A Reflective Architecture for an Adaptable Object-Oriented Operating System Based on C++

Frank Schubert

Chemnitz University of Technology
Department of Computer Science
Operating Systems Group
09107 Chemnitz
Germany

e-mail: fsc@informatik.tu-chemnitz.de

May 1997

1 Introduction

Todays operating systems have to support applications and hardware with highly specialized requirements. Traditional all-purpose operating systems can't be an optimal runtime environment for all the diverse applications. Therefore, instead of huge universal operating systems small tailored systems are needed to provide exactly the services and properties really required in a concrete situation.

In the field of software development the object-oriented paradigm has been widely accepted as powerful method to achieve adaptability. Hence, if the set of required properties remains the same during the whole runtime of a system, object-oriented frameworks like PEACE [Schröder–Preikschat93], Choices [Russo91] or Tigger [Cahill94] are well suitable to manufacture tailored operating systems. However, once booted, such a system can not be adapted to changed requirements ($\leadsto$ *static adaptability*). If rebooting is not acceptable (due to the required availability or the effort for rebooting) a way for dealing with future requirements by dynamic modification of the running system is needed ($\leadsto$ *dynamic adaptability*). Several commercial systems (Solaris, AIX, etc.) and a lot of research systems (Spin [Bershad95], Bridge [Lucco94], Apertos [Yokote93]) are already dynamically modifiable. Also some of the named frameworks have been extended to support dynamic adaptation (e. g. PEACE [Schmidt95] and Choices [Madany92]). However, possible modifications are often highly restricted and complicated to perform. Even though a system has been structured in an object–oriented manner and implemented in an object-oriented language, during runtime usually nothing of that structuring information is still available. After compiling and linking the system, there is no knowledge about classes, class membership, etc. Therefore, some operating systems are constructed as object management systems (e. g. BirliX [Härtig90] or Clouds [Dasgupta91]). They are able to manage objects in various ways and to use objects as the basic components for service providing, adaptation, migration, etc. However, the resulting object structure differs considerably from the structures used during software development (source-code level). Because of the often very heavy-weight objects (private address space and own thread of control) no fine–grained adaptation is possible.

In the CHEOPS[1] project we are applying an approach for a reflective, object-oriented system architecture, to support fine-grained, dynamic adaptability. It is based on the idea to close the

---

[1] CHEOPS – **CH**emnitz **OP**erating **S**ystem

gap between models and abstractions used during development (design and implementation) and the identifiable entities in the running system (see also [CHEOPS96]). By retaining most of the structuring information about classes, objects, and the relations between them it should be possible to perform the same extensions and modifications as have been done to the system's description (source code) within the running system itself.

The second section of this paper presents our point of view to the underlying concepts: reflection and runtime representations of abstractions. After that, the class-object architecture of CHEOPS is introduced. Section 4 gives a short impression about the implementation of this architecture based on C++. Finally other related and future work is discussed.

2 Reflection and runtime representations of abstractions

A clear and well comprehensible system architecture forms the general basis for each modification of a system. To perform the same steps of adaptation in the running system as have been done at source-code level we need an open architecture that fulfills the following requirements:

- The identifiable objects in the running system have to be the same (in granularity and functionality) as at description level, modeled by means of an object-oriented programming language.
- *Meta-level informations* as
  - available classes,
  - class hierarchy (*is-a* relations),
  - *using*-relations,
  - correlation of objects to classes,
  - and affinity relations between objects

  have to be still available in the running system.
- *Meta functionality* as creation, destruction, storage and life-time management of objects as well as object invocation mechanisms (all usually performed by the run-time environment) has to be opened up and assigned to identifiable entities in the system.

The solution is based on the concepts of *reflection* and *run-time representations of abstractions*. Reflection has been introduced by Brian Smith [Smith82]. Later the ideas have been broadened to the object-oriented world by Pattie Maes [Maes87]. The concept can be shortly outlined as the ability of objects (so called base-level objects) to know about their run-time environment (also called *infrastructure* or *meta level*) and to be able to make that environment to the matter of computation itself. In this way objects are able to change their (meta-)properties by modifying the meta level. In an object-oriented system the meta level itself may be also composed by objects.

Although the most work in the field of reflection has been done related to several programming languages (e. g. the meta-object protocol of CLOS [Kizcales93]) the Apertos-OS has shown that the concept is also suitable for an operating systems architecture [Lea95]. In our opinion the precondition for applying reflection within a running system is the availability of identifiable system components as run-time representations of the meta-level abstractions used during development. According to that we propose to transform classes into the running system and to assign them all the tasks resultant from the requirements above.

3 The Class-Object Architecture of CHEOPS

Dynamic adaptation in CHEOPS is done by adding or exchanging classes and objects during the system's runtime. Therefore, as explained above we need a representation of classes within the running system, the so called *class objects*. To distinguish the base-level objects from the class objects the former are called *regular objects* (see Fig. 2). A class object is an identifiable object within the system. One class object exists for each description-level class in the system. The class object manages the objects belonging to its class. and is responsible for:

- creation and destruction of objects,
- object management (registering, localization),
- service negotiation,
- access control (e. g. by access control lists or capabilities),
- supporting object exchange by using the knowledge about the class hierarchy (abstract classes, polymorphism).

According to our basic architecture, class objects are part of the infrastructure of regular objects and therefore influence their meta properties. To modify these meta properties we would have to modify the class objects. For example, if object A invokes a service of object B, the real invocation has to be performed by the infrastructure, precisely by the class object of object B. Dependent on the class object, object invocation will be performed by a simple call, by a remote procedure call (RPC), or by sending a message, etc. To A and B this can be absolutely transparent. The functionality for performing object invocation could be modified, for instance to add access control or parameter conversion, without any notification to the communicating objects.

Class objects are specific for each class. As each object is described by its class, for each class object a class description, the *class-object class (COC)*, exists as well ($\rightsquigarrow$*meta-meta level*). Based on the class definition of regular objects, this class-object class is generated automatically (at source code level) by the *COC-generator*.

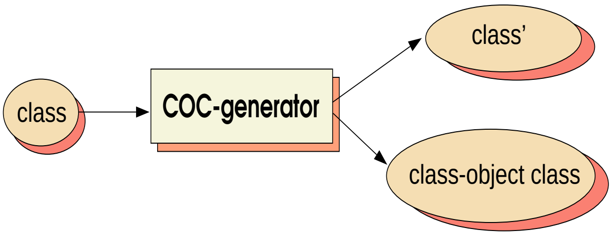

Figure 1. The COC-generator

The COC-generator creates a class-object class, which is able to deal with the basic tasks of the class object. To extend or to change that functionality, the developer could specialize the class-object class by derivation.

Class-objects can be added or removed to/from the system dynamically. Loading classes and creating class objects is based on dynamic linking and supported by the so called *class-object manager (COM)*.

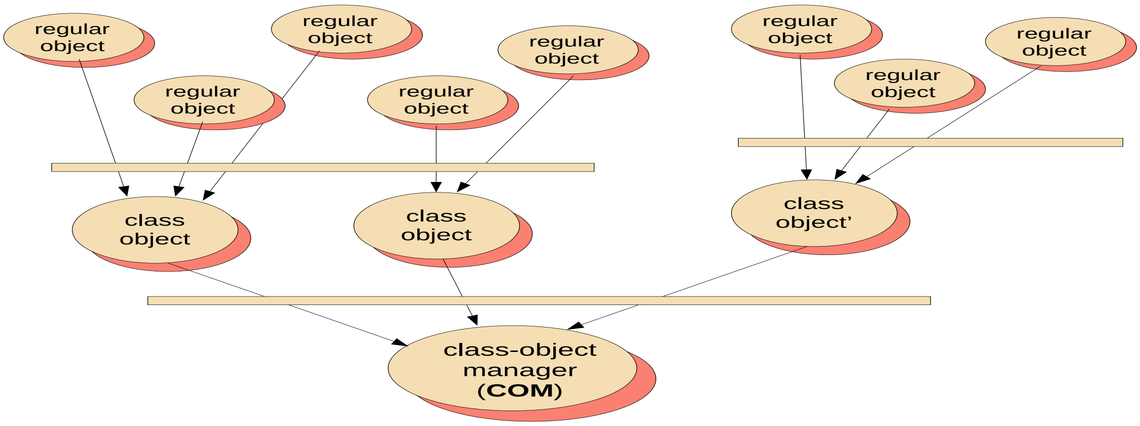

Figure 2. The Class-Object Architecture

The COM exists right from the start within the system and is responsible for:

- loading new classes (loading the class and the class-object class, creating the class object),
- removing classes,
- exchanging classes,
- managing the class hierarchy.

While adding a new class is a simple kind of extension (always possible), removing a class is only allowed, if currently no objects of that class exist. Exchanging classes can be used, for example, for correction of programming errors in an existing class. In this case the class object of the old class has to provide a service to determine and store the current state of all its objects. The new class object has to use these data to reconstruct all objects. This kind of services is not part of the COC created by the current version of the COC-generator and have to be added by specialization of the COC.

Similar approaches have been already used in other systems, e. g. in SmallTalk [Goldberg83], where it is possible to create hierarchies of classes and meta classes. However, these systems are working by source code interpretation and, consequently, are mostly too inefficient to be used within an operating system's kernel. Therefore our approach is based on using a "compiler based", object-oriented programming language.

4 Realization Based on C++

4.1 Language and Restrictions

The realization of the shown approach is based on the C++ programming language. The decision to use this language was influenced, among others, by the following advantages:

- C++ compilers and appropriate development tools (e. g. class browsers) are available for a lot of hardware platforms,
- a lot of implementations in the field of system software and operating systems (e. g. Choices) have been done in C++ and show its suitability for constructing efficient systems.

Dynamic adaption in CHEOPS is based on adding, removing or exchanging classes and objects. If a new created object of a derived class has to substitute an old one, the new object generally can't be stored at the same place, because of different object sizes. Furthermore, objects have to be able to migrate into other infrastructures to change their meta properties. Therefore, *location transparency* for all those objects is needed. To avoid direct access to objects and influenced by the idea to use the C++-calling mechanism for virtual methods to implement our alternative object invocation mechanism we have decided to make some restrictions to the used language:

- no public data members are allowed,
- all methods have to be virtual,
- all objects are created dynamically,

Furthermore, because of the resulting equivocations[2] and their tricky implementation no multiple inheritance is allowed.

4.2 Implementation

Basics

The platform of our implementation is formed by the CHEOPS kernel. It is running standalone on Intel-based PC's (protected mode) and provides the basic functionality for memory management, thread management and message passing. On that base the class-object manager runs as kernel thread. To be able to load and reload classes during runtime it contains a small set of functions to support the dynamic linking process based on ELF's (Executable and Linking Format) position-independent code.

The COC-generator was implemented by using yacc and lex. The current prototype parses not the complete C++ syntax and expects syntactical correct code. As all the other necessary development tools it runs on top of Linux. By using an implemented communication mechanism (based on UDP) the COM is able to communicate with an special module-loader process to transfer compiled object modules into the CHEOPS kernel (see Fig 3). In this way development and first testing can be done on top of Linux[3]. After that, the object module is transfered to the CHEOPS-kernel and the modifications are performed dynamically. One of the resultant advantages is the very short turn-around time during the incremental kernel development, because frequent rebooting is not necessary.

---

[2] caused by same member names within different base classes

[3] The first prototypes of COC-generator and also the class-object manager were running on top of Solaris and later Linux. That testing environment is still used for testing new code.

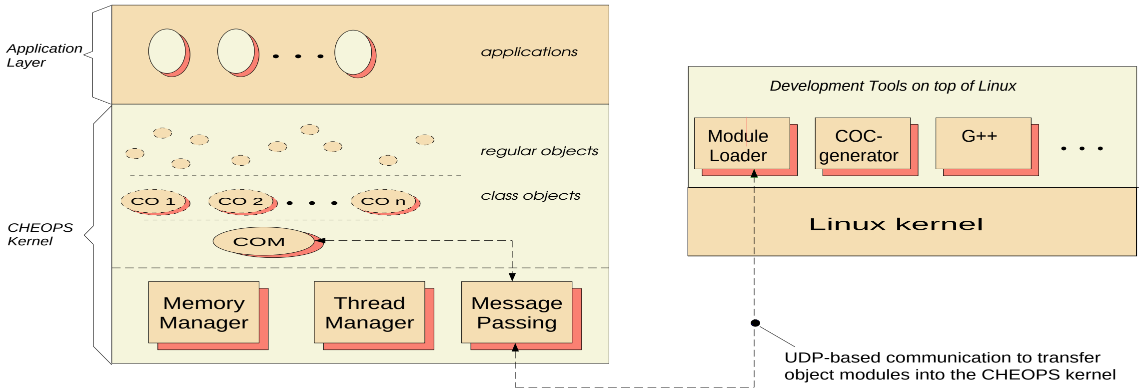

Figure 3. Loading Object Modules into the CHEOPS kernel

However, the system developer has to do several steps to add a new class to the system. As usual the developer has to create a class definition and implementation at source code level. After that he can build the class-object class by using the COC-generator. This class can be specialized by derivation as necessary. Finally the class-object manager loads the code of the class and the generated class-object class into the system and instantiates the class object.

All further tasks for managing objects of the loaded class have to be done by the new class object. For the implementation of class objects we have modified the mechanisms for object identification and method invocation.

Alternative Object Access resp. Method Invocation

All dynamically created C++ objects are referenced through the address of their data area [Stroustrup90]. Calls to virtual methods are performed indirectly via a virtual method table (VMT) referenced by a special component in the object's data area.

To obtain location transparency for objects we have modified this mechanism:

- Because different kinds of objects have to be handled differently (e. g. local or remote objects), potentially each object has to get its own VMT to meet the object's special requirements. Grouping of objects within the class to use the same VMT is possible.
- Class-objects manage all necessary information about the objects of its class. The creation mechanism for objects was modified in such a way, that it delivers not a pointer to an object but a pointer to an object description entry managed by the class object. As the first component in each object description entry the pointer to the (modified) VMT is stored so that the compiler generated method invocation is still working.

Figure 4 demonstrates the resulting procedure of method invocation. In the current implementation the COC-generator creates a class-object method corresponding to each method of the appropriate class. The new VMT refers to the class-object method instead to the object's method. Within that class-object method decisions can be made, how to proceed with the object's method invocation. Using the object description entry, the class object can detect if the object exists locally or remotely, if the caller and the callee are related to the same or to different threads, etc.

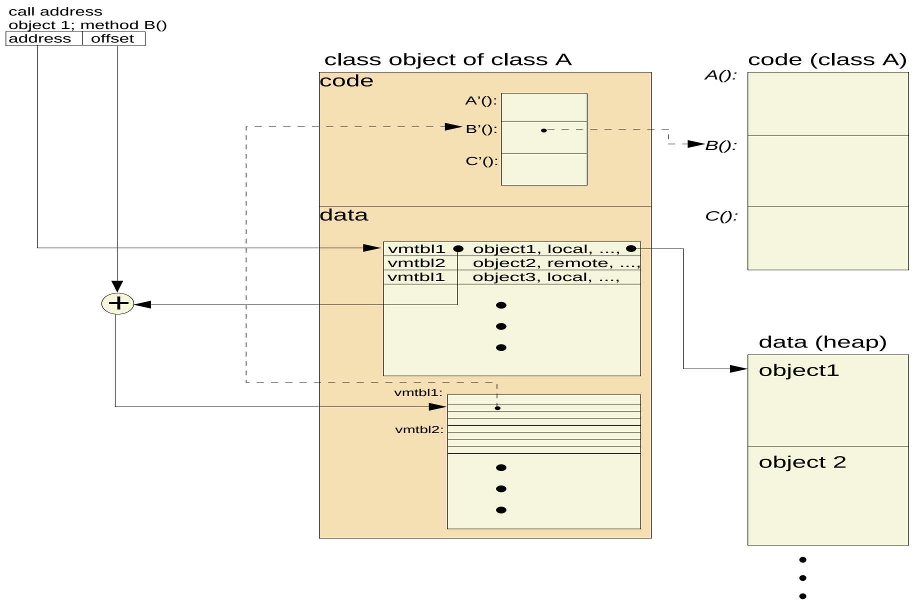

Figure 4. The Modified Structure for Object Invocation

A typical sequence of actions could be as follows:

1. register object invocation, for logging objects state,
2. check access permissions
3. check object's location and object-activity relations,
4. invoke method, either by a simple call or by an RPC,
5. register end of method invocation,
6. deliver output parameters.

To change, extend or reduce these steps, the system developer has to specialize the automatically generated class-object class and to tell the class-object manager to use this new class instead. For instance, in the current implementation only synchronous method invocation is supported, performed by a local call or an RPC. If it is necessary to modify this meta property of objects, the class object has to be modified (changing infrastructure).

A similar approach to achieve dynamic adaptability by modifying the management and the invocation structure of C++ objects has been presented by the Object Binary Interface approach [Goldstein94]. However, in contrast to that work, our approach is based on using the available C++ compilers without the necessity of compiler modifications.

5 Related and Future Work

Whereas the projects PEACE and Tigger focus on building a framework for the development of statically tailored operating systems, Apertos, BirliX, or DAS [Goullon78], as we do, focus on dynamic adaptation. However, in contrast to those systems, we propose fine-grained adaptation, based on retaining the fine-grained structures which exist during software development within the running system. Furthermore, we are able to modify the infrastructure of regular objects, by modifying/exchanging the appropriate class objects.

Research projects in the field of operating systems employing the notion of meta objects are, for instance, Apertos, Tigger, AEON [Gowing95], and FlexMach(OMOS) [Orr92]. They are related to our project in the respect, that our class objects and the class-object manager are special meta objects. Further projects investigating new approaches to flexible operating systems are, for example, Bridge [Lucco94], and Spin [Bershad95]. These projects mainly explore secure ways to bring new code into the operating system kernel in order to adapt its behavior. Choices provides an loading mechanism to add new services to the running kernel based on run-time representations of classes. In contrast to that work the CHEOPS class-object classes are specific to each class. In this way each class object can perform exactly the services needed for the appropriate class.

During the next time some experiments with the implemented class objects have to be performed, in order to gain more experiences. The most suitable default COC functionality has still to be determined. This is necessary to prevent, that always specialized COC's have to be written. Till now, the whole prototype is located in a single address space and is running entirely in kernel mode. The support for application layer processes is still under construction. One further field of our investigations is the support of adaptation management by means of the class-object architecture [Wohlrab97]. Beyond this, we plan to perform a series of efficiency tests, in order to determine where the overhead introduced by the class objects is too big to be compensated by their advantages. The result of these tests will consist of guidelines, which components may be implemented based on the class-object architecture and which parts have to be realized in a traditional manner.

References

[Bershad95] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers: *Extensibility, Safety, and Performance in the SPIN Operating System.* University of Washington, Department of Computer Science and Engineering, Draft of March 30, 1995.

[Cahill94] V. Cahill, C. Hogan, A. Judge, D. O'Grady, B. Tangney, and P. Taylor: *Extensible Systems – The Tigger Approach.* Proc. of the SIGOPS European Workshop 1994, or:
University of Dublin, Trinity College, Department of Computer Science, Distributed Systems Group: Technical Report TCD–CS–94.

[Gowing95] B. Gowing and V. J. Cahil: *Making Meta Object Protocols Practical for Operating Systems.* IWOOOS '95, Workshop Proceedings, 1995.

[CHEOPS96] S. Graupner, W. Kalfa, F. Schubert, R. Vogel, J. Werner, and L. Wohlrab: *Dynamische Adaption in Betriebssystemen – Das Cheops Projekt.* In: Winfried Kalfa (Ed.): *CHEOPS: Betriebssystemforschung in Chemnitz.* Chemnitzer Informatik–Berichte, Technische Universität Chemnitz–Zwickau, Fakultät für Informatik, 1996.

[Dasgupta91] P. Dasgupta, R. J. LeBlanc, M. Ahamad, U. Ramachandran: *The Clouds Distributed Operating System.* IEEE Computer, pp.34-44, Nov.1991.

[Goldberg83] A. Goldberg and D. Robson: *SmallTalk–80: The Language and its Implementation.* Addison–Wesley, 1983.

[Goldstein94] T. C. Goldstein and A. D. Sloane: *The Object Binary Interface – C++ Objects for Evolvable Shared Class Libraries.* Technical Report SMLI TR–94–26, Sun Microsystems Laboratories, Inc., 1994.

[Goullon78] H. Goullon, R. Isle, and K.–L. Löhr: *Dynamic Restructuring in an Experimental Operating System.* IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, 1978.

[Härtig90] H. Härtig, W. E. Kühnhauser, W. Lux, W. Reck: *Architecture of the BirliX Operating System.* GMD, St.Augustin, 6p., 1990.

[Kiczales93] G. Kiczales et. al.: *The Art of the Meta-Object Protocol.* Cambridge: MIT Press, 1993.

[Lea95] R. Lea, Y. Yokote, and J. Itoh: *Adaptive operating system design using reflection.* Draft, to be presented at HTOS 1995.

[Lucco94] S. Lucco, R. Wahbe, and S. L. Graham: *Adaptable Binary Programs.* Carnegie Mellon University, Technical Report CMU–CS–94–137, also Proc. of Winter USENIX 1994.

[Maes87] P. Maes: *Concepts and Experiments in Computational Reflection.* OOPSLA 87 conference proceedings, pp. 233-240, 1987.

[Madany92] P. Madany, N. Islam, P. Kougiouris, and R. H. Champbell: *Practical Examples of Reification and Reflection in C++.* International Workshop on Reflection and Meta-Level Architectures, pp. 76-82, 1992.

[Orr92] D. Orr and R. Mecklenburg: *OMOS - An Object Server for Program Execution.* IWOOOS '92, Workshop Proceedings, 1992.

[Russo91] V. F. Russo: *An Object-Oriented Operating System.* PhD Thesis, University of Illinois, 154p., 1991.

[Schmidt95] H. Schmidt: *Dynamisch veränderbare Betriebssystemstrukturen.* PhD Thesis, University Potsdam, 1995.

[Schröder–Preikschat93] W. Schröder–Preikschat: *Design Principles of Parallel Operating Systems – A PEACE Case Study.* Technical Report 93–020, ICSI Berkeley, 1993.

[Smith82] *Reflection and Semantics in a Procedural Language.* PhD Thesis, Massachusetts Institute of Technology, 1982.

[Stroustrup90] B. Stroustrup, M. Ellis: *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Wohlrab97] L. Wohlrab: *Configuration and Adaptation Management for Object-Oriented Operating Systems.* ECOOP 97 workshop "Object-Orientation and Operating Systems", Jyväskylä, Finland, 1997.

[Yokote93] Y. Yokote: *Kernel Structuring for Object–Oriented Operating Systems: The Apertos Approach.* Sony CSL, Technical Report SCSL–TR–93–014, 1993.