一种基于 ELF 目标文件的可复用 EOS 构件制作方法*

滕启明+, 陈向群, 赵 霞

(北京大学 软件研究所 操作系统实验室,北京 100080)

On Building Reusable EOS Components from ELF Object Files

TENG Qi-Ming+, CHEN Xiang-Qun, ZHAO Xia

(Operating System Laboratory, Institute of Software, Peking University, Beijing 100871, China)

+ Corresponding author: E-mail: tqm@cs.pku.edu.cn, http://os.pku.edu.cn

Received 2004-01-16; Accepted 2004-03-29

Teng QM, Chen XQ, Zhao X. On building reusable EOS components from ELF object files. *Journal of Software*, 2004,15(Suppl.):157~163.

Abstract: In this paper, an approach to generate reusable components from ELF object files is presented. Compared with other software component encapsulation technologies, building reusable software component directly from binary form ELF files is a straightforward process. Symbol tables and relocation information encapsulated in ELF files can be automatically reprogrammed into a self-contained reusable entity. By augmenting the component with syntactic interface descriptions, ambiguity caused by duplicate names in ELF files is eliminated. Binary code scanning is employed to enhance the safety of the component. Further, to prevent unintentional or hostile modifications, the component is signed with an MD5 fingerprint.

Key words: software reuse; software component; embedded operating system; object file

摘 要: 给出了一种基于 ELF 目标文件生成二进制可复用软件构件的方法,并在 EOS 领域作了初步尝试.与其它软件构件封装技术相比较,从 ELF 目标文件生成可复用二进制软件构件(称作 CELF 格式)的过程更为直接.为解决原 ELF 格式中名字冲突问题,非安全指令问题以及来源信任问题,分别采用了接口元素名称再编码、源指令流扫描及 MD5 文摘生成技术.可复用的 EOS 构件中包含源 ELF 文件的代码及数据,改进了其中符号表和重定位信息组织,独立部署、参与组装的可能性得到增强.

关键词: 软件复用;软件构件;嵌入式操作系统;目标文件

Due to the two main drawbacks inherent in pure micro-kernel operating system design, i.e., poor performance and inability to tailor servers to applications demands and needs, an alternative way to construct extensible

---

* Supported by the National High-Tech Research and Development Plan of China under Grant No2002AA1Z2301 (国家高技术研究发展计划(863))

TENG Qi-Ming was born in 1975. He is a Ph.D. candidate at the Institute of Software, Peking University. His research interests are operating system and software engineering **CHEN Xiang-Qun** was born in 1961. She is a professor and post-graduates supervisor of the Institute of Software, Peking University. Her current research areas are operating system and software engineering. **ZHAO Xia** was born in 1972. She is a Ph.D. candidate at the Institute of Software, Peking University. Her research areas are operating system and software engineering.

© 中国科学院软件研究所 http://www.jos.org.cn

operating system is to investigate better architectural abstractions that enable finer granularity in OS components. Although the community has brought forward many ideas with prototype systems around the "extensible" operating system approach, issues related to the efficient, flexible, safe extensions are still to be addressed[1,2].

In this paper, we introduce an approach to produce reusable software components directly from compiled code, namely, an ELF object file. The organization of this paper is as following: In Section 1, we present the primary challenges in producing reusable EOS components from object files; in Section 2, an overview of the binary format of ELF object files is given; Details on conversion along with additional considerations about the usage scenario are presented in Section 3; related works related to drafting extensions into OS kernel is introduced in Section 4; the initial lessons obtained and future works are given in Section 5.

1 Introduction

The background of this research is an on-going project to investigate feasible approaches to construct component based embedded operating systems. The aim is to introduce the CBSD (Component Based Software Development) process into embedded application developments, in hope that the target system features a flexible structure and ability to be customized in various application fields. In such an embedded system, there might exist no memory protection supports by hardware, available resources (power, RAM etc) are strictly constrained. However, when compared with general operating system requirements are more application specific, e.g. some applications require real time capability, while others not. The move towards extensible embedded operating systems is an evolutionary step on the path from monolithic kernels to microkernel architectures. Related researches indicate that, along with adding functionality, extensible systems can provide applications with the ability to override policy decisions[3]. In his work, Christopher proposed taxonomy of grafts and of grafting architectures. One of the conclusions drawn from his work is that by controlling compiled languages and patching binary codes, one can provide software protection while enjoying the feasibility brought by drafting extensions into the kernel[3].

Thus, the question becomes that how to inject extensions into the kernel that can

- enable new functionalities without compromising system reliability
- enable new paradigm to construct EOSes without imposing too much burden on developers
- enable flexible customization to systems without apparently degrading the overall performance

Our solution to these problems is to semi-automatically convert object files from legacy systems into a standard format, which is described in Section 3. By supporting flexible late binding, dynamic loading and unloading, trusted components (e.g. conventional services such as scheduler, drivers) can interact with each other via direct calls, while untrusted components (e.g. user applications) can interact with service components with indirect bindings.

2 Evaluation of the ELF Format

The ELF (stands for Executable Linking Format) was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). It is a replacement to the A.OUT format used by older versions of UNIX systems. The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments[4]. The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works across multiple 32-bit environments for a variety of operating systems. The ELF is widely accepted in the UNIX family operating systems and others.

The standard is divided into three parts: 1) ELF general; 2) Processor Specific; 3) Operating System Specific. There are three main types of object files in ELF, namely:

© 中国科学院软件研究所 http://www.jos.org.cn

- A *relocatable file* contains code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution on an operating system.
- A *shared object file* contains code and data suitable for dynamic linking purpose.

Our concern is focused on the *relocatable file* format, which is the immediate output of a compiler (e.g. the Gnu Compiler Collection, GCC). Created by the assembler and link editor, object files are binary representations of programs intended to execute directly on a processor. When compared with source code representations, binary format of the program is free of lexical, syntactic tricks such as compiler specific directives, macros, inline functions, configure and building scripts. The internal organizations of control flow and data flow are determined, and thus exhibit a suitable candidate for a raw black-box component.

2.1 Object file format

The general layout of an ELF object file is given in Fig.1. General information is given in the ELF header part, while information about all sections is organized as a table as the last part. Each type of section has its own record or data format, details about which can be found in Ref.[4].

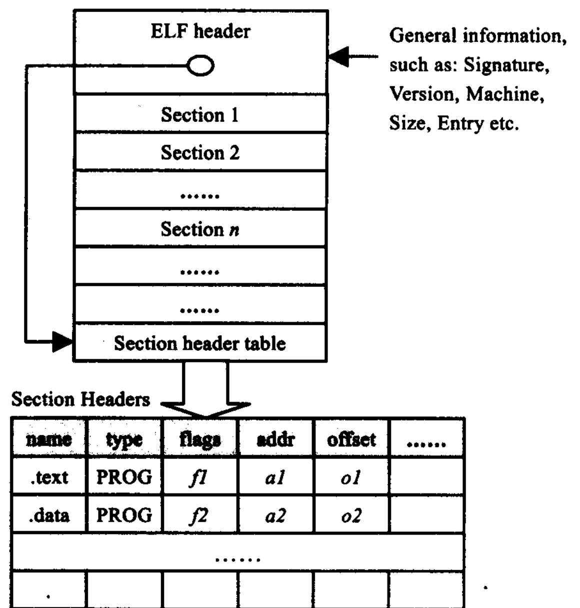

Fig.1 General layout of a ELF

Among these sections, we are most interested in the ".text", ".data", ".rodata", ".bss", ".strtab", ".symtab", ".rel.text", ".rel.data", ".rel.rodata" sections (Table 1). From these sections, we can extract the following information:

- size and content of the code, data, read only data of the object
- size of the uninitialized data of the object
- name and entry of each global/local functions
- name and location of all global/local variables
- name and referenced location of all external functions and/or variables, and the way to resolve it when linked with other object file(s)
- initial value for initialized data and read only data

Based on the information provided by the object file itself, linkers can generate executable programs and/or relocatable objects (i.e. composite components).

2.2 Cons of ELF object files

Conventional tool chains put much more emphasis on compilers than on linkers, with the latter often left stupid, simple tasks, i.e. symbol resolving, relocating etc. This is due to the implicit assumption that the compiler has already done every thing related to program semantics well. When encountering conflicted names or unresolved symbols, the linker just exits with a complaint.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 1 Significant sections

name	type	attribute	note
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR	program code
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE	program data (initialized)
.rodata	SHT_PROGBITS	SHF_ALLOC	program data (read only)
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE	program data (not initialized)
.symtab	SHT_SYMTAB	(SHF_ALLOC)	symbols
.strtab	SHT_STRTAB	(SHF_ALLOC)	names of all symbols
.rel.text	SHT_REL(A)		relocation info for code section
.rel.data	SHT_REL(A)		relocation info for initialized data
.rel.rodata	SHT_REL(A)		relocation info for read only data

Although it is possible to develop an equivalent format which might be simpler than ELF, the fact that an object is not intended for being used by third parties restricts its expressiveness. The major shortages of the ELF format are: 1) Data type information is rarely left in the file. From the object file, only the size of a particular variable can be obtained so that viable type-safe linking is almost impossible. 2) Signatures of functions are often removed by the compiler (if the source code is written using C, for example). This prohibits composition of modules containing functions whose names are identical, but implementing distinct functionalities. 3) Due to the lack of global information, an ELF object file does not distinguish external variables from functions explicitly.

To make an ELF object file into a self-contained reusable software entity, we have to augment it with syntactic knowledge that can be used by a run-time component manager for successful binding.

3 Conversion of ELF Object Files into Reusable Components

3.1 The target format

In this subsection, we present the intended target format of a reusable binary object file named EBC (ELF Binary Component). The conversion process will be given in Subsection 3.2. The conceptual image of a reusable component generated from an ELF file is as shown in Fig.2.

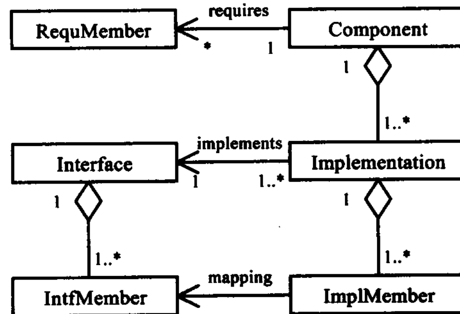

Fig.2 Conceptual diagram of EBC

We look upon a component as a container for one or more implementation(s) of some 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. Specifying a NULL *Impl* element means 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. This is a special optimization considering that trusted components can directly access any exported data members of others. 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 that direct bindings among

© 中国科学院软件研究所 http://www.jos.org.cn

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 that case, all bindings among components reside in different protection domains should be done using special facilities (e.g. trappings, upcalls).

Besides these, we have to deal with problems such as naming confusion and data integrity. Currently, we are adopting the C++ name mangling scheme implemented by the GNU Compiler Collection for function signatures. For example, an interface item named "IScheduler::Suspend(unsigned int)" is mangled as "Suspend_10ISchedulerUi". For convenience, we want both the original name obtained from ELF file ("Suspend" in this case) and its mangled name stored in the target EBC file. To prevent the component from malevolent modification, we want a MD5 fingerprint appended to the EBC file as suggested in Ref.[5]. The layout of EBC file is given in Fig.3.

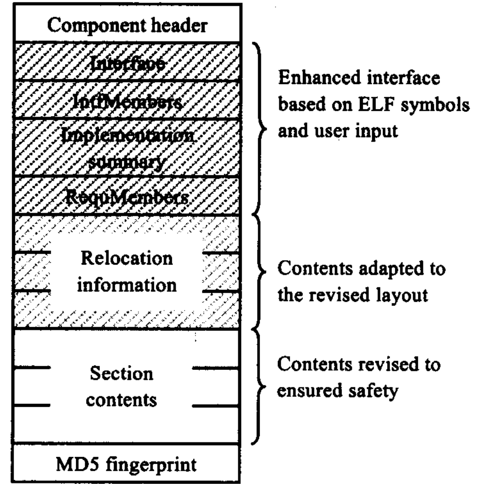

Fig.3 Layout of the EBC file

3.2 The conversion process

The conversion process constitutes of five steps as shown in Fig.4. A tool named E2C is specially developed for integrate the five steps into an interactive environment.

1) Unsafe code scanning: To prevent unauthorized component from issuing privileged instructions such as disable interrupt, accessing an I/O port, the code scanner analyzes the code section of the ELF file. When encountered unsafe opcode in code streams, a warn will be issued by the E2C Tool. On IA32 architecture, besides the 28 privileged instructions are forbidden, IN/OUT instructions are generally not allowed either.

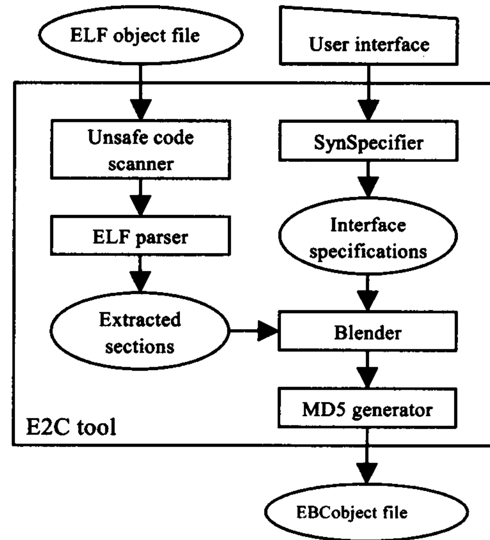

Fig.4 The conversion process illustrated

2) If the object file successfully passed the code validation, then its original organization is broken by the ELF parser. The result is a collection of sections, along with some useful general information.

3) Provided (and required) functions/variables in the object file are presented to the user for syntactic information recovering. User can choose to hide specific *IntfMember*(s) from outside, as well as to specify alias for elements so that flexible matching purpose during the runtime binding is possible.

4) Given the interface specifications and extracted sections, the *Blender* component performs reprogramming on the input, emitting a ready-to-use component.

5) The last step is to pass the component through a MD5 filter. A 128bit fingerprint value computed from the component is then appended at the end.

Component developer can specify complementary attributes for the component and/or for each *IntfMember*. Some attributes are given in Table 2.

© 中国科学院软件研究所 http://www.jos.org.cn

Table 2 Complementary attributes

Object	Sample attributes & meanings	
Component	CP_RELOCATABLE(0x01)	The component can be relocated when loaded, or else the component specifies an absolute address to be loaded at.
	CP_LOADPHASE1(0x02)	The component must be loaded when interrupts are still disabled at system boot time.
	CP_LOADPHASE2(0x04)	The component can be loaded when interrupts are enabled.
	CP_DYNAMIC(0x08)	The component will be loaded when the target system has been initialized as a multi-tasking environment.
	CP_PERMANENT(0x10)	The component that can not be removed once loaded into kernel.
Functional interface member	IMO_EXCLUSIVE(0x01)	The function must be entered exclusively. This critical section should be ensured by target system.
	IMO_NOINTR(0x02)	Whether interrupts are allowed when this entry is used.
	IMO_SYNC(0x04)	When this entry is already entered by a thread of execution, further attempts should be queued.
Data interface member	IMA_READONLY(0x01)	The data member exported can be read only, no modification is allowed.
	IMA_READWRITE(0x2)	The data member can be modified.

4 Related Work

Building binary components can benefit from the fact that the programming language used to write the source component is not restricted. Microsoft COM technology exhibits this feature when orchestrating the interactions among binary components. Components can be written in C++, Visual Basic or other languages, provided that the programming language supports calling a function via in-memory pointers[6]. However, imposing an indirection on each call is sometimes not acceptable in embedded systems, particularly those with demanding real-time requirements.

The SPIN system[7] is written in and uses as its extension language, Modula-3[8]. Modula-3 is a strongly-typed, garbage-collected language, designed so that it is impossible for a program to construct a pointer to an arbitrary memory location. There are still other systems that introducing language specific support into the operating system kernel[9]. However, we consider it not a flexible (if feasible) approach to address the reuse problems.

5 Initial Lessons and Future Researches

Through our initial works by converting existing ELF object files into the EBC format, we believe that reusing modules from legacy systems is possible. The issues to be addressed include: how to recover the semantics effectively since the component developer might misunderstand the intended usage of an interface member; how to extract function level reusable assets from these legacy properties etc. We are currently focusing on the implementation of composite component. Future research will be focusing on a even finer granularity of component form, i.e. function body extraction.

References:

[1] Margo IS, Christopher S, Keith S. The case for extensible operating systems. Technical Report TR-16-95, Department of Computer Science, Harvard University, 1995.

[2] Cheung WH, Anthony H, Loong S. Exploring issues of operating systems structuring: from microkernel to extensible systems. Operating Systems Review, 1995,29(4):4~16.

[3] Small C, Seltzer M. A comparison of OS extension technologies. In: Proc. of the 1996 USENIX Conf.. San Diego, CA, Jan. 1996. 41~54.

[4] TIS(Tool Interface Standard) Committee, Executable and Linking Format (ELF) Specification, Version 1.2, May 1995.

[5] Rivest R. The MD5 message-digest algorithm. Network Working Group RFC 1321, April 1992.

[6] Microsoft Corp. The Component Object Model Specification. Version 0.9. Redmond, WA, 1995

© 中国科学院软件研究所 http://www.jos.org.cn

[7] Bershad B, Savage S, Pardyak P, *et al*. Extensibility, safety, and performance in the SPIN operating system. In: Proc. of the 15th SOSP. Copper Mountain, Co. December 1995.

[8] Nelson G. Systems Programming with Modula-3. Prentice Hall, Englewood Cliffs, NJ, 1991.

[9] Wilson CH, Marc EF, Charles G, *et al*. Language support for extensible operating systems. In: Proc. of the Workshop on Compiler Support for System Software, February 1996.

© 中国科学院软件研究所 http://www.jos.org.cn