EarlGray: A Component-Based Java Virtual Machine for Embedded Systems

Hiroo Ishikawa, Tatsuo Nakajima
Department of Computer Science
Waseda University
{ishikawa,tatsuo}@dcl.info.waseda.ac.jp

ABSTRACT

EarlGray is a component-based Java virtual machine (JVM) that can be configured to satisfy various kinds of requirements for building future information appliances and embedded systems. While the modification and extension on an existing JVM tend to be done in an ad-hoc manner, EarlGray allows developers to customize the JVM in a systematic manner by decomposing it into components and the explicit descriptions of the relationship between the components. We also examine three case studies on the customization of the EarlGray: exchanging a scheduler, memory management, and class file verifier components. These case studies shows the benefits and drawbacks of the current component-based technologies.

1. INTRODUCTION

The Java programming language becomes popular in embedded systems because it provides useful programming abstractions such as object-orientation, multi-threading, and exception handling. Since applications for embedded systems become more and more complex, the chance to adopt an object-oriented programming approach will increase. In addition, advanced embedded platforms are recently developed with multi-core processors. Concurrent programming supports are crucial for application development environments for future embedded systems. The abstractions provided by Java are useful for developing software of complicated embedded systems.

The interest to a customized Java virtual machine(JVM)[8] is increasing due to a variety of requirements of embedded systems. The embedded systems need to satisfy resource constraints such as the size of memory, input/output peripherals, CPU power, and timing constraints. Thus, software systems on an embedded system should be customized to satisfy the constraints. Sun microsystems proposes the Java environment specifications for embedded systems[2, 3], where JVM is customized to each platform by system developers. However, in spite of the requirements for customizing JVM, most of them provide only compile time option parameters for their customization. Thus, a developer may have to modify the source code of a JVM to satisfy the requirements in an ad-hoc way. In terms of the program quality, the modification or customization should be done in a systematic way.

Modularization of a system is a basic technique for developing complex software. The software modules can be developed by different programmers. Moreover, in embedded systems, modularization is important when hardware-software co-design is taken into account. We can find several JVMs implemented in hardware[12]. If a JVM is appropriately modularized, exchanging software components with hardware components is also facilitated and becomes one of attractive features for building embedded systems. However, though the modularization is a basic technique, few embedded systems exploit the benefits of the modularization because they are usually modified and extended in an ad-hoc way. For example, CELinux[15], which is one version of the Linux operating system for embedded systems, has been developed in different domains by different working groups, and then the results are integrated. In this case, the developers have to investigate the inconsistency and unanticipated conflicts among the patches at the patch integration time and at runtime. Thus, they have to explore around the source code to identify the problems. Component-based software[13], which is one of modularization techniques, facilitates the problem because each software component is defined as a module with the explicit relationship information between components and context information on which the module properly works. The notion of component-based software is well-known in the desktop and enterprise computing area, but these features are also useful in embedded systems that need to satisfy resource constraints.

EarlGray is a component-based JVM. The system manages the dependencies among components, thus it prevents components from unanticipated linking. EarlGray borrows the source code from an open source JVM, Wonka virtual machine. We have modified Wonka and added the component description. Despite these extra work, the size of the EarlGray executable is only a few hundred bytes bigger than that of Wonka, and the benchmarking results are also almost the same.

This paper presents the design and implementation of EarlGray and some problems while implementing it. The problems that we found is that implicit assumptions in respective components may cause a serious problem when customzing the configuration of the system. We describe the the detailed discussions of the problem in [6]

The rest of the paper is organized as follows. The next section describes the design of EarlGray. We identify the issues on embedded systems. In Section 3, we explain the implementation of EarlGray, and introduce the off-the-shelf virtual machine and the component description tool on which EarlGray is developed. Section 4 describes the evaluation of EarlGray in terms of the size and performance. In Section

Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)
0-7695-2356-0/05 $ 20.00 **IEEE**

5, we show three case studies of the system configuration of EarlGray and the experiences with the configuration. Section 6 describes related work in terms of modular and extensible systems, and component-based system configurations. Section 7 concludes the paper and shows some future directions.

2. DESIGN ISSUES

The goal of EarlGray is to offer systematic customization to satisfy various requirements of future embedded systems. It can be configured and extended in a flexible and consistent way. The systematic configuration is very important for the future embedded systems. For example, ubiquitous/pervasive computing[17] makes embedded systems more complex because it requires computers to be embedded in various artifacts including daily objects and to interact with each other in order to provide new services. In this case, embedded systems are required to be customized according to the various purposes and situations because they do not have enough resources to realize the one-size-fits-all.

Object-oriented programming provides us another way to configure a system in a more systematic way. A system is implemented as a collection of software objects with strictly typed interfaces. An object is an implementation of one or more interfaces. Objects are inter-changeable as long as they implement the same interface. Thus, developers can choose different objects with the same interface for different purposes or in different situations. The architecture of a system is defined as a collection of interfaces, thus, developers can customize the system in a systematic way, for example, replacing an object with an alternative. Usually objects in an object-oriented programming language are dynamically loaded and instantiated at runtime. However, the objects declare only outgoing interfaces or services they provide. The problem is that the runtime system knows nothing about what objects are required for loading an object before accessing the object.

In terms of the system configuration, a component-based system should offer systematic configuration management. Many of the existing systems can be modified their settings by choosing parameters prepared for the system configurations (e.g. Linux kernel configuration). This requires unified configuration parameters through the entire system. A component-based system forces components to support the configuration parameters by defining dedicated software interface. In the case of non-component-based systems, parameters affect a language functions such as the `#ifdef` directive in the C language. Thus, in the non-component-based system, it is difficult to figure out the dependency between parameters. Moreover, parameters are global in a system, thus the name of the parameters have to be differentiated. Software components communicate with others only through their interface. Thus, it is easy to maintain the dependencies among components or to replace components according to the system's requirements.

A developer can configure the EarlGray by organizing or exchanging the EarlGray components. By unifying a way of configuration in this manner, the configuration can be done seamlessly between software/hardware components. A hardware component can be deployed with several software wrappers. On an embedded system in which a software system tends to depend upon its hardware platform, this makes the other software components independent of hardware components as possible.

3. IMPLEMENTATION OF EARLGRAY

We are still using many programs written in the C language, that have been developed in the past, but we like to reuse them although they will be used under various requirements that need to be satisfied. We believe that our studies offer useful information showing how to reuse existing programs in the future, and what problems should be solved to use them under future requirements.

To achieve the goal, we take into account the following two issues to implement EarlGray.

Using the existing system:. Most embedded systems are written in the C language because the C language is originally developed for writing an operating system, thus it is suitable for programs to access to underlying hardware resources. It is very important to build a component-based embedded system based on such the existing embedded systems because developing a component-based system from scratch requires high development cost.

Exposing relationships between components:. An EarlGray component has to be specified with a set of interfaces. Each EarlGray component has a set of input and output interfaces. The inputs specify the services a component requires, and the outputs specify the services a component provides to others. The inputs must be connected with the outputs. In addition, it is also required to describe the links between components explicitly for avoiding link failures at runtime by resolving the dependencies among components.

For satisfying the first requirement, the implementation of EarlGray is based on Wonka, an open source Java virtual machine[18]. Since most embedded systems are developed in the C language, it is difficult to configure the systems in a systematic way. We verify the practical effectiveness of the software component technology though the development of EarlGray based on the existing open source JVM.

For the second requirement, we adopt Knit[11] as the component description language to describe the EarlGray components because: 1) it deals with component implementations in the C and assembly language, 2) we can define both input and output interfaces of a component, and 3) the links among components are explicitly described. Consequently, the EarlGray components described in Knit satisfy the requirements.

3.1 Component Description Language

We have adopted Knit to describe the components of EarlGray. Knit is a component description language developed by the Flux research group at University of Utah for describing software components in OSKit[5].

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 an interface, and the interface is 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 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" }
}
```

Figure 1: An example of an atomic component. `bundletype` defines an interface of a component in which function names 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 to avoid the 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. For example, a function name `gc_create` is renamed to `Collector_gc_create` for avoiding the conflict of the names of functions defined in each component. (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 the phase (2). This is because Knit allows more than one component to be implemented the same interface. The compiler distinguishes the 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 the C or assembly language.

3.2 Structure of EarlGray

This section describes the structure of EarlGray. We divide EarlGray components into three layers as shown in Figure 3.

The interface layer contains two components that access to the lower layer. The Java API component includes components that implement native interface in the Java API. The Native Interface component enables Java and C programs to access to JVM functionalities. The Java API and

```
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 <- [];
  }
}
```

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

Native Interface components directly communicate with the VM layer.

The structure of the VM layer is inspired by the JVM conceptual structure described in [14]. The gray box indicates a component that is a singleton in the system. The white box indicates a component of which multiple instances exist in the system.

The following components are the core components for building the EarlGray Java virtual machine.

Engine: The Engine component plays a central role of the Java virtual machine. The Engine component includes a bytecode interpreter and exception dispatcher.

Loader: The Loader component loads and parses Java class files into the classfile object provided by the Classfile component. The Loader component provides interface to the Engine component and the Heap component but also the Native Interface component, so that the java.lang.ClassLoader class can access the Loader component.

Heap and Collector: The Heap component allocates object instances. The Engine component accesses the instances' variables and methods through the interface of the Heap component. The lifecycle of the instances are managed by the Collector component. The Collector component also accesses to the instances allocated by the Heap component through the Heap component's interface.

The Classfile and Instance component are separated from the Loader and the Heap component respectively because

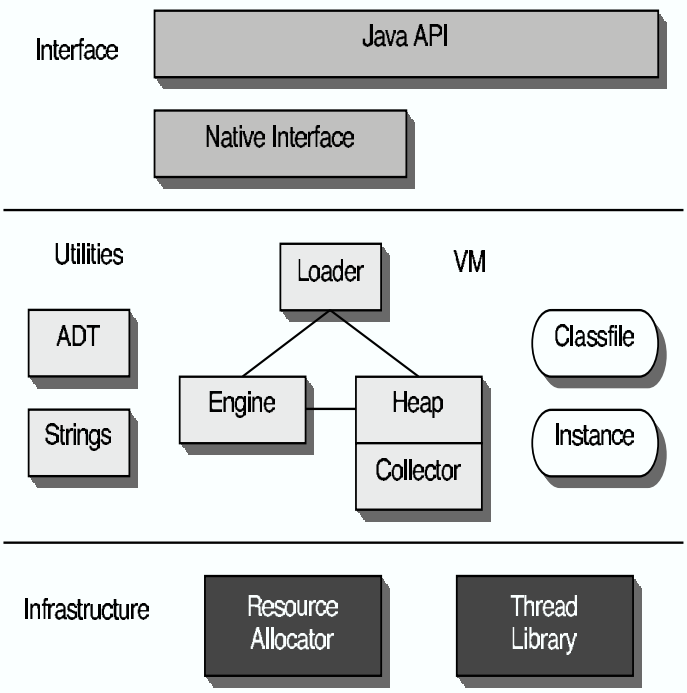

Figure 3: The structure of EarlGray implementation. EarlGray consists of three layers: interface, VM, and infrastructure. The VM layer includes utility components that are accessed from any components in the VM layer. The ADT (Abstract Data Type) component mainly provides the FIFO and hash table object, and the String component provides the functionality to manipulate Java strings.

they are globally accessed from various components. The Classfile component is instantiated for each Java class file loaded into the system. The ADT (Abstract Data Type) component and Strings component are the utilities components used in the VM layer. The ADT component provides several data structures commonly used from other components such as the FIFO and hash table object.

The infrastructure layer consists of the Resource Allocator and Thread Library component. Since these two components provide common functionality that are used in the components in the VM layer, they are placed in the lower layer.

All of the EarlGray component in the figure includes one or more small components. The implementations of EarlGray components are based on the source code of Wonka. The Java API, ADT, Strings, Resource Allocator, and Thread Library component developed by reusing the source code of Wonka.

3.3 Implementation of Components

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 in EarlGray, because it includes many components implementing Java API. On the other hand, the Class Loader component contains only four atomic components.

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.

Currently, the infrastructure layer contains 25 atomic components and 3 compound components. Lastly, the VM layer contains 108 atomic components and 8 compound components, and the interface layer contains 70 atomic components and 7 compound components. All the atomic components are described in Knit and implemented in C.

4. PERFORMANCE EVALUATION

This section compares EarlGray with Wonka which is the original JVM of EarlGray in terms of the program size and performance. In spite of the component description in Knit, EarlGray is as almost the same size and performance as Wonka. Each JVM is compiled by `gcc` version 2.95 with `-O6` option without any debugging options, and does not include the JIT compiler nor AWT support.

4.1 Program Size

The size of each JVM without symbols is almost the same as shown in 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.

EarlGray is 128bytes bigger than Wonka. A component in Knit can include the initialization and/or finalization functions for itself. Knit compiler automatically generates the global initialization and finalization functions that invoke all initialization and finalization functions for components. The difference of the size is caused by these automatically generated routines.

Table 1: The size of program in EarlGray and Wonka

Program	Size (byte)
Earl Gray	567496
Wonka	567368

4.2 Performance

In order to measure the performance of EarlGray, we have executed the Richards and DeltaBlue benchmark programs[16] on EarlGray and Wonka. The Richards is a set of medium-sized language benchmark programs that simulates the task dispatcher in the kernel of an operating system. The DeltaBlue is a constraint solver benchmark program.

Table 2 shows the results of the benchmarks on EarlGray and Wonka. All benchmarks were measured on a 1.2GHz Pentium 3 with 1024MB of RAM running Linux version 2.4.20. EarlGray was compiled with `gcc` version 2.95.4 at optimization level `-O6`. The results were reported by using

Table 2: Performance Evaluation (Execution Time)

Benchmarks	Wonka	EarlGray
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

the benchmark programs themselves. Therefore, they does not include any JVM initializations.

The performance of EarlGray is as almost the same as that of Wonka. Each result is the average of 100 times execution. There are a few differences between EarlGray and Wonka. This is because the locations of functions in an executable file compiled by Knit are different from the one in the original executable file and the result improves the cache effect. In the Knit version, two sets of functions in two atomic components respectively are placed closely in the executable file, if the components are compounded into one component.

5. CASE STUDIES ON COMPONENT-BASED CONFIGURATION

This section shows three configurations by replacing or adding components and describes the side effect of the configurations as case studies. In each case study, we found some problems of a component-based system. Although component interfaces indicate inter-component dependencies, there are other inter-component dependencies that component interfaces cannot 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: *replacing thread scheduler* with a scheduler provided by a host operating system, *modifying bytecode verifier* with a bytecode verifier executed in a remote machine, and *adding a scoped memory management feature* for real-time applications that is one of the features described in the Real-Time Specification for Java[1], to EarlGray.

5.1 Configuration Method

The EarlGray specifies a collection of component interfaces. Each component should implement one of the interfaces. A component interface is the definition of an end point that other components connect to and communicate with. A 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. For example, in the second case study we developed another verifier component that implements the classfile verifier interface `Verifier_T`, and then switched the connection of the original verifier to the new verifier. The reconnection is operated on the compound component that involves the verifier components.

This approach makes the system architecture or the relationship between components clearer than the original source code. For example, it is difficult to understand the relationship among the functions without examining all source code files in Wonka. However, it is much easier to understand the relationship among components by examining component description files. The component description allows us to configure the system by changing links between components. Since the links define all dependencies among components, the configuration can be determined in a systematic way.

5.2 Thread Scheduler

This experiment aims to change a system to use alternative functionalities provided by a platform, instead of them included originally. This change is realized by replacing components. The 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 mechanism of the Java virtual machine, the effect of the replacement must be examined.

We have replaced 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, the monitor and mutex components in the infrastructure layer are also replaced to use the Linux thread library to synchronize threads.

As a result of direct mapping to the scheduler provided by the host operating system, the number of components in the infrastructure layer was decreased. The components in the infrastructure layer originally consist of 17 core components and 4 sub components. 8 components in 17 core components are used only inside of the infrastructure layer. 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 infrastructure layer is completely separated from other components, the new implementation forces none of the other components to be modified in terms of explicit dependencies among components. It is difficult to implement the scheduler interface because there is no development environment for developing a component individually. So the new scheduler component is based on the previous scheduler component. However, in terms of system configurations, EarlGray achieves to separate the scheduler component well.

5.3 Bytecode Verifier

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.

We had an experiment on local-remote configuration by means of bytecode verifier components. The bytecode verifier component running in a local host is the original implementation. We developed an another bytecode verifier component that runs on a remote host based on the origi-

nal 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 `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[9], which is one of the CORBA implementations, as an RPC mechanism.

The configuration in this experiment is done as described in Section 5.1. Although a compound component identifies links among components in the compound component, it cannot describe a link through the computer network. It means that Knit cannot treat the dependencies among components on different hosts. Thus, in this case, the system configuration is not safe.

5.4 Scoped Memory

The aim in the third experiment is to investigate the effect of a change when adding a new component. This experiment implements the scoped memory feature which is one of the features described in Real-time Specification for Java[1]. 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 release 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.

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 still have to be careful to extend a component-based system with additional components. Because there is a chain of implicit dependencies among components.

5.5 Discussions

According to the three 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.

5.5.1 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. 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 to 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.

5.5.2 Implicit Component Behavior

Since the verifier component is located on a remote machine, we have to consider the effect of the network connection between EarlGray and the verifier component. The original verifier component is located on the local machine and composed with in EarlGray 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.

5.6 Summary

The result of the case studies indicates the existance of behavioral dependencies among components. The problem occurs because some assumptions to use respective components are hidden behind their interfaces. Advanced software development methods based on component-based frameworks and aspect-oriented programming do not take into account the ensurance of behavioral assumptions among components. These assumptions should be described explicitly in component specifications, and the assumptions should be checked when components are connected. We believe that these extensions should be incorporated in future component-based frameworks. Also, the extension will be useful in aspect-

oriented programming not to violate the assumptions of an aspect's behavior.

6. RELATED WORK

Jupiter is a modular and extensible JVM developed from scratch[4]. 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, does not address the dependency issues.

Knit has been adopted for building the current version of OSKit[5]. 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[11]. This is the same as our observation that a component-based system contributes comprehensibility. OSKit, however, does not address the dependency issues except interface dependency processed by Knit.

Kon and Campbell[7] 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 do not take into account of any component behaviors.

7. CONCLUSION AND FUTURE DIRECTIONS

As the number of embedded systems grows, Java and Java Virtual Machine are utilized to facilitate complex programming. However, embedded systems usually involves resource constraints, thus system software such as JVM has to be customized. We have presented EarlGray, a configurable component based JVM based on the Wonka virtual machine, and some configuration experiments. Thanks to the component description language, the visibility of the EarlGray's architecture is fine, because the component description language has exposed links among the components. In addition, although these additional implementation, the size and performance of EarlGray has been almost unchanged from the original JVM implementation. Thus, in the case of EarlGray, the software component technology enhanced the system software. According to our case studies, several problems of the current EarlGray implementation are found. Especially, the limit of component interfaces must be solved because reliability and safety are the primary concerns of embedded systems in general. In addition, function-rich consumer embedded systems such as mobile phones require to download software components from the Internet. Thus we need to consider the property of the link descriptions among components. Currently we are improving EarlGray from the functional point of view such as more real-time supports. We are also developing a highly reliable operating system[10], and we are planning to adopt the software component technology to the operating system services such as a file system and networking system.

8. 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] Java Community Process, "Connected Limited Device Configuration".

[3] Java Community Process, "Connected Device Configuration.

[4] P. Doyle and T.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.

[5] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A Substrate for Kernel and Language Research," In proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[6] H. Ishikawa and T. Nakajima, "A Case Study on a Component-based System and its Configuration", In Proceedings of 7th International Workshop on Software and Compilers for Embedded Systems(SCOPES 2003), 2003.

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

[8] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1999.

[9] ORBit, `http://orbit-resource.sourceforge.net/`

[10] S. Oikawa, H. Ishikawa, M. Iwasaki, T. Nakajima, " Constructing Secure Operating Environments by Co-Locating Multiple Embedded Operating Systems", 2nd IEEE Consumer Communications and Networking Conference, ICCN 2005.

[11] A. Reid, M. Flatt, L. Stoller, J. 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.

[12] M.Schoeberl, "Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs", In Proceedings of 10th International Conference on Real-Time and Embedded Computing, Systems and Applications, 2004.

[13] C. Szyperski, D. Gruntz, and S. Murer, "Component Software: Beyond Object-Oriented Programming, 2nd ed.," Addison-Wesley, 2002.

[14] B. Venners, "Inside The Java 2 Virtual Machine," MacGraw Hill, 2000.

[15] CE Linux Forum. `http://celinuxforum.org/`

[16] M. Wolczko. Benchmarking Java with the Richards benchmark. `http://research.sun.com/people/mario/java\_benchmarking/richards/richards.html`

[17] M. Weiser, "The Computer of the 21st Century", Scientific American, Vol.265, No.3, 1991.

[18] Wonka - The Embedded VM from ACUNIA. `http://wonka.acunia.com`