Towards a Software Framework for Building Highly Flexible Component-based Embedded Operating Systems

Dong XU, Hua WANG, Qiming TENG, Xiangqun CHEN

Operating System Laboratory, Department of Computer Science and Technology
School of EECS, Peking University, Beijing, 100871
{xudong, wanghua, tqm}os.pku.edu.cn, cherry@cs.pku.edu.cn

Abstract. Emerging new computing models make embedded systems become more ubiquitous and pervasive. To adapt the dynamic computing environment, future embedded operating system (EOS) is required to be highly flexible: the static image composition can be configured and the runtime structure can dynamically evolve. In this paper, we present a software framework for building such an EOS through a component-based approach. One unique feature of our framework is its ability of supporting black-box software reuse. This capability permits components from third-party systems to be reused, and frees component developers from the burden of meeting certain implementation constraints imposed by the component model. Based on a flexible binding model, the component runtime service that resides in the nucleus of this framework provides reconfiguration functions to support runtime changes in components and connectors at different levels. To evaluate this framework, we have reorganized uC/OS-II into a component-based one, and we also have implemented a prototype system named as TICK which consists of both native components and reused components. Experiment results show the performance cost induced by our framework is controllable and acceptable.

1 Introduction

As the rapid expansion of the application domain of computing technology, and the growing maturity of network infrastructure, emerging computing models make embedded systems become more ubiquitous and pervasive[1,2]. This trend forces embedded systems to put more focus on the issues of minimization, self adaptability, personalization and etc. To adapt the dynamic computing environment, future embedded systems should be highly flexible, which indicates that the static image composition can be configured and the runtime structure can dynamically evolve. For example, a smart computing node, whose static image composition is configured with its only required components, knows to load TCP/IP protocol stack when it enters a wired network environment, while replacing the TCP/IP stack with Bluetooth protocol when switching to wireless network. The whole process should be done without human administration.

Building flexible EOS from components has been an active area of operating system research. Previous work such as OSKit [3], eCos [4], PURE [5] provide configuration capability by encapsulating functionalities into components, eCos and

PURE also provide tools to aid the configuration process. Systems like Choices [6], 2K [7], Pebble [8] and MMLite [9] provide reconfiguration capabilities by different approaches, but few of them provide reconfigurable ability for system level OS components. Moreover, the component-based approaches of these systems lie in the predefined ways component can interact and bound together.

Another problem in existing component-based EOSes is that there are no rules or tools for supporting extract or produce components from third party systems. Component developers cannot directly reuse components from the system adopting a different component model, and also, without supporting tools, people cannot directly reuse legacy codes in existing non component-based EOS.

This paper presents a software framework for building EOSes from binary components. The binary components can be purposely made or extracted from pre-existing ELF files and composed at build, boot, or runtime to build a system.

The primary contributions of this paper are as follows:

1) Within our knowledge, our framework is the first one that supports black-box software reuse. We propose a unique approach to extract and produce reusable binary components from existing systems, and then reuse them in the framework.

2) We build a component runtime service as the runtime infrastructure that manages system components and user components in a unified way. It enables reconfiguration ability on both system/user components and connectors, and makes the runtime structure evolvable.

3) Our supporting toolkit covers almost the whole construction process.

To evaluate our framework, we have successfully reorganized uC/OS-II into a component-based one. We have also built a prototype system TICK[1] which consists of native components and reused components. Experiment results show the performance cost induced by our framework is controllable and acceptable.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 details our software framework. Section 4 presents the experimental study and section 5 concludes this paper.

2 Related Work

There have been several works in the past decade on building component-based EOS. OSKit [3], an early component system, thinks that application orientation and reusability can be achieved by providing a "box of building blocks", but it has no considerations on the building rules. Systems like eCos [4], PURE [5], Choices [6], Pebble [8], 2K [7] are built based on certain frameworks in which the kernel design specified fixed ways components can interact and be bound together. This issue has been addressed by THINK [10]. Its proposed software framework allows OS architects to assemble components in varied ways, but it does not support dynamic reconfiguration functionalities. All these systems adopt different component models, but none of them provide means to reuse components from each other, and even from the third party systems.

[1] TICK stands for Tick Is Component Kernel

Our framework does not impose a particular kernel design on the OS architect. In this respect, our framework is similar to THINK. Unlike THINK, our framework provides reconfiguration functions to support run-time changes in components and their connectors at both user level and system level. Other differences between our framework and THINK include:

1) Component model: Compared with THINK's component model ODP [11], our component model is a conceptual model, it does not put any constraints on component implementations.

2) Software reusability: Our framework supports black-box software reuse. Codes in either component-based or non component-based systems can be reused, regardless of the different component models.

3) Binding model: Unlike THINK, our binding model dynamically generates connectors from a pre-translated binary binding routine repository, with which synchronization and mutual exclusion issues can be made transparent to components.

3 An overview of our software framework

In this section, we first introduce the overall structure of our framework. Then the approaches of supporting black-box software reuse and runtime structure evolution are presented. We at last discuss the construction process and supporting tools.

3. 1 The overall structure

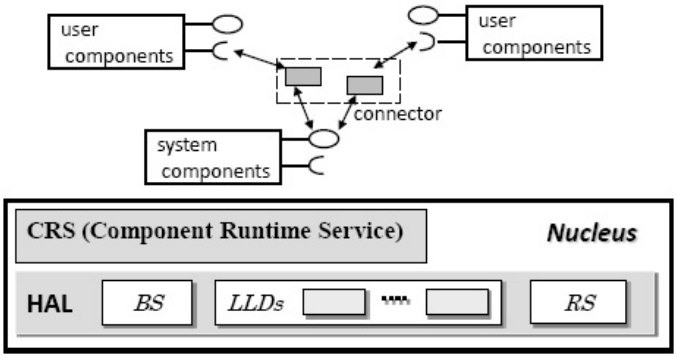

Fig. 1. The overall structure of our framework. BS stands for Bootstrap Service, LLD stands for Low Level Driver and RS stands for Runtime Service.

As illustrated in Figure 1, our framework models the target system as a collection of components and connectors that are collaboratively running with the supports from the underlying nucleus. A component in our framework is a functional entity that encapsulates data and behavior. Our framework supports black-box software reuse, so there are two kinds of components: the native ones are developed from scratch, and the others are directly reused from existing systems. Connectors are a sort of abstraction of the connection relationships amongst components. They can be a few of

instructions that perform a direct call, or be coarse-grained components that encapsulate complicate inter-component communication semantics.

The nucleus, which comprises a HAL and a component runtime service (CRS), provides an abstract interface to hardware and management facilities for upper-level components. HAL provides portability for upper-level components, while the CRS plays a role as runtime infrastructure that supports run-time changes in both components and connectors at different levels. Readers can refer to [12] for an explanation of HAL design. As to CRS, we discuss it in section 3.3.

All ingredients in this framework are statically configurable, including the upper-level components, connectors and the underlying nucleus.

3.2 Supports for black-box software reuse

In our framework, we build components directly from ELF object files [13]. Choosing the binary object files as raw component candidates is reasonable:1)Building binary components can benefit from the fact that programming language used to write the source component is not restricted, developers can write component codes without meeting certain implementation constraints imposed by the component model.2) Building components directly from object files makes it possible to reuse components using different component models without source-level re-encapsulating.3)ELF standard is widely accepted in UNIX family operating systems and other systems, there potentially exist lots of high quality legacy codes to be examined.

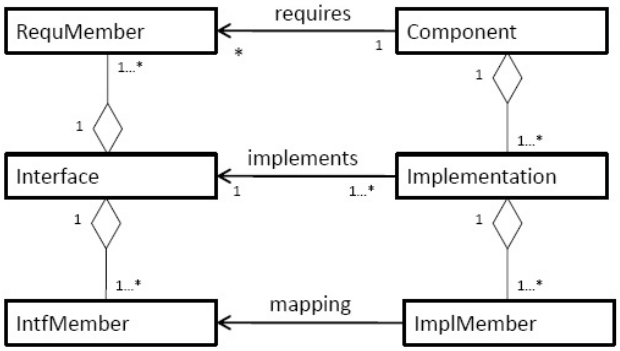

Fig. 2. Conceptual diagram of the component model. We look upon a component as a container for one or more implementation(s) of certain interface(s).

Figure 2 depicts the conceptual diagram of our component model. We look upon a component as a container for one or more implementation(s) of certain interface(s). To correctly operate in a target context, a component may require facilities from other entities in the system. These required (imported) items are referred to as *RequMember*, which are modeled as a tuple (*Intf*, *Impl*, *Member*). When requiring a portal from outside of the component, the component can optionally specify the *Impl* element. A NULL *Impl* element indicates that any implementation that implements the specified interface is acceptable. A component must have at least one implementation of an interface, otherwise the component is regarded useless. For trusted components, the

interface member, viz. *IntfMember*, can be either a variable or a function. Untrusted components can have only functions as interface members. Since the intended domain is an embedded system that might have no memory protection supports, the assumption of direct bindings among components are reasonable. However, for those systems that do support memory protection or privilege modes, no interface can export or import data members directly. In this case, all bindings among components that reside in different protection domains should be done using special facilities (e.g. trappings, upcalls).

Following steps comprise our component production process:

1) For developers who develop components from scratch can directly go to step 3 after compiling the component source codes into object files.

2) Identify and extract candidate object files from legacy systems. We developed a tool named DEREF which can automatically extract and visualize the architecture structure of legacy systems based on cross references among object files [14].With the assistance of related documents and domain knowledge, people can identify and extract required software modules easily.

3) Transform selected object files into reusable, self-contained components. The conversion process is constituted of the following steps: First, we present functions/variables in the object file to users for syntactic information recovering. All function/variable names are reprogrammed upon their signatures, which comes from header files or related documents. By this way, ambiguity caused by duplicate names in ELF files is eliminated. Second, we specify the properties of interface members, such as the reentrancy, visibility and so on. Third, we extract codes and data in the related sections of object files into implementation members, thus the implementations are separated from interfaces. At the same time, we also extract relocation information from object files into the component files. Fourth, in order to prevent hostile modifications, we sign the component with a MD5 fingerprint. Finally, we attach the component description information in the component header.

Since the binary object file is no longer restrained by the component model, it is feasible to re-produce binary components with different component models into our component format. Reusing legacy codes in existing systems can also benefit from this approach.

3.3 Supports for runtime structure evolution

The key nucleus component for supporting target system runtime structure evolution is the CRS. In this section, we primarily focus on the binding model used by the CRS, we refer reader to [15] for the details of all the parts of CRS.

Connectors generated by CRS represent some sorts of inter-component communication semantics. Most of them are implemented into two code snippets named prolog and epilog, respectively. Figure 3 depicts our binding model [16] adopted by CRS: supposing that two components *A* and *B* are loaded into memory by CRS, and that the *functionA* in *A* requires a service *functionB* from *B*, a connector is needed. The connector is dynamically generated according to the user specified binding type. In this case, *functionB* in *B* is specified to be accessed exclusively. The corresponding prolog and epilog thus incorporate *seizeMutex* and *freeMutex* logic

copied from the *Binding Routines* repository. The CRS redirects function call (a relocatable call instruction) in the caller's code to the prolog in which the last *goto* instruction points to *functionB*. The CRS also modifies the calling stack frame of *functionA*, in which the return address of *functionB* ('loc_call' in Figure 3) is replaced by the start address of the epilog, so that it appears to *B* that *functionB* was called by the epilog. When *component B* finishes servicing *A*, the return statement actually transfers the execution to the epilog, which finally goes back to the 'loc_call'. Thus *component A* is ignorant of the intercepting procedure happened: it seems to *A* that the function call behaves just like usual cases, and everything is done as expected.

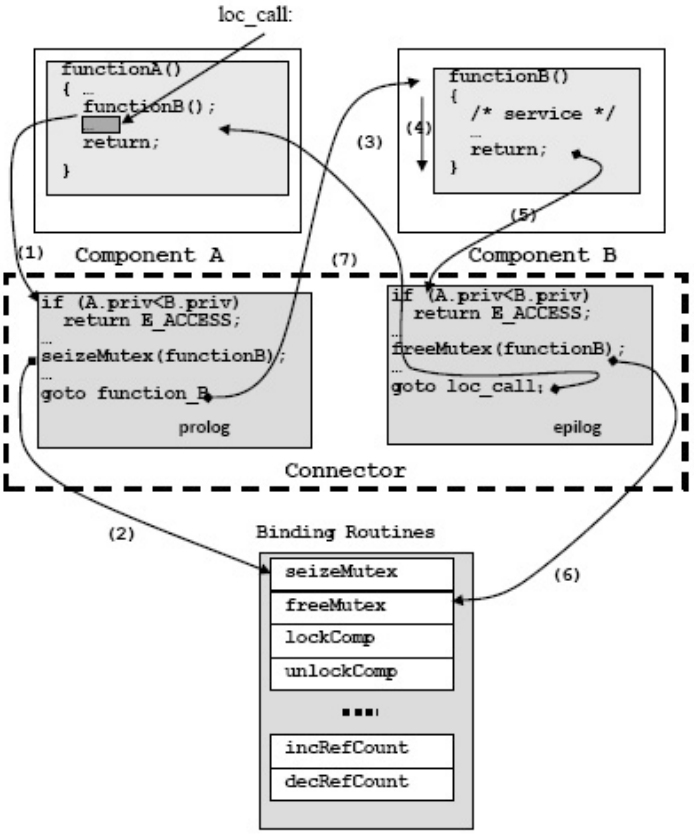

Fig. 3. The binding model. Binding routines are pre-translated binary relocatable code snippets shipped along with the CRS component. This repository is customizable when building the target image.

An interesting point of this binding model is that reference counting, component locking, and cross-domain interactions can be implemented in the same way. Supporting more complicated inter-component communication semantics such as RPC is possible by encapsulating them into coarse grained connector components, thus they can be bound with other components in the same way. By abstracting the

OS specific communication semantics into connectors, issues such as synchronization and mutual exclusion can be made transparent to connected components.

We enable reconfiguration ability of components by adopting techniques like reference count and r/w locks in the connectors. The replacement of connectors is similar to that of components since we can view connector as a special kind of component. We do not support state transfer like the K42 [17] does. State transfer usually adds constraints on component interface semantics, which limits the range of components to be used.

The CRS manages the system/user components and connectors in a unified way, since the OS mechanisms and policies are all encapsulated in upper-level components. By enabling reconfiguration functions, the runtime changes in components and connectors at different levels are supported. As a result, the runtime structure can dynamically evolve.

3.4 Construction process and supporting tools

The construction process for a particular EOS based on our framework is similar to common component-based software development approaches, which follows two major steps:

1) Component acquisition/production: components in target system can either be bought from third-party vendors, or be developed from scratch. They can also be extracted and produced directly from legacy systems (e.g., RTEMS, eCos etc.). All components produced in this phase are collected into a component library.

2) Configuration and composition: at the configuration stage, OS architects select components from the component library, specify the connectors, and configure the components and the nucleus. Once the checking process of composition relationships and constraints passed, the target can be composed at build, boot or runtime.

Tools developed to aid this process include:

- Component repository system: a component library management tool, features including component storage, search etc.
- DEREF: this tool extracts and visualizes architecture structure of legacy systems which helps OS architect identify raw black-box components.
- Component maker: a semi-automatic tool used to product components from ELF object files.
- Composition tool: a visual environment that supports the target system configuration and the component composition.

4 Experimental study

In this section, we have reorganized the non component-based real-time kernel uC/OS-II into a component-based one. We also have built a prototype system named TICK based on our framework. All measurements given below are performed on SAMSUNG S3C44B0X board with a 66MHZ ARM7TDMI processor.

4.1 Reorganize uC/OS-II into a component-based one

We have extracted 14 object files directly from the kernel of uC/OS-II for S3C44B0. We also have encapsulated the kernel C library references into a Clib object file. We have written an application as our benchmark which covers most of uC/OS-II functionalities, and it is used to verify the correctness of the reorganized system. Table 1 compares the size of object files and their corresponding components. We can see that the size of uC/OS components is smaller than that of the object files. This is because the uC/OS object files contain many debugging sections which are not extracted into the corresponding components. The Clib object file does not contain such sections, the attached self description information in Clib component makes it bigger than the original object file. We have built a HAL which only contains a BS module, and all the components are configured to be connected by direct call connectors. The target system composition is set to be done at boot-time.

Table 1. size comparison between object file and component.

	obj. file size(bytes)	**comp. file size(bytes)**
Clib	23,696	44,184
uC/OS	46,646	42,724

Table 2. time cost of boot-time components loading and binding.

instructions	**time(μs)**	**cycles**
5,489	271.65	17,929

The experiment shows that the application successfully starts to run. It proves that our component production approach is feasible to support the black-box software reuse. The time cost of boot-time components loading and binding is shown in Table 2. This cost is acceptable because it has no impacts on the performance of the target system itself. As compared to the statically linked uC/OS-II image which is 67,308 bytes sized, we have introduced more than 19,600 bytes extra space cost. However, this space cost is controllable: because there are no components need to be reconfigurable, the extra space cost induced by component files can be reclaimed after the CRS completes the target system composition. Under the circumstance that the total component size exceeds the memory limitation, we can compose the system at build time rather than at boot-time, thus there is no space cost induced.

4.2 TICK, a component-based EOS

We have built a component-based EOS named TICK based on our software framework. TICK's structure is shown in Figure 4. TICK consists of both native components and reused components. Five types of connectors are configured to connect components. The inter-component communication semantics in these connectors include direct call, system call, mutual exclusion, enable/disable interrupt and lock/unlock component.

4.2.1 Reuse black-box components from other systems

The memory management component in TICK is reused from ThreadX [18] which implements a kind of linear memory management algorithm. Also, we extracted 8 object files from VxWorks I/O subsystem, and developed an 'adapter component' to convert their external dependencies into TICK's components, interfaces of TICK components involved in this adaptation including semaphore and memory management.

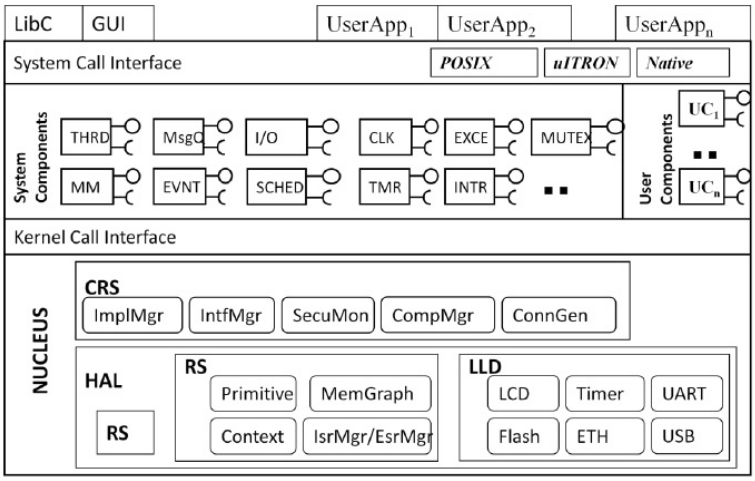

Fig. 4. The structure of TICK.CLK is a component for clock management. TMR is timer component and EXCE is exception management component. ETH stands for the low-level driver of Ethernet adapter.

4.2.2 Reconfigurable ability

In TICK, a prolog/epilog template is used to generate connectors:

```
UINT32 bindConnectCode[] = {
    /*prolog:*/
    0xe92d100f,     /* stmfd sp!, {r0-r3,ip},   push registers */
    0xe59f0038,     /* write Param1 of preStub */
    0xe59f1038,     /* write Param2 of preStub */
    0xe59f2038,     /* write Param3 of preStub */
    0xe59f3038,     /* write Param4 of preStub */
    0xeb000000,     /* call preStub,need to be   relocated */
    0xe8bd100f,     /* ldmfd sp!, {r0-r3, ip},   pop registers */
    0xeb000000,     /*branch required_func,need   to be relocated*/
    /* epilog: */
    0xe92d100f,     /* stmfd sp!, {r0-r3, ip} */
    0xe59f001C,     /* write Param1 of postStub */
    0xe59f101C,     /* write Param2 of postStub */
    0xe59f201C,     /* write Param3 of postStub */
```

```
    0xe59f301C,     /* write Param4 of postStub */
    0xeb000000,     /* call postStub,need to be relocated */
    0xe8bd100f,     /* ldmfd sp!, {r0-r3, ip} */
    0xea000000,     /*back to the instruction after required_func */
    0x00000000,     /* value of preStub Param#1,to be relocated */
    0x00000000,     /* value of preStubParam#2, to be relocated* /
    0x00000000,     /* value of preStub Param#3,to be relocated */
    0x00000000,     /* value of preStub Param#4,to be relocated */
    0x00000000,     /*value of postStub Param#1,to be relocated*/
    0x00000000,     /*value of postStub Param#2,to be relocated*/
    0x00000000,     /*value of postStub Param#3,to be relocated*/
    0x00000000,     /*value of postStub Param#4,to be relocated*/
};
```

Based on this template, we induced extra 96 bytes, 64 bytes of which are used for 16 instructions, and the other 32 bytes are used by data for at most 8 parameters. The cost of this template is given in Table 3.

The 'preStub' and 'postStub' in this template refers to the routines that implemented certain communication semantics. Table 4 summaries performance of connectors in TICK. A direct call connector is generated by filling the callee's address into the caller's function call instruction, so there is no extra cost in this case. System call connector does not use the prolog/epilog code in this template. We registered a 'system call binding service' as a system call routine that will call the function specified in its parameters in supervisor mode. When a system call binding is required, the connector code will put the required function's address and parameters in general purpose registers, and then directly call the software interrupt instruction to request the system service.

Table 3. Performance cost of prolog/epilog template

Instructions	time(μs)	cycles
16	0.93	62

Table 4. Performance of connectors in TICK

Communication semantics	instructions	time(μs)	cycles
system call	61	2.15	142
mutual exclusion	260	9.01	595
enable/disable interrupt	26	1.21	80
lock/unlock component	761	24.98	1,649

The performance cost of each connector equals to the cost of prelog/prolog template plus the cost of corresponding preStub/postStub. Table 3 tells us that the cost of prolog/epilog is negligible. The cost of each preStub and postStub which implement certain communication semantics depends on its implementation, and this kind of cost is inherent in the interacting scheme.

To evaluate the performance of dynamic reconfiguration, we built two components A and B which encapsulated different priority-based scheduling policies. A is only

bound with scheduler component, and B is stored in host machine. Besides of the access semantics, the connectors between A and scheduler component encapsulate logic of reference counting and read lock acquisition. When dynamic reconfiguration event (we implemented it as a kind of user-defined event) is triggered, the CRS will download component B from host machine. After loading component B into memory, CRS will add a write lock on component A, and will rebind the scheduler component with component B once the reference count on component A becomes zero.

Table 5. Performance evaluation on a dynamic reconfiguration case

Step	instructions	time(μs)	cycles
load and internal relocate	1,644	49.43	3,263
bind	35	1.13	75

Table 5 shows the performance of this dynamic reconfiguration case. The cost of internal relocation is requisite when loading relocatable component into memory. The cost of runtime component binding depends on two factors: 1) the number of required functions/variables in components. In current implementation, each required function/variable needs to take 15 cycles when connecting itself with the corresponding provided one. Since the number of required functions/variables is configurable, this kind of cost is controllable.2) the performance of connectors. As we analyzed before, the extra cost induced by connectors is negligible. In real application scenario, only a few components (e.g., those encapsulate system policies) often need to be replaced, so the performance cost of dynamic reconfiguration is actually small.

5 Conclusion and future works

We have presented a software framework for building highly flexible component-based EOS that the static image composition can be configured and the runtime structure can dynamically evolve. We have discussed the component production approach as well as the component model, with which the black-box software reuse is supported. We have detailed the runtime infrastructure primarily on the binding model, which provides reconfiguration ability on the runtime structure. Also we have briefly introduced the construction process and supporting tools. To evaluate this framework, we have reorganized uC/OS-II into a component-based one, and we also have built a prototype system TICK using our framework. The evaluation results show that our approach is feasible and the performance penalties are controllable and acceptable.

Future works are to be carried out in the following aspects:

- Investigating method to verify the correctness of the binary component composition.
- Optimizing the CRS code, as well as the component organization format, thus the space cost can be lowered.
- Developing more kinds of connectors to support more complicated inter-component communications such as RPC.
- Exploiting existing EOS to enhance the component library.

Acknowledgments
Special thanks to Yi ZHAO who helped us tremendously improve on an earlier version of this paper. This work is sponsored by the National High-Tech Research and Development Plan of China under Grant No.2002AA1Z2301 and 2004AA1Z2400.

References

1. Mark Weiser, Hot Topics: Ubiquitous Computing. IEEE Computer, Oct.1993, 26(7):71-72.
2. Ben Hendrickson , Adam MacBeth , Steven Gribble, Systems Directions for Pervasive Computing, In Proc. of the 8th Workshop on HOTOS, p.147, May 20-22, 2001
3. Bryan Ford , Godmar Back , Greg Benson , et al. The Flux OSKit: a substrate for kernel and language research, In Proc. of the 16th ACM SOSP, p.38-51, Oct. 05-08, 1997
4. eCos. http://sources.redhat.com/ecos/.
5. Beuche, D., Guerrouat, A., Papajewski, H., et al. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In Proc. of the 2nd IEEE international Symposium on Object-Oriented Real-Time Distributed Computing (May 02 - 05, 1999). ISORC. IEEE Computer Society, Washington, DC, pp.45-53
6. Roy Campbell, Nayeem Islam, Peter Madany, et al. Designing and Implementing Choices: An Object-Oriented System in C++, Communications of the ACM, vol. 36,no. 9, Sept. 1993, pp. 117-126.
7. Kon, F., Singhai, A., Campbell, R. H., et al. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In Workshop Ion on Object-Oriented Technology. LNCS vol. 1543. Springer-Verlag, London, 388-389.
8. Gabber, E., Small, C., Bruno, J., et al. The pebble component-based operating system. In Proc. of USENIX Annual Technical Conference 1999, CA, 20-20.
9. Johannes Helander, Alessandro Forin, MMLite: A Highly Componentized System Architecture, In Proc. of 8th ACM SIGOPS on Operating Systems European Workshop ,ACM Press, New York, 1998, pp. 96-103.
10. J-P. Fassino, J-B Stefani, J. Lawall, et al. THINK: A Software Framework for Component-based Operating System Kernels.USENIX 2002 Annual Technical Conference, CA, 73-86.
11. ITU-T Recommendation X.903 | ISO/IEC International Standard 10746-3. ODP Reference Model: Architecture.ITU-T | ISO/IEC, 1995.
12. Teng, Q., Wang, H., and Chen, X. 2005. A HAL for Component-Based Embedded Operating Systems. In Proc.of the COMPSAC 2005. IEEE CS, Washington, DC, 23-24.
13. TENG Qiming, CHEN Xiangqun, ZHAO Xia. On Building Reusable EOS Components from ELF Object Files. Journal of Software, Vol. 15, Issue 12a, 2004, pp. 157-163.
14. Teng, Q., Chen, X., Zhao, X., et al. Extraction and Visualization of Architectural Structure Based on Cross References among Object Files. In Proc. of COMPSAC 2004. IEEE Computer Society, Washington, DC, 508-513.
15. Dong Xu, Qiming Teng and Xiangqun Chen, Supports for Components Loading and Binding at Boot-time for Component-based Embedded Operating Systems, In Proc. of ICESS 2005. IEEE CS Press, Dec. 2005, Xi'an, pp. 46-52
16. Qiming TENG, Xiangqun CHEN, Xia ZHAO. On Generalizing Interrupt Handling into A Flexible Binding Model for Kernel Components. In Proc. of ICESS 2004, pp.323-330
17. Baumann, A., Heiser, G., Appavoo, J., et al. Providing dynamic update in an operating system. Proc. of USENIX 2005 Annual Technical Conference. CA, 32-32.
18. ThreadX. http://www.rtos.com/