Requirements for a Component Framework of Future Ubiquitous Computing

Hiroo Ishikawa Yuuki Ogata Kazuto Adachi Tatsuo Nakajima
Department of Information and Computer Science
Waseda University
{ishikawa,ogata,karvolta,tatsuo}@dcl.info.waseda.ac.jp

Abstract

System software for future computing environments becomes more complex and heterogeneous. Portability becomes one of the important properties of the future system software. Component-based systems contribute the portability issues. However, component behaviors cause dependencies among components and thus prevent consistent system compositions. We have built a component-based Java virtual machine and tried three configurations with components in order to investigate the inter-component dependencies and the effect of the component behaviors. Also, this paper proposes requirements for a component framework for future ubiquitous computing.

1 Introduction

In ubiquitous computing environments[8], people can access information anywhere using a computer in the place. Many computers in the environments are embedded in hand-held devices and a space. We call such a device and a space, a smart appliance and a smart space respectively.

System software such as middleware and operating systems, for smart appliances or smart spaces becomes heterogeneous and complex at two points. One is the diversity of embedded platforms. An embedded system is specialized for an application due to the resource constraints and costs. There will be various embedded systems in the ubiquitous computing environments, system software will be heterogeneous increasingly. The other is complexity and heterogeneity of services. Although many current embedded systems provide a few services, smart appliances and smart spaces should be able to cooperate with each other and provide various services to users. Software becomes complex in order to integrate services or provide context-aware services[3].

Heterogeneity and complexity of system software inevitably require portability. It is not necessary for a portable system to program the complicated codes again and again. Some portable systems provide various parameters or accessibility to their code for the differences of platforms or requirements.

We believe that component oriented programming is suitable for building future embedded systems. Although component software tends to be studied in terms of building systems by assembling building blocks, we focus on configuration capability of component-based systems. A component-based system allows us to modify it flexibly by replacing components. This property increases system's portability in the heterogeneous environments.

However, the dependencies among components and resource constraints become a serious problem. Components may be developed by different developers, different languages, and different methodologies. When components are assembled, it is necessary to understand the premises before the use. However, in the environment where software is built automatically, or the environment where various systems must be built quickly, the dependencies among components have to be clarified.

The goal of this paper is to investigate the dependencies among components through case studies. The requirements towards component frameworks for future ubiquitous computing are shown as the result of investigation.

The reminder of this paper is organized as follows. The next section describes about a component-based Java virtual machine that we have built as case studies. Three case studies on component dependencies and the result are described in Section 3 and 4. This paper finishes with conclusions and future directions in Section 5.

2 Earl Gray: A Component-Based Java Virtual Machine

For experimental environment, this research project has built a component-based JVM, called Earl Gray, based on the Wonka virtual machine[1] which is developed for embedded systems. All components of Earl Gray are described in Knit component description language[6].

Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES'03)
0-7695-1937-7/03 $17.00 © 2003 IEEE

A component of Earl Gray consists of a set of typed input ports and output ports. The input ports of a component are all the services that the component requires for correct functioning, while the output ports consist of all the services that the component will provide. A port type can be an interface type. An interface type consists of a set of methods, named constants, and other interface types. A component in Knit is a black-box component. The internal implementation of a component is hidden from the clients.

There are two types of components. An atomic component is the smallest unit of composition, while a compound component includes atomic components and/or other compound components. A system is structured by the combination of these two types of components. The implementation of an atomic component is written in C or Assembly language. The atomic component consists of more than one C and/or Assembly source files.

Figure 1 depicts the overview of Earl Gray. Earl Gray consists of three layers. Each layer is represented by a layer component, which is a kind of compound components. Each layer component includes atomic and compound components.

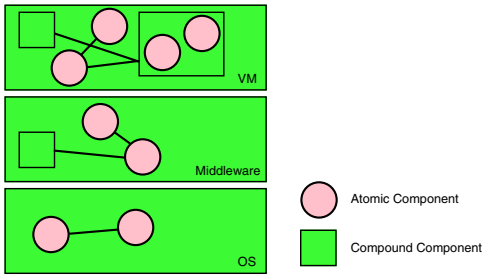

Figure 1. Architecture Overview of Earl Gray

Comprehensibility has been improved by Earl Gray. It is not easy for someone to understand the JVM from the source code, because function caller-callee connections are not explicitly shown, and therefore it takes time to understand the structure of the JVM. However, the links and ports of component descriptions show the structure explicitly and help you to understand it more easily.

3 Case Studies on Inter-Component Dependency

The purpose of this experiment is to explore the implicit dependencies among components. Component interfaces indicate certain inter-component dependencies. However there must be inter-component dependencies that component interfaces can not indicate explicitly. These case studies show the implicit inter-component dependency as a result of several configurations. These case studies consist of the following three cases: (1) replacing the thread scheduler, (2) replacing the bytecode verifier, and (3) extending a real-time feature.

3.1 Replacing Thread Scheduler

The aim of this case study is to investigate how deep the replacement of the default scheduler effects to the system. The thread scheduler is one of the core mechanisms of the Java Virtual Machine. Changing the scheduler ought to affect the system and applications deeply. In other words, this case study investigates which components must be changed as a result of the replacement.

3.1.1 Implementation

This case study replaces the original thread scheduler with a scheduler in the POSIX threads library provided by the host operating system. The original user-level thread scheduler implementation by Earl Gray includes a thread dispatcher mechanism. In other words, the replacement of the scheduler means removing the scheduler mechanism from Earl Gray.

When implementing a new scheduler component, most components in the `OS` layer (see Figure 1) are also replaced because the `OS` layer includes monitors and mutexes which are needed to synchronize threads. Functions for synchronization are provided by the POSIX threads library and are well-suited for the job. Thus they should be used.

Since the `OS` layer is completely separated from other layers, new implementation does not affect the other components and the entire system in terms of interface dependency.

3.1.2 Implicit Dependency on Scheduling Policy

The new component causes the system to stop unexpectedly. A race condition occurred in a function (uncompressing a zip file) where *push* and *pop* functions are invoked. The implementation of the queue structure didn't account for differences in thread-switch timings among the scheduling policies. Originally, the queue was not prepared with a locking mechanism for thread synchronization.

In order to solve the race condition, an additional code for synchronization had to be implemented for the queue component. There was an implicit dependency between the scheduling component and the queue component behind the component's interfaces. It is required for the locking mechanism to provide an initialization and a finalization procedure in addition to lock and unlock functions. As a result, I had to add code for the initialization and finalization for the thread synchronization mechanism outside the queue component.

Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES'03)
0-7695-1937-7/03 $17.00 © 2003 IEEE

3.2 Replacing Bytecode Verifier

The aim of this case study is to investigate the difference between a local component and a remote component, and the effect of the change. A bytecode verifier can throw exceptions when it detects an invalid bytecode sequence. However, in case of a remote verifier, the exceptions have to be invoked not only by an invalid bytecode sequence, but also by network faults such as network crash and latency. In other words, the remote bytecode verifier requires a virtual machine to manage the exceptions of network faults in addition to the default exceptions.

3.2.1 Implementation

The remote bytecode verifier consists of two parts, a stub component and a remote verifier component. Figure 2 depicts the verifier setting. `VM` component requires a component providing the service with a `Verifier_T` interface. `Verifier` (local or stub) component provides that service.

The stub component provides the same interface as a local bytecode verifier component. This is similar to polymorphism in the object-oriented programming(OOP). While the polymorphism in the OOP is the technique to change implementation at run-time, this polymorphism-like technique is executed at compile time. This kind of interchangeability is a benefit of black box component models. Though the interface is the same, implementations may vary from component to component.

The remote verifier communicates with the stub component by means of the RPC (Remote Procedure Call). This implementation adopts ORBit[5], which is one of the CORBA implementations, as the RPC mechanism.

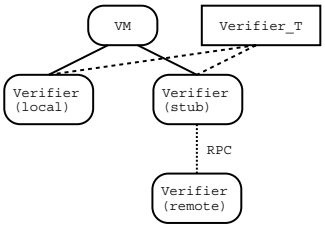

Figure 2. Verifier Setting

3.2.2 Dependency on Exception

The exception is regarded as the implicit dependency in this case. The `Verify_T` interface includes a function that creates a `java.lang.verifyError`, which is thrown when the verifier detects the inconsistent bytecode. Although network errors corresponding to network reliability can occur in the case of the remote bytecode verifier, the interface does not include any functions that create network errors or exceptions. Thus the system does not detect any network errors corresponding to the remote bytecode verifier.

In the case of this implementation, the virtual machine and applications never expect that the remote verifier definitely returns the error, while they may assume that the local verifier returns the result whenever it is called. In other words, the other components in the virtual machine and applications depend on the connection with the verifier component. This is regarded as an implicit dependency because the verifier interface cannot indicate anything about that.

3.3 Extending a Real-Time Feature

The aim in this case is to investigate the effect of a change when adding a new component. A component is a unit of deployment, thus it will not be difficult to extend a system by adding a new component. This case study implements a scoped memory management component, and integrates it with Earl Gray.

3.3.1 Implementation

The scoped memory feature, which is one of the features in Real-time Specification for Java[2], is implemented for this case study. The scoped memory enables an application to deallocate the memory area explicitly by means of scopes or blocks. 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 returns. In other words, instances in the scoped memory area are never collected by the garbage collector. Instead, applications respond to memory allocation with deallocation.

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 that function as a bridge between Java real-time APIs and the virtual machine. The bridge components could be implemented by means of Java Native Interface (JNI), but requires the additional execution cost. Thus the real-time APIs access the virtual machine through the bridge components and the internal API table which maps an API to a bridge function in an one-to-one manner.

3.3.2 Extensibility for the Real-Time Feature

In this case, the extensibility of Earl Gray is the problem. This is a kind of a dependency because the real-time feature requires either the modification of the existing components or adding new components. As mentioned in the previous section, the additional method entries are required on the

Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES'03)
0-7695-1937-7/03 $17.00 © 2003 IEEE

internal API table. Moreover, it is necessary to add a code invoking the `NoHeap` component which is an implementation of the scoped memory manager, because Earl Gray did not prepare any port to extend the memory allocation mechanism. And the thread structure had to be extended because of `RealtimeThread` and `NoHeapRealtimeThread` in `java.realtime`.

Eventually, in order to add the scoped memory component, three components were extended or modified in addition, instead of adding new associated components. Since the scoped memory feature requires the modification of the core of the system, it was impossible to solve the dependency by merely appending the associated components.

Consequently, it is difficult for a component-based system to extend to a real-time system by means of the component technology, especially in case that the system does not provide the extensibility for real-time features.

4 Discussions

According to the experiments, when a system is configured with components, there are implicit dependencies between components even though components are well-separated as in the first case. These studies indicate the dependencies on component behavior. Since component interfaces cannot describe the component behavior, another methodology should describe it. This chapter proposes a methodology to describe component behavior based on finite state machines. The last case study showed that component design and architecture design are important for the evolution of a component-based system.

Currently, we are proposing *Component Interaction Protocol* (CIP) as a requirement for our component framework. The CIP statically describes the behavior of components by means of finite state machines. The benefit of CIP is to be able to describe the internal state transitions of a component which cannot be described by pre- or post-conditions of functions. For example, state transitions of a thread cannot be described because the function to create a thread can effect only a state of creation. Behavior descriptions of finite state machines defined by IOA[4] or CORAL[7] allow a description to express the internal state transitions by means of special signatures. We are currently designing the component behavioral protocols based on IOA and CORAL.

5 Conclusions

In a ubiquitous computing environment where system software becomes heterogeneous and complex, software portability is an important issue. Component-based system allows us flexible configuration to such heterogeneous platforms by means of components as building units. We have investigated the component-based systems in terms of configuration capability.

This research project has built a component-based Java Virtual Machine, Earl Gray, and has investigated the component dependencies through three case studies. The building process of Earl Gray showed the difficulty of decomposing a monolithic system and the comprehensibility of component-based systems. In any cases, the component interface design have affected overall architecture.

The case study revealed the dependencies between components. The dependencies in the first case and the second case were implicit, thus they appeared after building the system, while the dependency in the third case appeared during building the system.

Although a component is defined as a unit of independent deployment, the result of the case study indicates the existence of behavioral dependencies among components. The number of software components will increase in the future, and the constraints for deploying a component become more rigid because of rising embedded systems. The behavioral dependencies have to be considered for consistent composition.

This paper has proposed finite state machines to help solving the component behavioral dependencies. We are currently designing the *Component Interaction Protocol* based on IOA and CORAL.

References

[1] Acunia. Wonka Virtual Machine. `http://wonka.acunia.com`

[2] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr and M. Turnbull. The Real-Time Specification for Java. Addison-Wesley, 2000.

[3] A. K. Dey, G. D. Abowd and D. 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.

[4] S. J. Garland, N. A. Lynch and M. Vaziri. IOA: A Language for Specifying, Programming, and Validating Distributed Systems. MIT Laboratory for Computer Science, October 2001.

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

[6] A. Reid, M. Flatt, L. Stoller, J. Lepreau and E. Eide. Knit: Component Composition for Systems Software. In proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000), October 2000.

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

[8] M. Weiser. The Computer for the 21st Century. Scientific American, 265(30), pp.94-104, 1991.

Proceedings of the IEEE Workshop on Software Technologies for Future Embedded Systems (WSTFES'03)
0-7695-1937-7/03 $17.00 © 2003 IEEE