It's Your Choice - On the Design and Implementation of a Flexible Metalevel Architecture

Chris Zimmermann Vinny Cahill

Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2, Ireland
{czimmerm, vjcahill}@dsg.cs.tcd.ie

February 15, 1996

Abstract

Traditional configurable operating systems typically provide a fixed and limited set of functionality. We propose a metalevel architecture, where application-defined objects can choose from a rich selection of possible configurations and are therefore able to dynamically change the way in which they are executed to the configuration that suits best. This allows applications to adapt operating system behaviour to even unanticipated requirements during run-time. A distributed application which processes multimedia data serves as an example to illustrate the concepts described.

1 Introduction

Traditional configurable operating systems typically provide a fixed and limited set of functionality. With the advent of a new generation of applications, such as multimedia systems, where applications must be able to adapt to changing requirements during run-time, such limited operating system support may be a hinderence to maintaining the level of efficiency that these operating systems were configured for. Since the applications have to be adaptable, operating systems in turn must be able to support adaption during run-time. Besides traditional functionality such as scheduling and storage management, this means offering interfaces that allow applications to control the way in which an operating system provides this functionality [KL93, KTW92]. It is no longer sufficient for an operating system just to offer scheduling functionality, for example, without providing a means of changing the way in which it does this scheduling, i.e. of choosing individual scheduling policies. Another example is persistence: should the size of objects, which are persistent, change during the life-time of the application, the operating system has to adapt its mechanisms and policies to provide the most efficient solution for a given object size [ZK93].

This paper presents a solution to this problem. We facilitate this adaptation process by employing a reflective architecture, where application-defined objects (the units of computation in our model) are able to choose from a rich selection of possible configurations and are therefore able to dynamically change the way in which they are executed to the configuration that suits best.

The remainder of this paper is organized as follows: after this introduction we discuss the underlying operating system framework named Tigger which serves as an implementation basis for the concepts presented in this paper. We use an example from the field of multimedia computing to illustrate the use of this architecture. Sections on implementation issues with some performance figures together with a discussion of related work conclude this paper.

2 Tigger

Tigger is an object-support operating system framework [CHJ+94] which serves as the environment for implementing the concepts presented in this paper. This framework provides typical operating system services such as persistence and scheduling using objects as the basic units of computation. A general feature of this framework is its ability to be tailored to the specific needs of different target environments.

Interactive video games provide one possible target environment for the Tigger framework. Since these video games are highly interactive, they have requirements in terms of support for (soft) real-time object behavior. This support then has to be provided by the underlying instantiation of Tigger. Therefore, an instantiation of the framework that is aimed at this target environment has to provide this real-time support by means of real-time scheduling of threads as well as real-time synchronization protocols. In contrast, the requirements in terms of persistence are constrained to simple services such as transferring an object state from and to disk if indeed persistence is needed at all.

In addition to these general requirements, individual

0-8186-7395-8/96 $05.00

applications which are part of these environments have their own demands in terms for specific algorithms or mechanisms which the underlying framework has to supply. A video game using multimedia data such as a video clip, for example, may need a particular real-time scheduling mechanism, depending on the type of multimedia data it has to process. These requirements may even vary during the run-time of this application.

The functionality of a particular instantiation of Tigger is subdivided into single components or subframeworks responsible for implementing specific object support services. These subframeworks can then be configured to meet the needs of specific application areas such as video games.

3 Structuring the Metalevel

From an application-oriented point of view the functionality supplied by an instantiation of Tigger is structured as depicted in Fig. 1. This overall architecture named Piglet is detailed in the following sections.

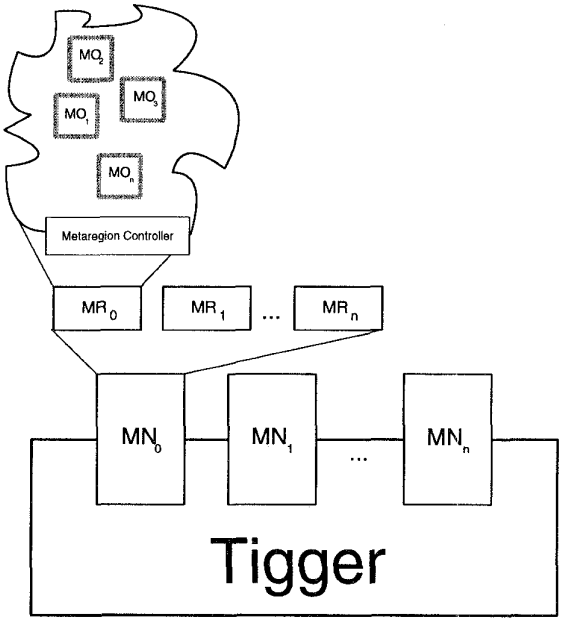

Figure 1: Metalevel Structure of Tigger

3.1 The Metalevel

The basic units of computation in our model are objects. To control the way in which these objects are executed, we employ a concept called *metaobjects* [Mae87]. Since these metaobjects know about the internal organization and structure of application-defined objects[1], they are able to control certain aspects of the execution of those baselevel objects. The entirety of all metaobjects is called a *metaspace*.

Piglet, the metalevel architecture of Tigger, structures this metalevel into three groups of entities which represent the nature of the underlying instantiation of a Tigger.

3.2 Metanuclei

As discussed above, Tigger is a collection of subframeworks responsible for implementing various algorithms dealing with single aspects such as process management. Towards applications, the contents of each of these subframeworks are represented by *metanuclei*. Metanuclei are a means for grouping all the mechanisms and policies regarding a certain functionality that a particular instantiation of Tigger supports. Take process management as an example. A process management metanucleus for real-time support includes real-time scheduling policies and synchronization protocols. Metanuclei therefore contain the relevant code for one specific operating system functionality.

3.3 Metaregions

Each metanucleus consists of a set of metaregions. A metaregion in turn consists of a (possibly empty) set of metaobjects implementing the same functionality and exposing the same interface to application-defined objects. The difference between the individual metaobjects forming a metaregion is their behavior when providing this functionality. For example a scheduler metaregion being part of a real-time metanucleus consists of different scheduler metaobjects realizing different real-time scheduling algorithms. They all exhibit the same functionality (i.e. scheduling) but do it differently according to the algorithm that they implement. Another example is a metaregion containing different pagers realizing object persistence. Pagers aimed at different object sizes all do the same thing: transferring the state of objects to and from disk. The way they achieve this is different depending on the size of the object.

In addition to the individual metaobjects, each metaregion has a controller which allows metaobjects to be attached to objects and allows a metaregion to be queried about its contents and the capabilities of the individual metaobjects. This part of the interface is used by objects to control which metaobject of the metaregion is controlling a certain aspect of its behavior. Besides this controller interface, each metaregion defines a common interface for the metaobjects this metaregion contains. This common interface of the metaobjects[2] is then used by applications to change

[1] Also called *baselevel* objects—or baseobjects for short [Zim96a]—to emphasize the difference between base- and metalevel.

[2] The interface of a metaobject is also known as a metaobject protocol (MOP) [KdRB91, Zim96b].

the way in which they are executed. To continue the scheduler example from above, methods provided by this scheduler MOP include `yielding` the CPU and `waiting` for another thread to finish; these methods all have to be implemented by the individual metaobjects which are part of the metaregion.

Compared to traditional approaches to organizing metaobjects such as the multi-model reflection framework [OIT93] or Apertos [Yok93], metaregions have two distinct advantages. First of all, a metaregion can capture a common part of the metaobjects that it contains. For example consider support for thread management to be provided by a scheduler metaregion. Since each scheduler metaobject has to manage queues containing various threads (such as queues for active or blocked threads), this common functionality can be implemented by the metaregion.

This can easily be done when an object-oriented programming language supporting inheritance such as C++ [Str92] is used to implement metaregions. The individual metaobjects are then derived from a baseclass providing the common interface of the metaregion.

The second advantage is that metaregions introduce a *statically typed* metalevel. One of the advantages of this static typing is that the source code can be compiled into machine code instead of being interpreted at run-time [CW85, DT88]. Although not all binding decisions can be made at compile-time, more efficient mechanisms such as virtual function tables [ES91] can be employed which result in faster overall execution of the program compared to just interpreting it at run-time. A detailed discussion of metaregions can be found elsewhere [Zim96a].

3.4 Metaobjects

Metaobjects implement the non-algorithmic behavior of objects, i.e. they control the way in which these objects are executed. The responsibilities of the metaobjects are twofold: on one side they are in charge of supplying the implementation of the MOP representing the metaregion as explained above; on the other side they intercept individual method calls to the baseobjects transparently. This allows the functionality associated with the metaobjects to actually take place.

Take persistence as an example. Suppose a programmer wants to maintain the state of an object beyond the run-time of a particular program that this object is part of. He or she does so by telling the metaobject that is in charge of implementing persistence that the state of the baseobject is now persistent by calling a suitable method provided by of this metaobject. When this object is about to be destroyed (for example at the end of the run-time of the application), the metaobject intercepts this destruction process in order to save the state for the object to secondary storage.

4 An Example

This section uses an example of a multimedia baseobject to explain the concepts discussed above. This

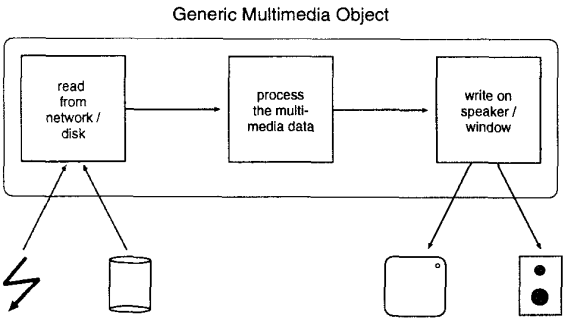

Figure 2: A Baseobject Processing Multimedia Data

object consists of three internal building blocks as depicted in Fig. 2. A source side gathers the data (either from a network connection or from secondary storage) which is then processed and displayed on a screen or output to a speaker. Examples of multimedia data include audio data streams which are filtered by the processing stage or video data streams encoded according to MPEG [Gal91] which are decompressed by the processing stage.

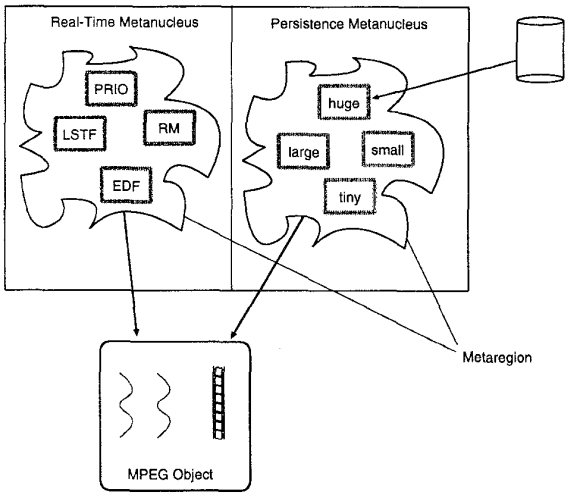

Figure 3: The Metalevel Structure of the Multimedia Object

This multimedia object uses an active object model allowing multiple threads to be active inside the object at any point in time. This baseobject employs two metanuclei: one for real-time functionality and one for persistence (see Fig. 3). Each of these metanuclei

contains one metaregion[3].

The real-time metanucleus contains a metaregion responsible for scheduling the threads associated with the different active object models (different models for active objects can be identified depending on when a thread is associated with a particular method of the object [YT89]). These schedulers implement different real-time scheduling algorithms such as priority-based scheduling (PRIO), Least Slack Time First (LSTF) [DM89], Rate-Monotonic (RM) and Earliest Deadline First (EDF) [LL73].

The persistence metanucleus contains a metaregion providing pagers supporting different object sizes. These pagers are responsible for transferring object states to and from secondary storage, and all employ different algorithms for doing so depending on the object size at which they are targeted. To be most efficient when dealing with small object sizes, the relevant metaobject groups small objects onto a memory page before storing this page on disk. In contrast to this, the metaobject aimed at supporting large object sizes may avail of certain features of the storage subsystem such as burst transfer.

To motivate the ability to change an object's behavior—i.e. the way its code is being executed—imagine the following situation. Currently, the object described above is displaying an MPEG-encoded video clip from disk. Depending on the parameters of this video clip such as compression ratio, an EDF-scheduler metaobject has been chosen from the scheduler metaregion to control the method which decompresses the video clip. Since MPEG-encoded video clips tend to be large, it also uses the metaobject aimed at large objects from the pager metaregion.

Now suppose the user on whose behalf this object is executing requests a change of media type. Displaying of the video clip has to be stopped and some audio track should be filtered and played back instead. In order to maintain efficiency, the object has to change both the scheduler metaobject and the pager metaobject. Since audio streams tend to be smaller than video streams, the object now selects a pager metaobject aimed at smaller object sizes. In addition to this, the scheduler metaobject is changed from EDF to a metaobject implementing an RM policy since the time it takes to process the audio data stream does not vary as this was the case with the MPEG-encoded video stream.

The necessary changes of the metaobjects are initiated via an interface provided by the metaregions. To the application programmer, the exchanging of metaobjects and the resulting metaregion-internal actions happen transparently. All he or she has to do to trigger this change is to `detach` the old metaobject from the object and to `attach` it to the new metaobject. Both methods are part of the metaregion controller interface.

$\texttt{query()} \rightarrow (\texttt{mo}_1, \texttt{mo}_2, \ldots, \texttt{mo}_n)$
`attach(metaobject)`
`detach(metaobject)`

Table 1: Generic Interface defined by the Metaregion

Tab. 1 briefly sketches the most important methods defined by the generic part of the metaregion interface. `query` returns a list of the metaobjects that a metaregion defines, `attach` allows an object to link a metaobject to itself, and `detach` finally reliquishes this link again.

5 Implementation

Because two of our main goals for this project are portability and efficiency, we chose C++ as our primary implementation language. C++ will also be our first supported language for application-development making use of our metalevel architecture. Since C++ is a compiled language, a preprocessor will need to be employed to modify the source code in order to make method interception possible [CM93, Chi95]. This modified source code is then compiled using an ordinary C++ compiler and linked with the necessary metalevel support.

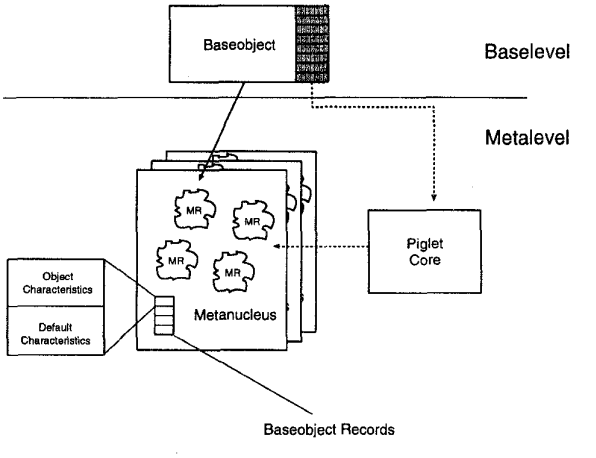

Figure 4: Implementation of the Metalevel Architecture

5.1 Internal Organization

Internally, our metalevel architecture is structured as depicted in Fig. 4. The solid arrow represents an explicit invocation of metaregions which are members of a metanucleus done by an application in order to control which metaobjects are attached to it or simply to

[3]The actual structure of these metanuclei is more complex. Due to space constraints, a detailed discussion is omitted.

call individual metaobjects. The dashed arrows represent the flow of control that takes place during method interception at run-time. When a method which is part of an object (the checked part of the baseobject in Fig. 4) is entered or left, a trap to the generic part of Piglet (named Piglet Core) occurs. This generic part then takes care of distributing the notification of this method call to the individual metaobjects in the different metanuclei. Upon method interception, these metaobjects can take the appropriate action depending on the policy or mechanism they implement.

Internally, a metanucleus maintains a database of baseobject characteristics. Due to the nature of the different metanuclei, these characteristics are highly metanucleus-dependent. To continue the real-time example from above, in addition to the particular real-time scheduling policy which can be derived from the metaobject controlling this baseobject, a record of this database would also contain parameters such as period and deadline of the threads attached to the methods.

Instead of having a metanucleus-wide database of baseobject records, an alternative would be to have the metaregions manage their own object database (perhaps derived from some generic baseclasses). However doing so would cause too much overhead in terms of memory consumption (esp. when there are many metaregions inside a metanucleus) and lookup-time, because then each metaobject would have to do its own lookup upon invocation instead of doing it once on entering a metanucleus (cf. Sect. 5.2).

These object characteristics can be subdivided into two parts: a *default* part which applies to all instances of a template and a baseobject-*specific* part which is individual to this baseobject. A template corresponds to a class in class-based languages such as C++[Str92] or Eiffel [Mey92] or to prototypes in languages based on delegation such as Self [US87]. This allows a two-staged hierarchy: a default value can be applied to all future instances of a template which then can be changed when an instance—a object—is actually created.

5.2 Performance

Tab. 2 gives some performance figures for our prototype implementation of Piglet (all values in microseconds). All measures were conducted using a 90 MHz Pentium-based PC clone with 32 MB of main memory and a second-level cache of 256 kB. To eliminate any effects of a cold cache, only a hundred objects were used. These objects then issued a synthetic load onto the metalevel in terms of method access and thereby trapping to the metalevel.

The first row reports the overhead associated with trapping from the base- to the metalevel. In the current implementation, this reflects the performance loss caused by the modified source code. Basically, this consists of testing a single bit, a conditional branch if no metalevel interception is to take place or an indirect call to Piglet Core if a metaobject requested a metalevel interception (MLI) and eventually an indirect call to the original method. This indirect call is responsible for the sub-optimal performance, because it causes the processor-internal pipeline to stall and issuing of wait-states until the pipeline is re-filled. Instead of wrapping the original method, an alternative would have been to inject the metalevel-trapping code directly into the original method of the object. But this would have caused various problems with the preprocessor so we preferred this simpler but slightly slower option.

Trap into Piglet Core	2.4
Null Metaobject	33.3
Object Lookup	14.4

Table 2: Performance Figures of the Prototype Implementation (all times in μseconds)

In addition to this overhead caused by the wrapping code, the second row gives the actual time it takes to route the flow of control from the base- to the metalevel. Here, a null metaobject which immediately returns after being called gives the overall overhead induced by the metalevel. This figure gives an impression of how big the price is that has to be paid for the benefits that our architecture entails.

The last row gives the time that it takes to perform a lookup of an object description in the database maintained by each metanucleus. Note that this cost occurs only once when the flow of control enters the metanucleus before the MLI is distributed by a suitable dispatcher to the individual metaregions inside the metanucleus. Another option would been to let each metaobject do its own lookup but this would have result in an additional performance penalty because most metaobjects need this information anyway.

5.3 Current Status

As discussed above, a prototype implementation of the metalevel architecture has been built and measured. But just a metalevel architecture on its own without additional functionality is not very useful but merely a proof of concept. Therefore, we are currently in the process of tying a small real-time executive, which is named Roo and is part of the overall Tigger instantiation for real-time support [ZC95], to this metalevel architecture. In addition to the scheduling support discussed above, this real-time executive, which is aimed at the support of soft real-time behavior, provides different real-time synchronization mechanisms and active object models.

In combination with Piglet, this allows applications to reconfigure their operating support environments to their specific needs in terms of support for soft real-time at run-time as discussed in Sect. 4. Connecting different subframeworks for the support of persistence and other operating system services to Piglet is planned for the future.

6 Related Work

The architecture discussed above allows applications to tailor the behavior of an operating system dynamically in order to be able to adapt changing requirements. Traditional operating system frameworks such as Choices [CIJ+91, CIMR93] and reflective systems like Apertos [Yok93, LYI95] typically offer only a very limited choice of mechanisms which they support. In contrast to this, our approach has the potential to provide the programmer with a rich selection of possible algorithms and policies to choose from. This prepackaging of functionality does not necessarily have to be expensive as one might expect. A recent effort concerned with the design and implemention of the process management metanucleus for real-time support showed that using operating system framework techniques result in small run-time overhead but provided a rich set of functionality [ZC95].

Another recent approach to adaptable systems consists of allowing applications to dynamically insert code into an operating system kernel. But as one can imagine, there are several problems associated with this variant; a breach of security and consistency of the overall system is one of them. In order to overcome this security problem, run-time checks have to be inserted, which result in a less efficient system. Our design, however, prevents any breaches of security. By placing metaregions and metaobjects in separate protection domains such as address spaces, we prevent any tampering with the code of metaregions and metaobjects.

Examples for this second flavor of adaptable systems are Spin and Bridge [BSP+95, W+93]. But these proposals tend to either compromise efficiency or are too restrictive regarding the mechanisms a programmer can use when developing kernel-code [B+94, SB94]. For example Wahbe et al [W+93] report up to 12 % fault isolation overhead when trying to prevent any breach of consistency.

7 Conclusion

We presented a metalevel architecture for the dynamic adaption of operating system behavior. By structuring the metalevel into metanuclei, metaregions and metaobjects, we provide a means for the application programmer to select the functionality the current environment requires, thereby catering for the ability to change this functionality should this become necessary. By using the example of a object processing multimedia data, we motivated the necessity for an adaptable application support environment.

We implemented our architecture using object-oriented design and mechanisms. Performance figures of a prototype implementation give some impressions of the costs which are involved using the above architecture.

References

[B+94] Brian N. Bershad et al. SPIN—An Extensible Microkernel for Application-specific Operating System Services. In *6th ACM SIGOPS European Workshop on "Matching Operating Systems To Application Needs"*, pages 68–71, 1994.

[BSP+95] Brian N. Bershad, Stefan Savage, Przemysław Pardyak, Emin Gün Sirer, Marc Fiuczynski, Susan Eggers, David Becker, and Craig Chambers. Extensiblity, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 267–284, 1995.

[Chi95] Shigeru Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Programming*, pages 285–299, 1995.

[CHJ+94] V. Cahill, Christine Hogan, Alan Judge, Darragh O'Grady, Brendan Tangney, and Paul Taylor. Extensible systems - the Tigger approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153. ACM SIGOPS, 1994.

[CIJ+91] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. Choices, Frameworks and Refinement. In Luis-Felipe Cabrera, Vincent Russo, and Marc Shapiro, editors, *Object-Orientation in Operating Systems*, pages 9–15. IEEE Computer Society Press, 1991.

[CIMR93] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and Implementing Choices: An Object-Oriented System in C++. *Communications of the ACM*, 36(9):117–125, 1993.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 483–501, 1993.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[DM89] Michael L. Dertouzos and Aloysius Ka-Lau Mok. Multiprocessor On-Line Scheduling of Hard Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 12 1989.

[DT88] Scott Danforth and Chris Thomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

[ES91] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing, 1991.

[Gal91] D. L. Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, 1991.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[KL93] Gregor Kiczales and John Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of the 3rd Workshop of Object-Oriented Operating Systems*, pages 25–30, 1993.

[KTW92] Gregor Kiczales, Marvin Theimer, and Brent Welch. A New Model of Abstraction for Operating System Design. In *Proceedings of the 2nd International Workshop on Object-Oriented Operating Systems*, pages 346–349, 1992.

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LYI95] Rodger Lea, Yasuhiko Yokote, and Jun-Ichiro Itoh. Adaptive Operating System Design Using Reflection. In *Proceedings of the 5th Workshop on Hot Topics on Operating Systems*, pages 95–100, 1995.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, 1987.

[Mey92] Betrand Meyer. *Eiffel—The Language.* Prentice Hall International, 1992.

[OIT93] H. Okamura, Y. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. In *Proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, pages 110–127. Springer Verlag, 1993.

[SB94] Stefan Savage and Brian N. Bershad. Issues in the Design of an Extensible Operating System. Technical report, Dept. of Comp. Science, University of Washington, 1994.

[Str92] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Second edition, 1992.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 227–241, 1987.

[W+93] Robert Wahbe et al. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating System Principles*, pages 203–216, 1993.

[Yok93] Yasuhiko Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *Proceedings of the 1st International Symposium on Object Technologies for Advanced Software*, pages 145–162. Springer Verlag, 1993.

[YT89] Akinori Yonezawa and Mario Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1989.

[ZC95] Chris Zimmermann and Vinny Cahill. Roo: A Framework for Real-Time Threads. In *Proceedings of the Workshop on Distributed and Parallel Real-Time Systems, held at the 9th International Parallel Processing Symposium*, pages 137–146, 1995.

[Zim96a] Chris Zimmermann. How to Structure Your Regional Meta—A New Approach to Organizing the Metalevel. In Chris Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.

[Zim96b] Chris Zimmermann. Metalevels, MOPs and what the Fuzz is all about. In Chris Zimmermann, editor, *Advances in Object-Oriented Metalevel Architectures and Reflection.* CRC Press, 1996.

[ZK93] Chris Zimmermann and Albrecht W. Kraas. *Mach: Concepts and Programming (in German).* Springer-Verlag, 1993.