Specialization Classes:
An Object Framework for Specialization *

Crispin Cowan, Andrew Black, Charles Krasic,
Calton Pu, and Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology

Charles Consel and Eugen-Nicolae Volanschi
University of Rennes / IRISA
(synthetix-request@cse.ogi.edu)

September 6, 1996

1 Introduction

Specialization is a growing area of interest in the operating systems community. OS components specialized to some particular circumstance can offer enhanced performance, functionality, or both. Complimentary partial evaluation techniques for automatically specializing programs are also reaching maturity. However, the problem of managing specialization remains: how to specify a specialization, when to apply it, and when to remove it. This problem is particularly important for long-running programs such as operating systems, where specializations are likely to be temporary.

This paper presents an object-oriented framework for specifying specializations in long-running programs such as operating systems. This model is based on the following concepts:

- Inheritance allows replacement implementations of of member functions. We thus use a graph of sub-classes to specify a set of potential specializations of a given facility by replacing generic implementations with specialized implementations.

- Specializations in long-running programs are temporary, because the particular circumstances that permit the use of a specialized implementation are likely to change eventually. We thus support *temporary* and even *optimistic* specializations [16].

- Ensuring that it is valid to use a specialized implementation can be more difficult than creating the specialized implementation [16]. We thus use a formal method to specify when a specialization is valid. This lets us automatically detect when specialization circumstances have changed [8], and also automatically generate specialized implementations using partial evaluation [6, 5].

Section 2 describes our specialization model, which is applicable both in OO operating systems and in legacy kernels. Section 3 describes compilation techniques for this model. Section 4 briefly describes some closely related work, and Section 5 concludes this position paper.

2 Specialization Classes

We first describe our model using an example, and then explain some details. Figure 1 illustrates specialization of a file system: the open file object FS, which understands the operations `read()` and `write()`, is said to be the target of the specialization.

Following modern usage [1, 14], we use the term *type* to refer to the interface exposed by an object and the term *class* to refer to the method code and the instance variables that implement that interface.

*This research is partially supported by ARPA grants N00014-94-1-0845 and F19628-95-C-0193, NSF grant CCR-9224375, and grants from the Hewlett-Packard Company and Tektronix.

Hence, the type of the file describes the fact that it can be read and written; in an OO system the type is merely the type of the FS object, and in a legacy OS coded in a non-OO language it is the type signature of the set of procedures that provides the file system functionality.

The *specialization plan* is a definition of all the ways in which the file system can be specialized. In each specialization, some of the methods of the target are replaced by various specialized implementations. The methods specialized by the specialization plan are the set of *specializable functions* that are replaced by various specialized implementations. Thus the specialization plan encapsulates the specializations to be applied to the system, independent of the degree of encapsulation provided by the system's source language.

The various specialization options within a plan are organized into a partial order of *specialization classes* according to the relation "more specialized than." Each specialization class adds some degree of specialization to the classes it inherits from, e.g. NFS is a specialization of generic, and NFS/exclusive is a specialization of both NFS and exclusive. Each specialization class describes a *specialization state* that the specialized facility can achieve. The "generic" - specialized state is the unique top of the partial order of specialization classes.

Each specialization class specifies the *conditions* that make the specialization applicable, and a subset of the members in the specialization plan to be replaced with specialized methods. The conditions of a specialization class imply the conditions of each of its parents. The truth of the conditions can change over time, and thus must be monitored as described in Section 2.1.

Specialization plans are compiled into specialized object generators, which when `new`'d create specialized objects as shown in Figure 1. A specialized object is a wrapper around the object being specialized. The specialized object represents the state of an instance of a specialization plan, i.e., bindings from the values in the conditions to data in the target, and bindings from the specializable functions to the specialized methods. We view the *type* of the target object as being unchanged by the specialization; from the point of view of the client, the same set of messages is understood, and they have the same effects. Thus, the type of the specialization object is statically determined by the type of the target.

In contrast, the *class* of the object changes dynamically according to the truth of the conditions, and causes changes in the method code bound to the specializable functions. Looking a little more closely, it may in fact the the case that the type changes: for example, if the conditions indicate that a certain message will never be sent, we might create a specialized object that eliminates that method altogether! However, our methodology guarantees that any such changes in type will be invisible to the client.

2.1 Conditions: Quasi-Invariants

Conditions specify *invariants*. A *true invariant* is a classical invariant: a property of the system that is guaranteed to be true at all times, stated as an expression using system variables that must evaluate to "true." A *quasi-invariant* is a property that is *likely* to remain true, but may become false at some future time. Specifying conditions using invariants allows the following key steps in the specialization process to be automated.

Invariants can be used by partial evaluators to automatically prepare a specialized implementation that has been optimized using the invariants. Our use of invariants for specialization was originally inspired by the invariant input specification for Tempo [6, 5], a powerful partial evaluator for C. Partial evaluation to exploit specialization gives us a formal relationship between the conditions and the optimized implementation.

Partial evaluation is independent of whether a condition is an invariant or a quasi-invariant. However, specializations that depend on quasi-invariants are not always valid, but instead depend on some temporary circumstance that begins when the quasi-invariants become true, and ends when the quasi-invariants become false. For instance, file system access can be optimized using a quasi-invariant that the file is not shared [16], but this condition can change unexpectedly if a separate process opens the file.

Our hand-specialization experiments showed that locating all components of the kernel that affect the state of quasi-invariants can be more difficult than the task of crafting specialized implementations. We have thus developed tools for locating kernel components that can potentially invalidate quasi-invariants, described in the following section.

2.2 Guarding for Changes in Quasi-Invariants

We have developed two ways to locate kernel components that can potentially alter quasi-invariant state. One is based on type-checking the kernel source code,

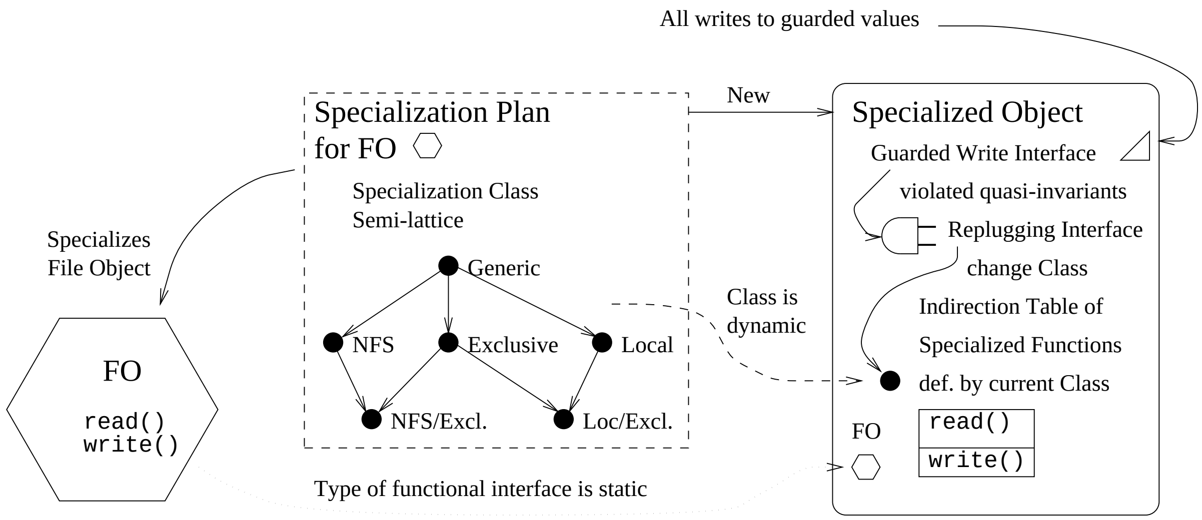

Figure 1: Example: Specialization of a File Object

and the other is based on fine-grained virtual memory protection. These techniques are discussed at length in [8], but what they produce is a list of kernel source code statements that may violate quasi-invariant state. These writes to quasi-invariant state must be *guarded*.

However, frequently such statements are accessing heap-allocated data structures, and only a few of many of these structures actually control a specialization, e.g. the quasi-invariant `inode.refcount == 1` may be true of some particular `inode`, but there are thousands of instances of the `inode` struct in the running kernel. The guards placed around writes decide whether the write is to an actual quasi-invariant, or only a write to a value of the same type as a quasi-invariant.

We distinguish among structs of the same type between those that contain quasi-invariant terms and those that do not by inserting a *Specialization IDentifier* field (SID). In the case that the `inode` struct is the instance referred to in the quasi-invariant expression, the SID field points to the specialized object that depends on that quasi-invariant.[1] The specialized object then performs the guarded write. For example, consider this update to `inode.refcount`:

```
inode.refcount = some_value;
```

A guarded update of the `inode.refcount` would be written as:

```
inode_set_refcount(some_value, SID);
```

The `inode_set_refcount` function writes the `inode.refcount` field in any case, but also atomically adjusts any specialized components that depend on quasi-invariant expressions that depend on this `inode.refcount` value.

2.3 Responding to Quasi-Invariant Changes: Replugging

When a quasi-invariant is violated, the specialized object must adapt its specialized implementation of the facility to the new circumstance without relying on the quasi-invariant. One very common action to be taken by the specialized object is to replace the dependent specialized components with other, differently specialized components, or with generic components. This replacement is called *replugging*, and requires fast, safe, concurrent dynamic linking. The problem is to facilitate very low latency execution of a function via an indirect function pointer, while concurrently allowing the pointer to be changed. Locks could be used, but locks may also substantially degrade performance. In [7], we describe a portable algorithm that supports low-latency invocation of replaceable functions while allowing concurrent update of pointers to those functions.

[1] A more complex scheme is used when `struct` instances are shared among multiple specializations, which we omit for simplicity.

3 Translation & Specialization

Our previous efforts have manually applied our various specialization tools [7, 8, 16, 17]. Automatic translation of specialization plans should convert the high level specification of how to specialize the system into running code that integrates the various components.

3.1 Specialization Plans

The specialization plan describes all possible ways in which the facility can be specialized. Given a list of quasi-invariants, there is an exponential number of combinations of such invariants, resulting in an exponential number of specialized functions. Specialization classes allow the programmer to specify *which* combinations are important, and thus should be exploited.

The specialization plan is translated into a code template for a specialized object, and two lists. The code manages the data structures described in Figure 1. The lists describe each specialization class, and are fed to other specialization tools as follows:

specializable functions — The list of specializable functions is taken from the specialization plan and built into the specialized object, and is fed to the Tempo partial evaluator (see Section 3.2).

quasi-invariants — The list of quasi-invariants is fed to the guarding tools, and to Tempo.

3.2 Partial Evaluation

A specialization class declares an opportunity for specialization, and is described by a list of (quasi-)invariants. If all the predicate conditions are of the form `variable = const_value` or `struct.field_name = const_value`, the specialized implementations can be automatically derived by a partial evaluator. Notice that such an automatic tool could be extended to deal with other classes of predicate conditions, e.g. of the form `variable < const_value`. If the complexity of the predicates is beyond the current capabilities of the partial evaluator, the programmer can still provide a hand-written implementation.

We are using Tempo, a partial evaluator for C programs developed at IRISA, [5, 6, 4]. Given a program and part of its inputs, it generates a specialized version of the program in which all the computations depending on the known inputs are performed. Tempo processes a program in two phases.

First, an analysis is performed, to decide which parts of the program are to be *reduced* (eliminated), and which other are to be left in the specialized program. Note that the analysis phase doesn't need the concrete values, it just propagates the known/unknown information. The interface to this first phase is the *analysis context*, which contains:

- a list of the known inputs, which can be either variables or `struct` field names
- a list of the functions to be specialized

In a second phase, the program is specialized, based on the annotations produced by the first phase and some concrete values for each known input previously declared. The interface to this second phase is the *specialization context*, binding an actual value to each invariant variable.

4 Related Work

Object-oriented OS research has advanced the state of the art in the interface provided to applications, and advanced the ability of operating systems to be dynamically configured. In particular, Choices [2, 11], AL-1/D [15], and Apertos [18] have investigated ways in which object-orientation can be used for OS re-configuration. Kiczales has been exploring the general question of how objects can be used as a *meta-interface* [13].

OS customization has also been studied outside the OO community. The SPIN project allows replacement OS components to be loaded into the kernel. SPIN uses a combination of static type checking and run-time checks to bound the damage potential of replacement components, but leaves the correctness of applying a specialization up to the application. The Aegis project provides more customizability by placing most OS functionality in a user-level library attached to user applications [10]. We discuss some of these approaches in [9].

At the language level, specialization classes are similar to Chambers' predicate classes [3], which allow, for example, the class of a buffer object to depend on whether the buffer is full, partially-full, or empty. Specialization classes can be thought of as an implementation of predicate classes in which guarding is used to change the class of an object in response to independent, concurrent events; this idea is hinted at in reference [3], but was not fully worked out or implemented. Specialization classes can also be applied to systems written in a language such as C, in which the objects are more conceptual than real.

Specialization plans are similar to the Aster distributed application configuration language [12]. Aster operates at a higher level, using predicates that cannot be checked mechanically, but can be reasoned about mechanically.

5 Future Research

We have proposed an object-oriented, mostly declarative model for specifying specializations in long-running programs such as operating systems. In the near term, we expect to demonstrate the utility of this programming model for enhancing flexibility and performance in operating systems through specialization. Subsequently, we hope that this model will prove itself to be a valuable addition to the family of modularity techniques.

References

[1] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, pages 65–76, January 1987.

[2] R. H. Campbell, N. Islam, and P. Madany. Choices: Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[3] C. Chambers. Predicate Classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserstautern, Germany, July 1993.

[4] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[5] C. Consel, L. Hornoff, J. Noye, F. Noël, and E.-N. Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. In *International Workshop on Partial Evaluation*, Dagstuhl Castle, Germany, February 1996. Springer-Verlag LNCS.

[6] C. Consel and F. Noël. A general approach to run-time specialization and its application to C. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburgh Beach, FL, January 1996.

[7] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[8] C. Cowan, A. Black, C. Krasic, C. Pu, and J. Walpole. Automated Guarding Tools for Adaptive Operating Systems. Work in progress, December 1996.

[9] C. Cowan, J. Walpole, A. Black, J. Inouye, C. Pu, and S. Cen. Adaptable Operating Systems. In R. Campbell and N. Islam, editors, *Modern Operating Systems Research*. IEEE Computer Society Press, 1996. To appear.

[10] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[11] A. Gopal, N. Islam, B.-H. Lim, and B. Mukherjee. Structuring Operating Systems using Adaptive Objects for Improving Performance. In *Proceedings of the Fourth International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*, pages 130–133, Lund, Sweden, August 1995.

[12] V. Issarny and C. Bidan. Aster: A Framework for Sound Customization of Distributed Runtime Systems. In *16th International Conference on Distributed Computing Systems (ICDCS'96)*, pages 586–593, Hong Kong, May 1996.

[13] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[14] W. LaLonde and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5), January 1991.

[15] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In A. Yonezawa and B. C. Smith, editors, *Proceedings of the International Workshop on New Models for Software Architecture '92, Reflection and Metalevel Architecture*, pages 36–47, Tokoyo, Japan, November 4-7 1992.

[16] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[17] E.-N. Volanschi, G. Muller, and C. Consel. Safe Operating system Specialization: The RPC Case Study. In *Proceedings of the First Annual Workshop on Compiler Support for System Software*, Tuscon, AZ, February 1996.

[18] Y. Yokote, G. Kiczales, and J. Lamping. Separation of Concerns and Operating Systems for Highly Heterogeneous Distributed Computing. In *Proceedings*

of the European ACM SIGOPS Workshop, September 1994.