier component, however, it is unsure whether the result of verification is returned immediately after the verification. The behavior of the remote bytecode verifier component is not only defined as verifying bytecode sequences, but also the condition of the network connection.

In the case of this implementation, the virtual machine never expects that the remote verifier definitely returns errors. Instead, the virtual machine assumes that the verifier returns a result whenever it is invoked. In other words, components that invoke the bytecode verifier depend on whether a local or a remote bytecode verifier is used.

The `Verify_T` interface includes a function that creates `java.lang.verifyError`, which is thrown when the verifier detects the inconsistency of bytecode. Although network errors can occur in the case of the remote bytecode verifier, the interface does not include any functions that handle network errors. Thus, the system does not detect any network errors caused by the remote bytecode verifier.

4.4 Extending the System

The aim in the third experiment is to investigate the effect of a change when adding a new component. A component is a unit of deployment[8]. Thus, it will not be difficult to extend the virtual machine by adding a new component.

Implementation: The scoped memory feature which is one of the features described in Real-time Specification for Java[1], is implemented. The scoped memory enables an application to deallocate memory area explicitly when a program exits from the current scope. For example, if a method allocates a local (within the method) instance in the scoped memory area, the scoped memory

feature makes sure that the instance is deallocated when the method is returned. In other words, instances in the scoped memory area are never collected by the garbage collector, instead, applications need to manage memory allocation and deallocation explicitly.

The scoped memory feature is realized by two components. One is a scoped memory allocation component. This component has own memory area in order to allocate the scoped objects, while the default allocation mechanism instantiates objects on the heap and registers them to the garbage collector. The other component consists of several native interface components which are bridges between Java real-time APIs and the virtual machine.

Adding a new Functionality: The implementation of the scoped memory API requires the thread structure to be extended in order to include a pointer to a scoped memory area. Because the specification defines that a scoped memory area is created in a thread and destroyed when the thread is terminated.

Fortunately, the extension of the thread data structure did not affect the other components. However, the modification of a data structure might affect the implementation of the other components because the memory layout is changed if the data structure is modified. This causes a chain of the modifications of components.

Consequently, this case study shows that we have to still be careful to extend a component-based system with additional components. Because there is a chain of implicit dependencies among components.

4.5 Discussions

According to the above experiments, there are implicit dependencies among components even though components are well-separated. The experiments show how implicit dependencies are caused according to the behavior of components. Since component interfaces cannot represent the component behavior, another mechanism is required to specify the behavior. The last case study shows that architecture design is very important for evolving a component-based system.

The following sentences summarize the implicit dependencies described in those experiments.

1. A critical section in the decompressor component depends on the original scheduler, thus a race condition occurs with a new scheduler component. This dependency is completely implicit. That is to say, the dependency is not appeared until the system is built and runs.
2. The behavior of components that invokes bytecode verifiers depends on whether the bytecode verifier runs on local or remote. A stub component allows us to replace from a local one to a remote one in a simple way. However, the networked components have to be taken into account of response delay and exceptions due to the network faults.

3. The scoped memory manager component depends on several other components. A programmer needs to understand the internal of the components, and the side effects of it at the same time.

The result of the case study indicates the existence of behavioral dependencies among components, despite a component is generally defined as a unit of independent deployment. The number of software components will be increased much more, and the constraints for deploying components will become more rigid in ubiquitous computing environments. The behavioral dependencies have to be considered to build a component-based system in a correct way.

Component behavior must be taken into account when building component-based systems, since the behavior causes the inter-component dependency that may prevent the component-based system from configuring in a flexible way. Currently, we are designing a new component framework to allow us to specify implicit dependencies among components by representing the behavior of components explicitly like IOA[5] and CORAL[9].

5 Related Work

Jupiter is a modular and extensible JVM developed from scratch[3]. It focuses on scalability issues of the JVM for high-performance computing. The principle of design and implementation of modules make interfaces small and simple such that UNIX shells build complex command pipelines out of discrete programs. That principle facilitates to modification of JVM functionality. This principle is similar to our component design. Jupiter, however, doesn't address the dependency issues.

Knit has been adopted for building the current version of OSKit[4]. The components of OSKit are well-modularized. Moreover, design patterns are partially adopted for flexibility. Reid et al. mentioned that Knit declarations for OSKit components revealed many properties and interactions among the components that a programmer would not have been able to learn from the documentation alone[7]. This is the same as our observation that a component-based system contributes comprehensibility. OSKit, however, doesn't address the dependency issues except interface dependency processed by Knit.

Kon and Campbell[6] proposed the inter-component dependency management by the human readable descriptions and event propagation mechanisms based on CORBA. Hardware and software requirements are described in a file (e.g. machine type, native OS, minimum RAM size, CPU speed/share, file system, and window manager) with human readable descriptions. And inter-component dependency is managed by the event propagation mechanisms with (un)hook and (un)registerClient methods. However, these methods don't take into account of any component behaviors.

6 Conclusion

This paper has described a component-based Java virtual machine, Earl Gray, and has investigated on component dependencies through three case studies. The component description explicitly draws the connection among architectural components. Thus, it increases configurability of software.

The case studies have presented dependencies among components. The current component description ensures the consistency between interfaces, but doesn't maintain the behavior of components. In the first case, the problem was caused by the timing of thread scheduling. In the second case, the problem was caused by network errors. In the third case, the problem was caused because adding a new functionality requires to understand various components.

Currently, we are designing a new component framework that can specify the behavior of components, and we will use the component framework to build middleware infrastructures for ubiquitous computing.

References

1. Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
2. Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *HUMAN-COMPUTER INTERACTION*, vol.16, pp.99-166, Lawrence Erlbaum Associates, 2001.
3. Patrick Doyle and Tarek S. Abdelrahman. A Modular and Extensible JVM Infrastructure. In proceedings of the 2nd Java Virtual Machine Research and Technology Symposium 2002 (JVM'02), August 2002.
4. Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.
5. Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A Language for Specifying, Programming, and Validating Distributed Systems. MIT Laboratory for Computer Science, October 2001.
6. Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. *IEEE Concurrency*, 8(1):26-36, January-March 2002.
7. Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), October 2000.
8. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.
9. Vugranam C. Sreedhar. ACOEL on CORAL: A Component Requirement and Abstraction Language. In OOPSLA workshop on Specification of Component-Based Systems, October 2001.
10. Bill Venners. *Inside The Java 2 Virtual Machine*. MacGraw Hill, 2000.
11. Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(30), pp.94-104, 1991.
12. CORBA. <http://www.corba.org>
13. ORBit. <http://orbit-resource.sourceforge.net>

14 Hiroo Ishikawa and Tatsuo Nakajima

14. Mario Wolczko. Benchmarking Java with the Richards benchmark. http://research.sun.com/people/mario/java_benchmarking/richards/richards.html
15. Wonka - The Embedded VM from ACUNIA. <http://wonka.acunia.com>