# Construction and Reconfiguration of a Component-based Embedded JVM

**Hiroo Ishikawa**[†] **and Tatsuo Nakajima**[††],

Department of Computer Science, Waseda University, Tokyo, Japan

**Summary**
Component software is a method to deal with the complexity of software structures and configurations. Despite the increase of complexity in systems software such as operating systems and virtual machines, most of such software are configured under a traditional way i.e. C preprocessor directives. This paper explores a component software tool. We modularize an implementation of Java virtual machine with a component description language, called Knit. We show the difference between the original implementation and the modularized one. The modularization introduces little overhead despite the clearer view of its architecture. Three case study on reconfiguration of our JVM implementation show the benefits and problems of current component software technology.

***Key words:***
*Software components, component-based configuration, JVM, case study, Earl Gray, Wonka*

## 1. Introduction

The complexity of systems software such as language runtime systems, operating systems has been increasing. For example, an operating system includes a lot of device drivers, different types of scheduling policies, memory management strategies, file systems and network protocols, and so forth. Consequently, it is difficult for programmers to capture its structure for a short time. They have to look though source code and identify the dependencies in it to apply another configuration or extend the functionality of a system. This prevents a system from evolving over time. On the other hand, component software consists of software modules with well-defined interfaces, thus it is easier for programmers to configure or extend a system[1].

We explore the component software in the context of systems software. While the notion of component software has been applied to many kinds of middleware and applications, it is less studied in the underlying systems software such as operating systems.

The contribution of this paper consists of two folds. First, we show tradeoffs in the component software. While we divide an implementation of Java virtual machine (JVM) into components and rebuild a component-based JVM, we face on some tradeoffs in component software such as granularity of components and interfaces. Second, we conduct a case study under three different configurations of the JVM. The configurations are performed by replacing components with alternatives. Our experience is summarized as lessons learned.

We develop a component-based JVM, called Earl Gray, based on an open source JVM implementation for embedded systems, Wonka[2]. The components of Earl Gray are managed by a component description tool, Knit[3]. Using the component description language, the relationships between components and the structure of a system become clear so that a programmer can easily understand the relationships between functions of the JVM.

We conduct a case study on the reconfiguration of Earl Gray. The case study shows the implicit dependency between components, that is a kind of inter-component dependency that is not described in an interface of a component.

The paper is structured as follows. The next section describes the relate work to explore the context of our research. Section 3 gives an introduction to the Knit component description language, which we adopted to construct our component-based JVM, called Earl Gray. Section 4 shows the design and implementation of Earl Gray. Section 5 evaluates Earl Gray from the software engineering and performance viewpoints. In Section 6, we conduct a case study on reconfiguration of Earl Gray. We enumerate the problems of component-based configuration in practice through three types of configurations. The paper is concluded in Section 7.

## 2. Related Work

The idea of component software is adopted by several research-oriented systems software. They focus on the construction of component software or component models, while our research project addresses not only the construction issues, but also the configuration of a system in practice.

Manuscript received July 5, 2008.
Manuscript revised July 20, 2008.

Knit is adopted for building the latest version of OSKit[4]. The authors 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[3]. This is the same as our observation that a component-based system contributes the comprehensibility of a large system.

Jupiter is a modular and extensible Java virtual machine (JVM) developed from scratch[5]. 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.

Kon and Campbell[6] proposed the inter-component dependency management by the human readable descriptions and event propagation mechanisms based on CORBA[7]. 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.

## 3. Component Description

Earl Gray components are defined by Knit component description language, which is developed by the Flux research group at University of Utah[3]. A component in Knit is a source level component. The Knit compiler parses component descriptions and generates a set of Makefiles for building the source code, which is written in a programming language such as C. This section introduces Knit for readers to understand the following sections.

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 implementation of a component is hidden from other components.

There are two types of components in Knit as shown in Figure 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 atomic components. `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.

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

Fig 2. An example of compound components. 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.

A components in Knit is a compile-time component. Components are statically combined into one executable binary after the compilation. Unlike CORBA or COM[8], 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 (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 may consist of more than one source files written in C or assembly language.

## 4. Design and Implemenation

We have developed a component-based Java virtual machine, named Earl Gray, based on an open source JVM implementation, Wonka. Wonka is an open source Java virtual machine implementation for embedded systems. It 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. Earl Gray components are written in a component description language, Knit. During the process of decomposition of the system, we had to make several decisions on component granularity, and component interface definitions.

### 4.1 Architecture Overview

Earl Gray is divided into three compound components, `Kernel`, `Middleware`, and `VM`, as shown in Figure 3. The kernel and the middleware components provide interfaces described in Knit to their upper layers. 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 and serial port drivers. The VM component contains the basic functionality to implement Java virtual machine such as class loader, a runtime memory area, an execution engine, and native interfaces bridging to JavaAPI[9].

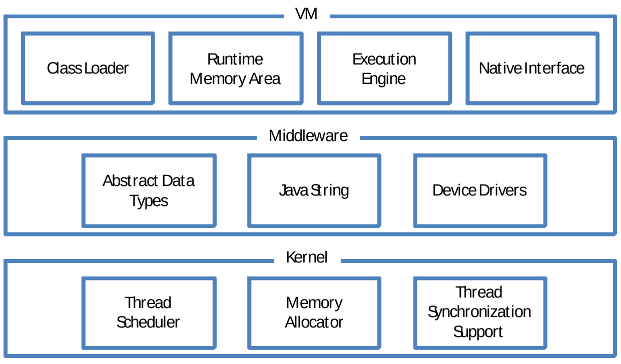

Fig 3. The architecture of Earl Gray

### 4.2 Component Granularity

The flexibility of the configuration of component software relies on the granularity of a component because the configuration is done by replacing, adding, or removing components. To translate Wonka into Earl Gray components, we first identified its functionality so that each atomic component provides one functionality. Wonka is fortunately a well-structured implementation and we followed the structure at the design level.

The granularity of compound components varies depending on their functionality. 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.

### 4.3 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.

### 4.4 Implementation

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 processors.
Knit allows an atomic component to be implemented in more than one files, but we defined an atomic component consists of only one file to avoid a dependency problem within an atomic component. Most of its implementation files (i.e. .c files) are individually wrapped by an atomic component.

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.

## 5. 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.

### 5.1 Program Size

The sizes of each JVM are very close (Table 1). We stripped out the symbol information of them. 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 | 567 368 |

### 5.2 Performance

The performance of Earl Gray is measured by the Richards and DeltaBlue benchmarks[10]. The Richards is a medium-sized language benchmark that simulates the task dispatcher of an operating system kernel. The DeltaBlue is a constraint solver benchmark. We compare the result of the benchmarks on EarlGray to that of Wonka.

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 |

Table 1 shows the results of the benchmarks on Earl Gray and Wonka. All benchmarks were measured on a 1.2GHz Pentium III with 1024MB of RAM running Linux version 2.4.20. The result is reported by using the benchmark programs themselves. Therefore, the time for JVM initialization is excluded from the results. Each result is the average of 100 times benchmarking.

There are a few differences between Earl Gray and Wonka. This is because the relocation policy of Knit is different from the normal build process. The atomic components in the same compound component are placed closely in the executable file, which is not the case for the normal build process.

## 6. Case Study

The case study shows Earl Gray under three different configurations. The configurations are done at component-level. In other words, when we add a new functionality to Earl Gray, we don't extend the existing component, but we developed an alternative component. The case study shows the side effect of the configurations. We expect the system consistency is kept as far as we satisfy the dependency at the component interface level. However, we found another kind of inter-component dependency, that we call implicit dependency.

In this section, we examine the following three configurations:

*Using a platform functionality*: Replacing the default scheduler involved in Earl Gray with a scheduler provided by a host operating system.

*Using a remotely available resource*: Replacing the default bytecode verifier with a bytecode verifier executed in a remote machine.

*Adding a new feature*: Adding scoped memory, that is one of the features described in the *Real-time Specification for Java*[11], to Earl Gray.

## 6.1 Using a Platform Functionality

This experiments aims at changing a system to use alternative functionality provided by its underlying 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.

### 6.1.1 Implementation

We replaced the thread scheduler component in the kernel component. Similar to a user-level thread library, the original thread scheduler of Earl Gray includes a thread dispatcher mechanism that runs as a single Linux thread. The alternative thread scheduler does not include a thread dispatcher mechanism, but maps each Java thread to a *pthread*. This change takes a scheduler mechanism away from Earl Gray, and the Linux kernel schedules the Java threads on behalf of it. The monitor and mutex components in the thread synchronization support component are also replaced.

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 from 21 to 13. The number of components of the kernel component are originally 17 core components and 4 sub components. The 8 components out of 17 core components are used only inside of the kernel component. They contain mechanisms for thread management such as interrupt handling, timer, generating random number, and so on. The direct mapping implementation does not need these 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 from the architectural point of view.

### 6.1.2 Implicit Dependency on the Scheduling Policy

We observed that Earl Gray with the alternative thread scheduler component was terminated unexpectedly. A race condition occurred in a component for ZIP file decompression (JAR file format is a superset of ZIP file format). The component called `Deflate Driver` didn't acquire a mutex to access to the fraction of a JAR file, which was supposed to be a critical region.

The original implementation assumes that the scheduler is not preemptive. Therefore, the queue structure in the Deflate Driver 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.

## 6.2 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, assuming load balancing. We investigate on the difference between a local component and a remote component, and the effect of the replacement.

A bytecode verifier is replaced for the experiment. A bytecode verifier generates an exception 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.

### 6.2.1 Implementation

The remote bytecode verifier consists of two components, a stub component and a remote verifier component. The VM component requires a component providing the service with the `Verifier_T` interface. The original verifier component, which runs on local, implements the `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 components.

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

### 6.2.2 Implicit Component Behavior

Since the alternative verifier component is located on a remote machine, we have to consider the effect of the network connection between Earl Gray and the verifier component. The original verifier component is composed within 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 problem here is that it is impossible for the current component interfaces to restrict the implementation of them. The original components cannot handle the remote extension properly. The `Verify_T` interface includes a function that generates an instance of `java.lang.VerifyError`, which is thrown when the verifier detects the inconsistency of bytecode. The interface does not include any other functions that handle network errors. Thus, the system cannot detect any network errors related to the remote bytecode verifier with the original interface.

## 6.3 Adding a New Feature

We examine how to extend a component-based system here. Comparing to replacing components, extending the structure may require changes in the existing interfaces. We investigate on the effect of adding a new component through this subsection.

### 6.3.1 Implementation

The scoped memory support is added to Earl Gray. The scoped memory is one of the features described in Real-time Specification for Java[11]. 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 implemented in two components. One is a scoped memory allocator component. This component has its own memory area in order to allocate scoped objects, while the default allocator instantiates objects on the heap and then registers them to the garbage collector. The other component provides the native interface for Java real-time APIs.

### 6.3.2 Extending the Core of the System

The implementation of the scoped memory API requires the thread structure to be extended to store a pointer to the scoped memory area. The specification defines that a scoped memory area is bound to a thread and destroyed when the thread is terminated.

The extension of the thread data structure did not affect to 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.

## 6.4 Discussions

The series of the experiments have shown the implicit dependencies among components due to the limitation of the current component interface descriptions; The current component interfaces cannot represent the component behavior. Consequently another method is required to specified the behavior.

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

- 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.

- 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.

- 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 the extension 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 stricter. The behavioral dependencies have to be considered to build a component-based system in a correct way.

Component behavior should 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 such as IOA[13], CORAL[14] and Sing#[15].

## 7. Conclusion

Component software is an important notion to deal with the complexity of a system. It is expected to enhance easy reconfigurations, component reuse, etc. This papers explores component software in the context of systems software such as language runtime and operating systems, by implementing and reconfiguring a component-based Java virtual machine, called Earl Gray. While a component description we used provided us a comprehensive view of the entire structure of JVM without any serious overhead of size and performance, we encountered the implicit inter-component dependencies which cause unexpected behavior of a system. This is because the current component description focuses on the interfaces of components. Our experiments have shown that the behavioral description of a component is necessary.

## References

[1] Clemens Szyperski, Dominik Gruntz, and Stephan Murar. Component Software: Beyond Object-Oriented Programming, 2nd ed. Addison-Wesley, 2002.

[2] Wonka – The Embedded VM from ACUNIA.

[3] 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.

[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] 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.

[6] Fabio Kon and Roy H. Campbell. Dependence Management in Component-Based Distributed Systems. IEEE Concurrency, 8(1):26-36, January-March 2002.

[7] CORBA. http://www.corba.org

[8] Don Box. Essential COM. Addison-Wesley, December 1997.

[9] Bill Venners. Inside The Java 2 Virtual Machine. MacGraw Hill, 2000.

[10] Mario Wolczko. Benchmarking Java with the Richards benchmark. http://research.sun.com/people/mario/java_benchmarking/richards/richards.html

[11] Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. The Real-Time Specification for Java. Addison-Wesley, 2000.

[12] ORBit. http://orbit-resource.sourceforge.net

[13] 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.

[14] Vugranam C. Sreedhar. ACOEL on CORAL: A Component Requirement and Abstraction Language. In OOPSLA workshop on Specification of Component-Based Systems, October 2001.

[15] Galen C. Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Microsoft Research Technical Report MSR-TR-2005-135, Microsoft Corporation, October 2005.

**Hiroo Ishikawa** received the B.S. and M.S. degrees in Computer Science from Waseda University in 2001 and 2003, respectively. He have developed a reliable operating system research at Waseda University since 2004.

**Tatsuo Nakajima** is a professor of Department of Computer Science and Engineering in Waseda University. His research interests are embedded systems , operating systems and ubiquitous computing.