Constructing Component-Based Extension Interfaces in Legacy Systems Code

Gilles Muller[†], Julia L. Lawall[**], Jean-Marc Menaud[†], and Mario Südholt[†]

[†]OBASCO group
École des Mines de Nantes/INRIA
44307 Nantes Cedex 3, France
{Gilles.Muller,Jean-Marc.Menaud,Mario.Sudholt}@emn.fr

[**]DIKU
University of Copenhagen
2100 Copenhagen Ø, Denmark
julia@diku.dk

Abstract

Implementing an extension of a legacy operating system requires knowing what functionalities the extension should provide and how the extension should be integrated with the legacy code. To resolve the first problem, we propose that the use of a component model can make explicit the interface between an extension and legacy code. To resolve the second problem, we propose to augment interface specifications with rewrite rules that integrate support for extensions in the legacy code. We illustrate our approach using extensions that add new scheduling policies to Linux and prefetching to the Squid Web cache. In both cases a small number of rules are sufficient to describe modifications that apply across the implementation of a large legacy system.

1 Introduction

Component models have been shown to be useful in building specialized operating systems (OSes), such as those dedicated to network routers [5], appliances [11], or the support of advanced programming languages [6]. Component-based systems include OSKit [6], Pebble [7], Think [5], Windows CE, and Windows XP Embedded. The use of a component model provides modularity, making it easy to add and remove functionalities, and allows automatic verification of component composition.

Previous component-based approaches have focused on creating a new OS from a set of existing components. We consider instead the use of components in enabling the extension of legacy OSes. In this setting, the OS amounts to a single component that exports an extension interface. Extensions are also encapsulated as components. This approach brings the aforementioned benefits of the use of a component model to the problem of OS extension. Nevertheless, there remains the issue of converting a legacy OS into such a component.

The OSKit has pioneered the idea of using legacy OS code to construct OS components. Implementations of system services, such as drivers and file systems, are borrowed from Linux and BSD. OSKit interfaces are constructed around this legacy code using wrappers that rename entry points, rearrange arguments and perform similar glue behaviors. This approach is sufficient when the need for OS modularity and the basic points of interaction are well-established. Constructing a new extension interface requires, however, not only wrapping existing entry points so that they can be used by an extension, but also modifying the legacy code so that it invokes the extension at appropriate points in its execution. Interposing this new behavior requires fine-grained reengineering of the legacy code.

In this paper, we propose to augment the specification of a component interface with rewrite rules that describe how to integrate support for the interface into legacy code. To allow interposing this support within existing function definitions, the rewrite rules use temporal logic to precisely describe rewriting sites in terms of control-flow patterns [1, 9]. To facilitate evolution to new versions of the legacy code, these rewritings are performed using an automatic tool. Once support for the component interface has been constructed in the legacy code, the resulting component and the extension components are assembled according to the underlying component model to produce the final system.

We illustrate our approach in the context of process scheduling and Web caching. In the context of

process scheduling, we use our approach to construct an interface in Linux 2.4 to support the Bossa framework for extending legacy OSes with new scheduling policies [10]. The extension interface for Linux 2.4 imports 13 functions, exports 20 functions, and uses 23 rewrite rules. Rewriting sites occur all across the kernel, requiring that the rewrite rules be applied to source code files amounting to almost 100MB. In the context of Web caching, we use our approach to construct an interface in the Squid Web cache to support extension of Squid with specialized prefetching policies. Such policies, *e.g.*, directed to new protocols or particular services, have been shown to significantly improve Web cache performance [8]. The extension interface for Squid imports 4 functions, exports 25 functions, and uses 4 rewrite rules. In previous work, we have developed automated tools that perform the legacy code modifications required for Bossa [1] and for Squid prefetching policies [13]. Nevertheless, the interfaces between the various components involved in these systems were not clearly defined. In both cases, expressing the extension interface explicitly in a component model enforces a clean separation between the legacy system and the extensions, facilitates evolution to new versions, and makes explicit both the resources that an extension must provide and the resources that it can expect to re-use from the legacy system.

The rest of this paper is organized as follows. Section 2 presents our approach in more detail. Section 3 applies this approach to the problem of extending Linux to support Bossa. Section 4 applies this approach to the problem of extending Squid to support prefetching. Finally, Section 5 presents related work and Section 6 concludes.

2 Extending Knit with Fine-Grained Rewrite Rules

We have instantiated our approach as an extension of the Knit component model [12]. Knit is targeted toward the needs of systems code and has been used in the context of the OSKit. We augment the interfaces of the Knit component model with rewrite rules that describe how to integrate support for an extension interface in a legacy system. A Knit interface is described as a *unit*, which lists the names imported and exported by a component, as well as optional dependency, initializer, and finalizer information. The imported and exported objects are defined using *bundle types*, which group related names. In our approach, each name declared within a bundle type is accompanied by a rewrite rule that describes how to add support in the legacy code for importing or exporting the named object.

The information contained in an extension interface is determined by the kinds of interactions that are needed between the extension and the legacy system. The main purpose of such an interface is to describe the entry points that should be provided by an extension, and thus imported by the legacy system. Extensions may, however, need some information from the legacy system itself. Examples include a means of accessing and affecting system state information and mechanisms provided by the legacy system that the extension should use. Thus, the interface should also describe the legacy-system entry points that are available to extensions.

Supporting an extension interface in a legacy system requires some modification of the legacy code. Importing an extension requires at least adding invocation of the extension entry points. When the extension replaces an existing functionality, then the existing implementation of this functionality must also be removed. Legacy code may also need to be modified to take into account the result of invoking the extension. Finally, exporting information from the legacy system to an extension may require wrapping this information for external use. These modifications may require low-level rewriting of the legacy code.

Figure 1 shows the language used to describe rewrite rules. We explain the main features of this language. Because we assume that only the legacy system requires re-engineering, a rule is indicated to apply either on importing the associated object to the legacy system or on exporting the associated object from the legacy system. A rule describes a modification either to a function body (rewrite), possibly restricted to a specific function, or to a function header (rewrite_hdr). A rule applying to a function body either replaces (`->`) code matching a pattern, or inserts code before (`-B>`) or after (`-A>`) the code matching a pattern. In the case of replacement, the keyword `ALL` indicates that all code matching the pattern is affected by the rule at once; in other cases, each construct matching the pattern is treated individually. The right-hand side of a rule (modification) indicates the code to be inserted by the rewrite rule. A rule applying to a function header can change the name of the function or its parameter list.

Patterns are described using formulas of a variant

```
start          ::= import: rule* | export: rule*
rule           ::= in fn_name : rewrite | rewrite | rewrite_hdr
rewrite        ::= ALL(pattern) -> modification |
                   pattern -> modification |
                   pattern -B> modification |
                   pattern -A> modification
pattern        ::= local_pattern | path_pattern
local_pattern  ::= TEST(local_pattern) | fn_name(args) | C-code
                   pattern & pattern | pattern | pattern |
                   !pattern | pattern => pattern
path_pattern   ::= EX(pattern) | AX(pattern) | EF(pattern) |
                   AF(pattern) | EG(pattern) | AG(pattern) |
                   E(pattern U pattern) | A(pattern U pattern) |
                   EX-back(pattern) | AX-back(pattern) |
                   EF-back(pattern) | AF-back(pattern) |
                   EG-back(pattern) | AG-back(pattern) |
                   E-back(pattern U pattern) |
                   A-back(pattern U pattern)
modification   ::= fn_name(arg,...,arg) | C-code
arg            ::= args | ID | C-expression
rewrite_hdr    ::= fn_name(params) -> fn_name(decl, ..., decl)
decl           ::= params | C-declaration
```

Figure 1: The rewrite rule language

of the temporal logic CTL [9], allowing the description of complex, context-sensitive properties. A pattern is either a local pattern or a path pattern. A local pattern describes properties of a single C statement or expression. The keyword `TEST` indicates a pattern that is restricted to conditional test expressions. The keyword `args` in a function call pattern matches any set of arguments. A local pattern can combine other patterns using *e.g.*, the standard conjunction and disjunction operators. Path patterns use the path operators of temporal logic to describe properties of the paths leading to the matched construct in a control-flow graph. Two variants of each operator are provided. Operators not containing the string `back` describe paths beginning at the matched construct, while the operators containing the string `back` describe paths ending at the matched construct. These operators allow describing a statement in terms of its relationship to other statements.

The rewriting engine not only applies the rules but also gives an error when the patterns described by the rules are not found. This feedback makes the approach resilient to minor differences between related versions of the legacy code.

3 A Linux Scheduling Interface

Our first example is a component-based implementation of the Bossa scheduling framework in Linux 2.4. Bossa allows the extension of a legacy OS with a hierarchy of scheduling policies. As shown in Figure 2, the implementation is structured using three kinds of components: an OS kernel component, a run-time system (RTS) component, and a component for each scheduling policy (only one policy is shown in the figure). The RTS manages the interaction between the kernel, which is fixed, and the policies, which are loaded dynamically. This structure increases the flexibility of the extension framework, allowing the use

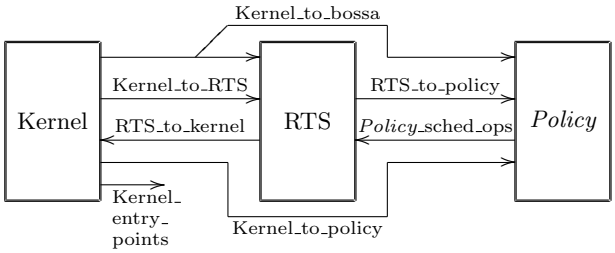

Figure 2: Bossa component structure

```
bundletype RTS_to_kernel =
{ rts_clocktick            // event notification
    {import: in update_times:
      calc_load(args) -A> rts_clocktick()},
  rts_schedule
    {import: in schedule:
      ALL(AF-back(spin_lock_irq(&runqueue_lock)) &
          AF(spin_unlock_irq(&runqueue_lock))) ->
      ... rts_schedule(caller) ...
     schedule() -> bossa_schedule(ID)
     schedule(void) -> bossa_schedule(int caller)},
   ...
  rts_blocked              // scheduler status
     {import: in cpu_idle:
       TEST(!current->need_resched) -> rts_blocked()}}

bundletype Kernel_to_bossa =
{ errno, kfree, kmalloc, sprintf, printk, ... }

bundletype Kernel_to_RTS =
{ add_timer, del_timer, mod_timer, jiffies, ... }

bundletype Kernel_to_policy = { panic, pidhash }

bundletype Kernel_entry_points = { .... }

unit Kernel = {
  imports [ rts_to_kernel: RTS_to_kernel ];
  exports [ kernel_to_bossa: Kernel_to_bossa,
            kernel_to_rts: Kernel_to_RTS,
            main: Kernel_entry_points ];
  depends { };
  files { ... }; }
```

Figure 3: Excerpts of the Bossa interface

of an RTS component that addresses special needs, such as safety checks that depend on the level of trust in the policies. The RTS and the policies are constructed for use with the Bossa framework, and thus implement the appropriate interfaces. We concentrate on the interface of the kernel component, as this component is made from legacy code. Excerpts of this interface are shown in Figure 3.

In the Bossa framework, the extension behavior is represented as a set of functions exported by the RTS to the kernel. These functions, listed in the `RTS_to_kernel` bundletype, allow the kernel to signal relevant state changes (*e.g.*, the blocking of a process) to the policies and to request the election of a process. In some cases, the Bossa behavior is orthogonal to the existing Linux behavior, and invocation of the corresponding RTS function must simply be added at the point where the associated state change has occurred. An example is `rts_clocktick`, which notifies Bossa of the passage of time. The corresponding rewrite rule adds invocation of this function to the function `update_times` of Linux, which itself is invoked on expiration of the Linux periodic timer. In other cases, the Bossa behavior must replace the existing Linux behavior. An example is `rts_schedule`, which implements process election. Linux process election is implemented as a part of the function `schedule`, which also performs the context switch. The rewrite rule associated with `rts_schedule` replaces the fragment of `schedule` between the initial obtaining and the final releasing of the runqueue lock, which contains the legacy process election code.

In addition to importing functions from the RTS, the kernel must also export to the RTS and the policies functions implementing a number of mechanisms, listed in the bundletype `Kernel_to_bossa`. These include generic mechanisms such as locks, memory allocation, file manipulation, as well as scheduling-specific mechanisms such as access to the Linux representation of the current process. Kernel mechanisms used only by the RTS are exported via the bundletype `Kernel_to_RTS` and kernel mechanisms used only by the policies are exported via the bundletype `Kernel_to_policy`. Finally, to allow an encapsulated kernel to interact with the boot loader, the kernel also exports a set of entry points (bundle type `Kernel_entry_points`).

The complete kernel interface for Linux uses 23 rewrite rules. Based on this interface, we have implemented a variety of scheduling policies, including the scheduling policy of Linux, Earliest-Deadline First, and Borrowed Virtual Time (BVT) scheduling [4].[1] We have constructed interfaces for Linux 2.4.18 and Linux 2.4.24. In general, the same rewrite rules can be used for both systems. The main change from Linux 2.4.18 to Linux 2.4.24 that is relevant to Bossa is the introduction of a `yield()` function that allows kernel functions to use the functionality of the `yield` system call. This structure implies that a code pattern indicating a state change relevant to Bossa crosses function boundaries, which is not supported by the rewrite rule patterns. In this case, an error rule indicates an incomplete match of an expected pattern, isolating for the user the part of the kernel code that must be considered by hand.

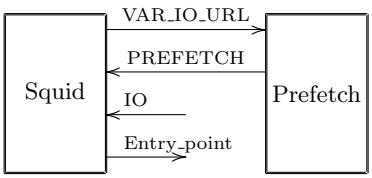

Figure 4: Squid prefetching component structure

```
bundletype VAR_IO_URL = { wrappedUrlParse, ... }
bundletype PREFETCH = { retrieveLinks, ... }
bundletype IO = { printf, puts ... }
bundletype Entry_point = { ... }

unit Squid = {
  imports [ prefetch: PREFETCH, io: IO ];
  exports [ var: VAR_IO_URL, main: Entry_point ];
  depends { };
  files   { ... };
}

unit Prefetch = {
  imports [ var: VAR_IO_URL ];
  exports [ prefetch: PREFETCH ];
  depends { exports needs imports };
  files   { prefetchingModule.c };
}
```

Figure 5: Squid prefetching interface

4 A Squid prefetching interface

Our second example is a component-based framework for extending Squid with prefetching policies. As illustrated in Figure 4, the framework uses two kinds of components: Squid itself, and components for the prefetching extensions (only one extension is shown in

[1] `http://www.emn.fr/x-info/bossa/`

the figure). Squid should invoke the extensions when certain runtime events related to page loading occur. An extension selects the pages to be prefetched, *e.g.*, by parsing the current page and identifying relevant links, and adds these pages to the Web cache. Excerpts of the extension interface are shown in Figure 5. The interface consists of two units, `Squid` and `Prefetch`, describing the functions imported and exported by Squid and the prefetching extension, respectively.

The main function exported by a prefetching extension is `retrieveLinks`, which analyzes Web pages in order to decide which pages to prefetch. Squid must be modified to invoke this function whenever it retrieves a page in response to a client request, and thus this function is part of the `PREFETCH` bundletype imported by the `Squid` unit. The rewrite rule associated with `retrieveLinks` modifies the Squid function `comm_write` to invoke the extension. The function `comm_write`, however, is used in response to all kinds of requests, including those from the extension itself. To avoid infinite looping, the rewrite rule associated with `retrieveLinks` also modifies the callers of `comm_write` to indicate the context from which `comm_write` is called. This added information enables `comm_write` to only invoke the extension when it is called from a client request.

A prefetching extension only decides when and what to prefetch; to actually retrieve and store the chosen pages, it uses the mechanisms of Squid. These mechanisms are exported by the `Squid` unit via the `VAR_IO_URL` bundletype. In many cases, the Squid functions take as an argument a data structure of which only some fields are relevant when the function is invoked from an extension. Thus, rewrite rules are used to create wrapped versions of these functions in Squid. In all, 25 Squid functions are exported to the prefetching extension.

The complete interface uses 4 instrumentation rules that are applied at about 40 call sites, scattered across 1000 lines of source code. We have used this interface to extend Squid with the prefetching strategy, "top 10 prefetching" [2].

5 Related Work

OSKit also uses componentized versions of legacy OS code [6]. The goals, and thus the components involved, are different, however. In OSKit, the goal is to enable the construction of various fixed OSes from a set of building blocks implementing basic OS functionalities. Thus, individual OS services, such as file systems, device drivers, etc., are provided as components. Our goal is to enable extension, *i.e.* a form of external interaction, of an existing OS. Thus, the complete OS forms a component, modified to enable new interactions.

Aspect-oriented programming (AOP) is a program design technique in which a functionality that crosscuts an application is isolated in a module, known as an aspect. An aspect contains both code fragments that implement the functionality and a formal description of how these code fragments should be integrated into the application. Our augmented interface specifications amount to aspects dedicated to the construction of a component interface in legacy code. AOP has also been used by Coady to describe OS evolution [3].

6 Conclusion

In this paper, we have considered the problem of providing support for the extension of legacy systems. We propose that extension be based on an interface that makes explicit the resources that an extension provides and the resources that it reuses from the legacy system. To allow extension that is not anticipated by the legacy code, we propose to augment the declarations in such an interface with rewrite rules that describe how to modify the legacy system to support the extension interface. Finally, we propose the use of tool support to carry out these modifications automatically.

Our approach enables reuse and evolution of an extended system. The contents of the interface can guide the development of multiple extensions, by making explicit the set of entry points that are needed for a complete implementation. The rewrite rules and automated tool support ensure that the difficult task of re-engineering the legacy code to support extension does not have to be repeated for each extension that is to be implemented. The tool support additionally facilitates evolution of existing extensions to new versions of the legacy system.

In our experiments, we have used Knit, which only supports static component composition. In the context of a long-running system such as an OS, it can be useful to load extensions dynamically, to adjust to changing needs. The introduction of rewrite rules should be straightforwardly applicable to other component models, such as Think, that are directed to dynamic composition [5]. While legacy code can be

prepared statically for subsequent dynamic component composition, it could also be useful to modify the legacy code dynamically, *e.g.*, when an extension is first loaded. We plan to investigate the use of our dynamic rewriting tool μ-Dyner [13] in this context.

Acknowledgments We thank Anne-Françoise Le Meur for comments on this paper. This work was supported in part by a Microsoft Embedded research grant and by ACI CORSS.

References

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montreal, Canada, Oct. 2003. IEEE.

[2] K.-I. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[3] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 50–59, Boston, MA, Mar. 2003.

[4] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 261–276, Kiawah Island Resort, SC, Dec. 1999.

[5] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *2000 USENIX Annual Technical Conference*, pages 73–86, Monterey, CA, June 2002.

[6] 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 (SOSP'97)*, pages 38–51, Saint-Malo, France, Oct. 1997.

[7] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 267–282, Monterey, CA, June 1999.

[8] V. Issarny, M. Banâtre, B. Charpiot, and J.-M. Menaud. Quality of service and electronic newspaper: The Etel solution. *Lecture Notes in Computer Science*, 1752:472–496, 2000.

[9] D. Lacey and O. de Moor. Imperative program transformation by rewriting. In R. Wilhelm, editor, *Intl. Conf. on Compiler Construction (CC)*, volume 2027 of *Lecture Notes in Computer Science*, pages 52–68, Genova, Italy, 2001.

[10] J. Lawall, G. Muller, and L. P. Barreto. Capturing OS expertise in a modular type system: the Bossa experience. In *ACM SIGOPS European Workshop 2002 (EW'2002)*, pages 54–61, Saint-Emilion, France, Sept. 2002.

[11] K. Magoutis, J. C. Brustoloni, E. Gabber, W. T. Ng, and A. Silberschatz. Building appliances out of reusable components using Pebble. In *Proceedings of the ACM SIGOPS European Workshop*, pages 211–216, Kolding, Denmark, Sept. 2000.

[12] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 347–360, San Diego, CA, Oct. 2000.

[13] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 110–119, Boston, MA, Mar. 2003.