Application-Oriented Operating Systems

Antônio Augusto Medeiros Fröhlich

June 2001

To my son Alek.

Acknowledgements

This dissertation is the summit of a research project that was supported by several people and institutions, to which I am very grateful. At first, I am very grateful to my wife Luciane for being my “best friend” and giving me emotional peace to enjoy this wonderful journey.

I would like to thank my supervisors Prof. Jähnichen and Prof. Schröder-Preikschat for the invaluable scientific assistance they provided. Prof. Jähnichen looked after a prolific academic environment and an incomparable research infrastructure, allowing this research to surpass its original goals. Prof. Schröder-Preikschat dedicated uncountable hours to discuss with me the most diverse topics of operating system construction. His exceptional perspective of the subject deeply influenced this dissertation, motivating me to move away from the mainstream and break barriers that formerly seemed insurmountable.

Many thanks to those who helped this doctoral project to come true. The Federal University of Santa Catarina for giving me leave to join GMD-FIRST, in particular the colleagues that took over my tuition duties and my representative Prof. Leite. The CAPES agency for the financial and organizational support. The GMD-FIRST institute, in particular Prof. Behr and former associate researcher Dr. Jörg Cordsen who “opened up the door”.

I also would like to thank the “Fluchtsofa” fellows—some of the most brilliant minds I ever had the pleasure to bother with my endless questions—for teaching me the importance of the “Reinheitsgebot”. In particular Friedrich Schön, for the comprehensive insight of software engineering, and my students Gilles P. Tientcheu and Sascha Römke for the dedicated work on EPOS.

Loving thanks to my parents Arno and Nadir for shortening the distance between Germany and Brazil with their daily e-mails and sporadic visits.

Berlin, June 2001.

Abstract

The majority of processors produced nowadays are targeted at *dedicated computing systems* that execute either a single application or a small set of previously known applications. In contrast to generic computing systems, these dedicated systems have very specific run-time support requirements, which are not properly fulfilled by general-purpose operating systems. The impossibility to anticipate which applications will be executed results in generic operating systems being forced to provide an extensive set of services targeted at making all resources available to all applications. The standardization of such generic system services locked general-purpose operating system inside a hard shell that prevents innovations from reaching applications. With regard to dedicated computing, these generic operating system provide uncountable services that are not used by individual applications, and yet fail to fulfill application demands.

This dissertation proposes a novel strategy to systematically construct application-oriented operating systems as arrangements of adaptable software components. Instead of standard compliance and hardware properties, the features offered by such a system emanate directly from application requirements, thus enabling it to be customized according to the needs of particular applications. Such application-tailored system instances are produced by selecting, configuring, and composing proper components. Even if applications refrain from the new application-oriented services in benefit of standard interfaces, most dedicated applications require such a small subset of those interfaces that mapping them to new system services—instead of porting their traditional implementations—is usually possible.

The *Application-Oriented System Design* multiparadigm design method proposed in this dissertation guides domain decomposition towards *families of scenario-independent system abstractions* that can be reused to build a variety of run-time support systems. Environmental dependencies observed during domain decomposition are separately modeled as *scenario aspects,* which can be transparently applied to system abstractions with the aid of *scenario adapters*. The assembling of such *software components* to produce a functioning system is assisted by *component frameworks*, which capture elements of reusable software architecture identified in the course of domain engineering. Usability is improved by *inflated interfaces,* which export whole families of abstractions to users as if they were single macrocomponents, passing the responsibility of selecting appropriate family members to the system.

■ Abstract

The concepts and techniques introduced by application-oriented system design were verified during the development of EPOS (*Embedded Parallel Operating System*), an application-oriented operating system for the domain of high-performance dedicated computing. The prototype of EPOS implemented for the SNOW cluster of workstations consists of a repository of software components that encapsulate system abstractions and scenario aspects, a statically metaprogrammed component framework, and a set of tools that is able to automatically select and configure components in order to generate application-oriented system instances.

Keywords: application-oriented operating system, parallel and embedded systems, software components, domain engineering, object-oriented design, family-based design, generative programming, aspect-oriented programming, static metaprogramming.

Zusammenfassung

Die Mehrzahl der heutzutage produzierten Prozessoren, wird in *Spezialsystemen* eingesetzt, die einer einzelnen oder von einer kleinen Zahl vorher bekannter Anwendungen gewidmet sind. Der Betrieb solcher Spezialsysteme unterliegt besonderen Anforderungen, die Allzweckbetriebssysteme, wie sie auf Arbeitsplatzrechnern zum Einsatz kommen, in der Regel nicht erfüllen können. Da es vorab nicht möglich ist anzugeben, welche Anwendungen ausgeführt werden sollen, bieten Allzweckbetriebssysteme potentiellen Anwendungen vorsichtshalber eine sehr große Menge von Diensten an. Die Standardisierung solcher allgemeinen Dienste schließt ein Allzweckbetriebssystem in starre Schranken ein und erschwert die Nutzbarmachung innovativer Konzepte für die Anwendung. Das Problem von Allzweckbetriebssystemen für Spezialrechensysteme ist, dass einerseits unzählige Dienste angeboten werden, die die Anwendungen gar nicht benötigen, andererseits aber spezielle Anforderungen nicht erfüllt werden.

In dieser Arbeit wird ein neuer Ansatz dargelegt, um aus anpassungsfähigen Softwarebausteinen systematisch anwendungsorientierte Betriebssysteme zu konstruieren. Anstatt die Eigenschaften eines Systems aus Standardfunktionen und Hardwareeigenschaften abzuleiten, bestimmen die Anforderungen der Anwendungen die konkrete Systemausprägung. Dies vereinfacht die Konstruktion angepasster Systeme. Solche anwendungsspezifischen Systeme werden durch Auswahl, Konfiguration und Integration geeigneter Komponenten gebildet. Dieser Ansatz ist sinnvoll selbst dann, wenn Anwendungen neue Spezialisierungen zunächst nicht nutzen und stattdessen ausschließlich auf Standardfunktionen zurückgreifen. Die Mehrzahl der Spezialsysteme benutzt nur eine Teilmenge der Standardfunktionen, so dass es vergleichsweise einfach ist, diese Teilmenge auf neue Systemdienste abzubilden, ohne dabei alle Standardfunktionen zu portieren.

Die in dieser Arbeit diskutierte *anwendungsorientierte Entwurfsmethodik* basiert auf der Zerlegung eines Anwendungsgebietes in eine Menge von *Familien szenario-unabhängiger Systemabstraktionen*, die je nach Bedarf zur Konstruktion eines angepassten Laufzeitsystems wiederverwendet werden. Umgebungsabhängigkeiten, die während der Zerlegung auftreten, werden getrennt als sogenannte *Szenario-Aspekte* modelliert und wirken eigenständig in Form von *Szenario-Adaptern* auf die Systemabstraktionen ein. Die Spezialisierung und Integration solcher *Softwarekomponenten* zu einem funktionsfähigen Gesamtsystem wird durch *Komponentengerüste* ergänzt, die die im Verlauf der Zerlegung eines Anwendungsgebietes identifizierten, wiederverwendbaren Elemente der Software-Architektur erfassen. Die Anwendbarkeit der Komponenten wird weiterhin durch die Bereitstellung einer *umfassenden Schnittstelle* verbessert. Jede dieser umfassenden Schnittstellen exportiert eine ganze Familie von Abstraktionen und wirkt dadurch als eine Art Makrokomponente. Daraus folgt, dass die Auswahl geeigneter Familienmitglieder dem System überlassen wird.

Die Konzepte und Techniken, die durch die *anwendungsorientierte Entwurfsmethodik* eingeführt wurden, wurden bei der Entwicklung von EPOS (*Embedded Parallel Operating System*), einem anwendungsorientierten Betriebssystem für das Gebiet des spezialisierten Hochleistungsrechnens, überprüft. Der Prototyp von EPOS wurde auf einem Cluster von Arbeitsplatzrechnern implementiert und besteht aus einer Sammlung von Softwarekomponenten, aus einem statisch meta-programmierten Komponentengerüst und schließlich einem Satz von Werkzeugen zur automatischen Auswahl, Konfiguration und Erzeugung anwendungsorientierter Systeminstanzen.

Schlagwörter: anwendungsorientierte Betriebssysteme, parallele und eingebettete Systeme, Softwarekomponenten, Domain Engineering, objektorientierter Entwurf, familienbasierter Entwurf, generative Programmierung, Aspekt-orientierte Programmierung, statische Metaprogrammierung.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter aims at establishing the context in which this dissertation has been written by briefly introducing its theme, *application-oriented operating systems*. Subsequently, the motivation and the goals defined for the scientific investigation that culminated in dissertation are presented. The main contributions of this work are then summarized, followed by an overview of the forthcoming chapters.

1.1 Prologue

A computational system, as the term suggests, exists to perform computations. Any resource used for something else is wasted. Nevertheless, translating a complex computation into boolean and arithmetic operations, regardless of sophisticated compiling techniques, is not always convenient. Abstracting physical resources into more easily usable logical entities has been accepted as an adequate alternative since the early days of electronic computing, thus yielding a layer of software between applications and hardware that we call the *operating system*.

Nowadays, when one thinks about an operating system, what usually comes to mind is an all-purpose operating system running on a workstation. These generic computing systems, however, count for just a small fraction of the total: according to Tennenhouse [Ten00], roughly 2% of the 8 billion microprocessors manufactured in the year 2000 found their way into a workstation, while dedicated systems, especially embedded ones, took the larger share.

Dedicated computing systems are designed aiming at specific applications that are known by the time the system is built. Therefore, delegating the resource management of such systems to generic operating systems, which are designed to support virtually any kind of application, would be mostly inadequate. As Anderson [And92] and Schröder-Preikschat [SP94b] emphasize, the adjectives *generic* and *optimal* cannot be assigned to the same operating system, since each application has particular demands concerning run-time support that cannot be fully supplied by a generic system. A specialized operating system can explore the proper means to precisely fulfill the requirements of particular applications, while a generic one is usually fastened to a compromise of making resources available to all applications.

Parallel computing is tied in with dedicated systems, not only because some parallel machines run a restricted set of applications that are known in advance, but mainly because parallel applications run for long periods in exclusive mode, i.e. they run alone in the set of nodes that has been assigned to them. Consequently, parallel execution environments can be seen as temporarily dedicated systems and can benefit from specialized run-time support systems just like permanently dedicated systems do. The time required to reconfigure the operating system in order to fulfill the requirements of a particular parallel application, even if a full reinitialization is needed, is easily compensated by the benefits such a system can bring.

Motivated by the large market of dedicated computing, several commodity operating system developers offer downgraded versions of their products. However, these patched systems can seldom be considered under software quality metrics other than performance, for their inflexibility prevents most structural enhancements. This is mainly a consequence of the no less inflexible standardized application program interfaces and protocols with which they have to comply. Even if such an operating system was able to achieve significant improvements internally, standardized interfaces would probably prevent them from reaching applications [Pik00].

Nevertheless, compared to the interactive, graphic, web-aware applications traditionally executed on workstations [BDF+97], dedicated applications adhere to a restricted set of standards. Parallel applications, for instance, usually implement graphical user interface and parallel computation as separate programs that run on different platforms. In this scenario, the user interface could be delegated to an ordinary workstation running a full-fledged operating system, while the parallel computation would take place on the nodes of a parallel machine supported by a dedicated operating system. Such an operating system would only need to comply with standard interfaces effectively used by the application. Furthermore, complying with standardized interfaces does not necessarily mean porting their traditional implementations—mapping them to optimized implementations is usually possible.

Unfortunately, most operating systems, including those designed to support large-scale customization, miss the case for dedicated computing when they associate configurable features with hardware aspects and standard compliance, ignoring further application requirements. Being able to configure the operating system to benefit from special hardware features is certainly an important design decision, but hardware-driven optimizations are useless for applications that do not need the corresponding hardware features.

A customizable operating system that emphasizes application requirements while defining its configuration mechanisms would be an ideal solution to support dedicated computing. Such an *application-oriented operating system* would only include the functionality effectively needed to support a given application and would deliver it in a way that is convenient for that application. The advantages of this kind of system would not be restricted to performance and usability: by properly scaling down the system, one could reduce its inherent complexity, improving software quality in general and correctness in particular [CMSW94].

However, the degree of scalability necessary to achieve an application-oriented operating system ruptures with the traditional view of system software architecture, forcing the operating system to present itself in a variety of architectures. The choice of a monolithic, μ-kernel-based, or embedded-into-the-application architecture becomes conditioned to application requirements. The barrier that usually separates application and system becomes volatile, with system functionality floating from one domain to another, or with both domains fusing into a single one [BFM$^+$00]. Consequently, this kind of system calls for sophisticated software engineering techniques.

The guidelines for the development of application-oriented operating systems began to be established yet in the seventies, along with modular programming. Dijkstra [Dij68] introduced the ideas of *separation of concerns* and *levels of abstraction* to bypass a monolithic design, while the concept of *program families* introduced by Parnas [Par76] called for commonality and variability analysis. Identifying and modeling commonality across software pieces enabled system designers to capture common elements in reusable modules, while variability counted for system specialization (program family members). Another key contribution was the *incremental system design* proposed by Habermann, Flon, and Cooprider [HFC76]. A system designed following that proposal relies on a "minimal subset of system functions" to define a platform of fundamental abstractions that can be used to implement "minimal system extensions". These extensions take place successively, with each new level being a new minimal basis for higher-level system extensions, and the application being the topmost extension.

Nevertheless, the modular programming from the seventies failed to deliver the reusability level needed to achieve comprehensive program families—the full reuse of modules implemented in the realm of old family members for the construction of new ones was impracticable with the software engineering tools then available. Therefore, system designers had often to choose between honoring family-based design and implementing generic modules. At that time, a new software development paradigm was emerging and would bring about answers to many of the questions raised by program families: *object-orientation*. Wegner's work on classification in object-oriented systems [Weg86] played an important role on the forthcoming methodologies, programming languages and tools, which could finally deliver the reusability demanded by program families. The PEACE system [SP94a] developed at GMD-FIRST for the SUPRENUM [BGM86] parallel computer is a significant example of this period.

The maturing of program families and object-orientation ultimately led to a new software development strategy that promotes the reuse of software parts by proper encapsulation, classification, and composition. Such *software components* have the potential to enable software construction in a fashion similar to the traditional assembly lines of other industries, and are shaping new horizons for software development.

Nevertheless, though component-based software engineering provides means to achieve a truly application-oriented operating system, no such system has been introduced to the scientific community so far. Perhaps, the biggest challenges to build such a system originate from the necessity of bringing application and operating system to interact during system configuration and generation. Appropriate mechanisms have to be made available to applications so they can express requirements regarding the run-time support system. Means must also be provided to interpret application requirements in order to select, adjust, and combine software components to produce an application-oriented operating system instance.

1.2 Motivation and Goals

The recognition that dedicated computing, despite an impressive expansion in recent years, is mostly deprived of proper run-time support means was an important motivational factor for the scientific investigation that culminated in this dissertation. Even though the demand for customizable operating systems—that could efficiently support the execution of dedicated applications—is unambiguous, and though some of the means needed to fulfill this demand has been available for a long time (e.g. family-based design), the field remains relatively unexplored. Indeed, the vast majority of dedicated systems continues to be developed relying on run-time support systems that have to be haphazardly patched for each new application, often failing to match up application expectations [And92, Mah94, DBM98].

Another determinant factor for this dissertation was the understanding that, in what concerns operating system organization, parallel computing is a particular case of dedicated computing with special emphasis on performance. The opportunity to supply the demands of dedicated applications in the defying panorama of parallel computing was extremely exciting and strongly encouraged this work. Additionally, it was evident that many of the software engineering techniques needed to accomplish the envisioned scenario of operating systems that can be tailored to applications were still to be conceived. Hence the research would often leave the realm of operating systems to venture into software engineering. Having a chance to investigate the frontiers and intersections of these fundamental areas of computer science was also stirring, especially when recalling the seminal work conducted on the field by prominent computer scientists like Edsger Wybe Dijkstra, Charles Anthony Richard Hoare, Per Brinch-Hansen, and David Lorge Parnas.

Motivated by these factors, a full-time doctoral project was initiated, after one year of preliminary studies, at the Research Institute for Computer Architecture and Software Engineering (FIRST) of the German National Research Center for Information Technology (GMD) in September 1997. The project also received support from the Federal University of Santa Catarina (UFSC) and the Fundação Coordenação de Aperfeiçoamento de Peossoal de Nível Superior (CAPES) of the Brazilian Education Ministry.

The goals of this doctoral project can be summarized as follows:

> To study the conditions that surround the development of application-oriented operating systems in the realm of dedicated computing, aiming at defining a strategy to enable the systematic development of such systems.

Honoring the tradition in the operating system field, the approach chosen to pursue these goals was experimentation. A design method to support the engineering of system-related domains as collections of software components that can be arranged according to the needs of particular applications in order to yield application-oriented operating systems was elaborated simultaneously with an experimental operating system. In this way, design concepts and techniques could be verified while being refined to reflect the necessities of a real operating system project.

1.3 Contributions

The doctoral project delineate in the previous section was executed to its totality and produced a series of results that will be presented throughout this dissertation. Concisely, this dissertation proposes a novel operating system design method that enables the development of run-time support systems that can be tailored to fulfill the requirements of particular applications. Entitled *Application-Oriented System Design*, this multiparadigm design method guides domain decomposition towards *families of scenario-independent system abstractions* that can be reused to build a variety of run-time support systems. Environmental dependencies observed during domain decomposition are separately modeled as *scenario aspects*, which can be transparently applied to system abstractions with the aid of *scenario adapters*. The assembling of such *software components* to produce a functioning system is assisted by *component frameworks*, which capture elements of reusable software architectures identified in the course of domain engineering.

The correspondence between domain and design promoted by application-oriented system design enables the construction of run-time support systems whose features transcend standard compliance and hardware facets: application requirements develop into system features. Besides enabling the engineering of truly application-oriented operating systems, this relationship between requirements and features makes it possible to tailor the operating system to applications automatically. A strategy that allows applications to specify system requirements simply by invoking well-known operations is another contribution of this dissertation. The strategy consists of performing a syntax analysis of the application source code to identify system invocations and draw a blueprint for the system that has to be generated. Subsequently, the minimal combination of system abstractions and scenario aspects that is able to support the application is compiled in the context of a component framework to yield a tailored run-time support system.

Though conceived with system-level software in mind, application-oriented system design is not restricted to this kind of software. Many of its concepts bear answers to fundamental questions concerning the development of component-based software, and therefore can be deployed in the construction of other kinds of software. The specification of *scenario aspects* as independent constructs that can be transparently applied to abstractions; the organization of abstractions and scenario aspects in *families*; the factorization of families to yield *common packages* and *configurable features*; the unification of family members without implying in one being subtype of another through *inflated interfaces*; the representation of software architectures through *component frameworks* that embed mechanisms to accomplish system-wide (cross-component) features; the use of *static metaprogramming* to support efficient component composition; are just some of the principles of application-oriented system design that can be applied to the construction of component-based software in general.

Besides the application-oriented system design method, this dissertation also encompasses a detailed description of EPOS, an experimental application-oriented operating system developed to verify the software engineering concepts and techniques proposed. EPOS (*Embedded Parallel Operating System*) is the outcome of an application-oriented decomposition of the high-performance dedicated computing domain. It covers a large spectrum of issues concerning the construction of customizable run-time support systems for embedded and parallel applications, from hardware initialization to automatic system generation.

A prototype of EPOS implemented for the SNOW cluster of workstations extends the contributions of this dissertation over the field of cluster computing. The prototype consists of a repository of software components that encapsulate system abstractions and scenario aspects, a statically metaprogrammed component framework, and a set of tools that is able to automatically select and configure components to generate application-oriented system instances. In contrast to the generic operating system typically deployed in the field, EPOS instances include only the components effectively used by applications, providing system services via an application-oriented interface.

1.4 Overview

In the *next chapter*, issues concerning the design and implementation of customizable operating systems will be addressed in an attempt to establish what is currently state-of-the-art in the field. Configuration mechanisms deployed by modern operating systems will be described, and their applicability for the construction of customizable operating systems will be considered. Subsequently, the software design methods that are able to guide the construction of such systems will be discussed. The third part of the chapter covers the implementation of customizable operating systems as arrangements of reusable software components. The chapter is illustrated with examples of significant systems.

Chapter 3 presents application-oriented system design, a novel design strategy to enable the construction of application-oriented operating system as arrangements of software components. Firstly, the application-oriented domain decomposition strategy is explained. It explores commonality and variability analysis to model families of highly reusable, adaptable, application-ready abstractions, isolating scenario aspects and capturing fragments of reusable system architectures. The refinement of design entities identified during domain analysis is subsequently approached, considering peculiarities in regard to the organization of abstractions in families, the conciliation of family members under a common interface, the modeling of scenario aspects that can be transparently applied, and representation of architectural aspects in component frameworks. Afterwards, considerations about the implementation of application-oriented system designs are stated, emphasizing implementations in the C++ programming language.

Chapter 4 describes EPOS, the experimental operating system developed in the scope of this dissertation to validate the concepts and techniques introduced in chapter 3. After an introduction of historical facts and fundamentals, the application-oriented system design of EPOS is presented, including families of system abstractions, scenario aspects, and system architectures that result from the decomposition of the high-performance dedicated computing domain. Subsequently a strategy to automatically configure the operating system according to the needs of particular applications is presented.

Chapter 5 describes a prototype implementation of EPOS for the SNOW cluster of workstations. This implementation was carried out with the aim of corroborating the application-oriented system design of EPOS. The chapter begins with a discussion about cluster computing, followed by a description of the SNOW cluster. Subsequently, the most relevant details of the prototype implementation are discussed, including configuration tools and system utilities.

Chapter 6 is the conclusion of this dissertation. It presents a reasoning about application-oriented system design and EPOS, identifying their highlights and limitations and comparing them with similar works. Finally, the perspectives for further development and deployment of the ideas proposed in this dissertation are considered.

Chapter 2

Customizable Operating Systems

This chapter addresses issues concerning the design and implementation of customizable operating systems, i.e., systems that can be configured to satisfy specific requirements dictated by the hardware, users, or applications. Firstly, mechanisms deployed by modern operating systems to achieve configurability will be considered, followed by software design methods that promote the construction of customizable operating systems. Afterwards, the implementation of customizable systems as arrangements of reusable components will be approached, covering recent advances in the filed.

Whenever possible, topics will be illustrated with examples of significant systems. However, most operating systems do not clear identify the methods that guided their construction. Indeed, the dominant subject in the operating system literature, despite the large amount of titles including the word “design”, is implementation. This renders the operating system *design* scene poor in examples.

2.1 Configurability

An operating system is said to be configurable when it provides means by which its features can be modified. Configurability is achieved either by modifying the parameters that control the behavior of the system, or by including, excluding, and replacing parts of the system. In this way, the system can be adjusted to meet the demands of a particular user, application, or architecture .

Indeed, virtually all operating systems are somehow configurable. Even Microsoft DOS allows for some level of configurability as it interprets a configuration file (`config.sys`) to obtain system parameters and decide which device drivers will be loaded. Microsoft WINDOWS explores configurability by highly parameterized initialization (`.ini` files) and *Dynamically Loadable Libraries* (DLL) that extend the functionality of the system as needed. UNIX-like systems, in turn, tackle configurability by means of device drivers that can be linked to the kernel and server processes (`daemons`).

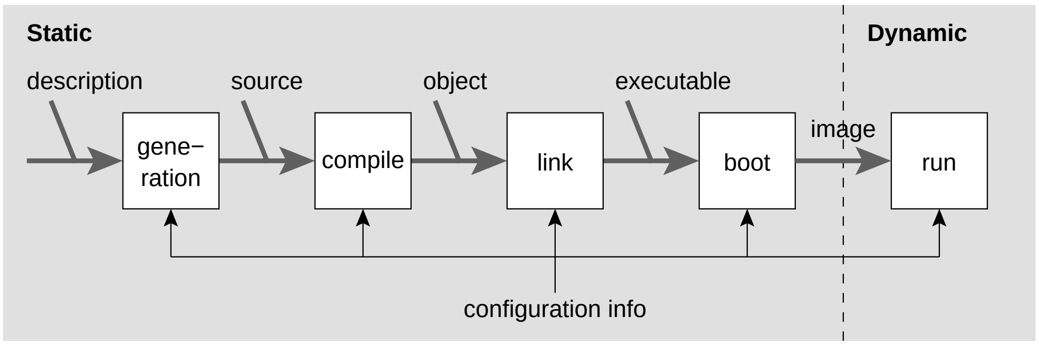

Figure 2.1: Stages in which an operating system can be configured.

Nevertheless, the fact of operating system being configurable does not automatically makes it customizable. Some systems try to improve on user-friendship and automatically configure themselves. Such systems usually lack (or hide) the control mechanisms that would allow users and applications to control the configuration process according to their needs. For example, detecting and activating available devices is a common practice that can hinder customizability, for the activated devices may be useless to currently running applications. In a customizable system, configuration must take place in such a way that users have the chance to select which features are present in the system at a given time.

A configurable system can be classified according to the time it is configured as static or dynamic (figure 2.1). In a statically configurable system, configuration takes place before the system begins to execute, while in a dynamically configurable system, it takes place during system execution. Static configuration has advantages on performance and resource utilization, since no reconfiguration mechanism has to be built into the running system. Dynamic configuration, in turn, has the benefit of extensibility: if the system faces an execution condition that demands features that have not been included in the initial configuration, it can reconfigure itself to include them.

All-purpose operating systems designed to equip workstations have long made the choice for dynamic configuration, since static configuration would be too restrictive in this scenario. With the actual technology, it would be unacceptable to request a workstation user to recompile the system, or even to reboot it, just because a new feature is required.

The universe of dedicated computing systems, however, has plenty of situations in which the requirements of applications that may come to run on the system are known in advance. In these cases, a statically configurable system would be of higher quality than a dynamically configurable one, since the absence of run-time reconfiguration mechanisms would result in a lighter system, and the elimination of complex reconfiguration operations would reduce the probability of crashes. Moreover, these benefits could be achieved without compromising flexibility, whereas all the features a dedicated application might demand from the operating system would be available from the instant it begins executing.

Nevertheless, the boundary between static and dynamic configuration is not always clear. A system could be statically configured to include some dynamic elements, giving the impression the configuration was dynamic. For example, a communication system could be statically configured to support a given network architecture and a set of communication protocols that can be dynamically switched at run-time. In this case, configuration continues to be static, since the set of protocols was defined before the system begun to execute and there is no way to include a new protocol afterwards. Another often observed case involves a statically configured μ-kernel that supports dynamic process creation. Although statically configured, such a system is open for dynamic extensions via server activation. That is, *the behavior of a statically configured system does not need to be static*.

2.1.1 Static Configurability

Static configuration takes place in an operating system before it begins to execute. Therefore, the criteria used to select which features will be included in the system have to consider the requirements of all applications that might come to run on it. If these criteria are not properly defined, applications may face "unavailable feature" conditions that will certainly compromise their execution. Most traditionally, users of a statically configurable system are requested to select features by hand and to probe-run it. However, features that are seldom used (e.g., triggered by exceptions) can easily be forgotten during the selection process, remaining unnoticed until they are effectively required at run-time.

Independently of the criteria and tools used to configure the system, static configuration relies on mechanisms that can be deployed in one of the following moments: link edition, compilation, or source code generation. Such mechanisms will be discussed next according to the time they are deployed. Static configuration could also be carried out at boot-time, but technically it is either restricted to the selection of a bootable image, or it falls in one of the other three cases.

Link-time: When implemented at link-time, static configuration is usually achieved by selecting precompiled object files from a repository (usually a library). The association of object files with system features supports configuration. The process of linking device drivers to a UNIX kernel follows this scheme [Bac87]. The major restrictions in this approach arises from the fact that object files are rigid structures, compiled in disregard of the conditions that will surround the execution of the resulting system. Moreover, several features cross the boundaries of object files, in the same way that object files may enclose more than a single feature.

The PEACE [SP94a] system developed at GMD-FIRST supports static configuration of its nucleus at link-time by automatically isolating class methods in separate object files. An ordinary link editor cares that only the methods that have been referred are included in the resulting executable. The HARMONY [Gen89] project at the National Research Council of Canada uses a static table to describe which object files are to be included in the resulting system. The FLUX [FBB+97] operating system toolkit at the University of Utah consists of a framework and a set of components (object files) organized in libraries. In order to configure an operating system, the user chooses between libraries and object files, which are then processed by a conventional link editor. A new version of FLUX [RFS+00] uses a custom language to describe binary components as well as to control the linking process, overcoming some of the restrictions imposed by ordinary libraries and link editors.

Compile-time: When static configuration is implemented at compile-time, it is mostly realized by conditional compilation, by "makefile" customization, or by special compilers. With conditional compilation, source code stretches are filtered out by a preprocessor according to externally controllable flags. The customization of "makefiles" can be used to select which source code units will be compiled, and how they will be compiled. Tools such as GNU AUTOCONF and X11 IMAKE are widely used for this purpose. Both mechanisms are often deployed in combination, with "makefile" customization controlling the preprocessor that supports conditional compilation.

The adoption of conditional compilation as a configuration mechanism is controversial. Some authors believe it to be a source of complications, especially regarding maintenance and correctness, because configuration elements get spread all over the code [PPD+95]. Notwithstanding, some other researches believe that, when properly used along with other techniques, conditional compilation represents an effective configuration mechanism, whereas it does not incur in run-time overhead [Cop98]. After all, systems that do not make use of conditional compilation at all are rare.

Static configuration at compile-time is explored in some systems by language extensions or even by completely new languages. Configuration information is included in the source code of the system by means of language specific constructs. It is interpreted later during system compilation to yield a particular system configuration. The MARS [KFG+93] project at the Technical University of Wien uses the MODULA/R language to support static configuration of the operating system in this fashion.

Generation-time: Static configuration can take place in a system during the generation of the corresponding source code. It can be accomplished by tools, by preprocessors, or by static metaprogramming. In the first case, tools are deployed to modify the system source code, or to generate it from a higher-level description, according to configuration information specified somewhere else. In the second case, the source code is annotated with configuration information, which is interpreted by a preprocessor to modify the associated code before it is fed into the compiler. In both cases, the configuration information is usually expressed in a *configuration language*. The third option utilizes the static metaprogramming [section 2.3.3.6] features of the language in which the system has been written. Configuration information is supplied as parameter to the metaprogram that, when executed, generates the source code for the corresponding system configuration. Clearly, this approach can only be used by systems written in languages that support static metaprogramming.

Static configuration at generation-time is being explored in several novel operating system projects, for it represents innumerable possibilities to achieve high configurability with low overhead. The PURE [SSPSS98] system under development at the University of Magdeburg focuses on deeply embedded applications. It defines a feature-based configuration scheme and uses aspect-oriented programming [section 2.2.5] techniques to manipulate the source code of system components. The GENESYS [Bau99] project at the University of Kaiserslautern uses parameterization in combination with generation techniques to fine-tune generic components in embedded systems. The EPOS system, which is the main experiment conducted in the realm of this dissertation and will be described in details in chapter 4, uses tools to identify the requirements of a given application and to select components that, when arranged in a framework, yield an application-oriented operating system.

Unrestricted to configuration, the approach of automatically generating source code is being studied in the realm of software development paradigms such as *subject-oriented programming* [HO93], *aspect-oriented programming* [KLM$^+$97], and *generative programming* [CE00]. These paradigms will be discussed later in section 2.2.

2.1.2 Dynamic Configurability

A system is considered dynamically configurable when its features can be changed while it is being executed. General-purpose systems designed to persist the execution of several distinct applications are the main motivation for dynamic configurability, since each application may challenge the system for particular features that cannot be determined in advance. As a dominant topic in operating system research, dynamic configuration has been extensively investigated with different approaches.

Dynamic process creation: An often-employed strategy to support dynamic configurability consists in implementing operating system duties outside the kernel, with dynamically created processes. Such server processes are only set to run when the functionality they implement is requested by an application. They can be implemented to share the operating system address space and run in supervisor-mode, or as ordinary user-mode processes. Servers interact with applications using means provided by the kernel, frequently communication channels or shared memory segments.

This approach has been used in so many systems that choosing examples becomes a challenge. Certainly one cannot forget the original UNIX [TR74] system developed at AT&T Bell Laboratories, which allowed some system services to be started on demand. The INTERNET services implementation in the BERKELEY SYSTEM DISTRIBUTION (BSD) of UNIX [LMK89] dynamically starts up servers with the aid of a kind of metaserver (`inetd`) that senses the network for the corresponding protocols. Close derivatives of this implementation are still in use in many contemporary systems.

The V-KERNEL [Che84] at the Stanford University innovated on system configurability by pushing the file system outside the kernel, thus allowing for dynamic reconfigurations. A similar strategy was used by the MACH system [ABB$^+$86] at the Carnegie-Mellon University to add on virtual memory and networking configurability. The AX system [Sch86] at the Technical University of Berlin supports process scheduling outside the kernel. The AMOEBA project [MT86] at the Vrije Universiteit Amsterdam uses the concept of *active objects* to implement configurable system services outside the kernel. The CHORUS system [RAA$^+$88], which was born at INRIA and is now commercially available from Sun Microsystems, allows dynamically created processes to run in supervisor-mode inside the address space of the kernel.

Kernel extensions: Dynamic configuration can also be implemented by supporting dynamic extensions of the operating system kernel. It can be accomplished via dynamic linking, on-the-fly compilation, or interpretation. In the first case, precompiled modules are linked to the kernel similarly to dynamic linked libraries—the attempt to access a module that has not yet been linked into the kernel invokes a built-in dynamic linker to fetch and link the respective module [DSS90, Dra93]. In the other two cases, the source code corresponding to the module, usually written in a simplified language, is fetched and then compiled or interpreted inside the kernel. The linker approach has performance advantages over the compiler or interpreter ones, since a compiler has high startup times, and an interpreter has to reinterpret the corresponding code every time it is invoked. Nevertheless, assuring safety in the linker approach is more complicated [SESS96].

The historical MULTICS system [Org72] introduced the “trap-on-use” mechanism to support dynamic system extensions. With this mechanism, only a small subset of system functions is initially loaded, while the memory regions where the remaining functions should have been loaded are configured to generate exceptions when accessed. The corresponding exception handler is able to load the missing functions and restart the application that generated the exception. This mechanism constitutes the basis for many other dynamic extension strategies.

LINUX uses a kernel thread (`kmod`) to automatically load missing modules, which adhere to the traditional UNIX pseudo-file scheme and have their interface with the kernel checked at load-time [Rub97]. Extension safety is delegated to the traditional file access control mechanism of UNIX. The VINO system [SS95] at the University of Harvard uses fault isolation techniques to preserve integrity after precompiled extensions, written in unspecified languages, are loaded into the kernel: all memory references in an extension are checked to fall within the boundaries of the allocated address space. The SPIN system [BSP$^+$95] at the University of Washington defines a core and a set of dynamically loadable extensions written in MODULA/3. The core has a built-in linker that react to events in order to load extensions. Core and extensions share the kernel address space in a protected domain scheme enforced by the MODULA/3 compiler. The SYNTHESIS system [PMI88] at the Columbia University uses an integrated compiler to generate specialized kernel services.

Reflection: A reflective operating system supports dynamic reconfigurations by exporting the *meta-information* associated to its objects through a *Meta-Object Protocol* (MOP) [KdRB91]. By interacting with meta-objects via the MOP interface, one can reconfigure the corresponding objects. Although very flexible, reflective systems pay a high price on performance, since they have to maintain a meta-level description of the whole system and to provide means to interact with it at run-time.

The OBERON programming environment [WG92] and the ETHOS system [Szy92], both developed at the Swiss Federal Institute of Technology, support reflection at the level of modules, which can be dynamically adjusted to match a given configuration. The APERTOS system [Yok92] at Sony Computer Science Laboratory defines a “metacore” that provides metaobject reflectors with the primitives needed to modify the configuration of the corresponding objects at run-time.

2.2 Designing for Customizability

As described in the previous section, a customizable operating system can rely on a variety of configuration mechanisms to support users and applications in selecting the features that will be available in a given system configuration. However, those mechanisms can only be deployed if the system as a whole is designed to endure customization. This section focuses on design strategies that promote customizability by enabling a system to be constructed as an assemblage of reusable parts. Such a system would be customized by selecting the proper parts and arranging them together.

Actually, the search for design methods to enable the development of software in a way similar to the assembly lines common to other industrial sectors has accompanied software engineering from the very beginning. It is true that the main motivation of this pursuit was the cost-effective development of new systems by reusing parts of preexisting ones, but the development of highly customizable systems can be attained based on the same principles: primarily partitioning the problem domain covered by the system in reusable and consistent units, and subsequently enabling the assembly of these units in a functioning system.

Nevertheless, there are several obstacles to achieve high levels of customizability in an operating system. Perhaps the most important one is the absence of design methods that explicitly consider the inherent peculiarities of system-level software. Although there are uncountable methodologies and tools that promote the reuse of implementation, design, analysis, know-how, and whatever takes part in the software development process, the vast majority of them have been proposed in terms of applicative software and are difficult to deploy at system-level. The simple fact that an operating system has nothing but the bare hardware to rely on is enough to break down with many application-level assumptions about memory management and synchronization. Altogether, system software has a particular compromise with correctness and performance, since both errors and delays propagate exponentially to the application-level.

Some design methods that promote customizability, and yet can meet the typical demands of system-level software, will be discussed next.

2.2.1 Family-Based Design

The roots of *Family-Based Design* can be tracked back to concepts such as Dijkstra's *separation of concerns* [Dij69] and Wirth's *stepwise refinements* [Wir71]; however, a family-based design method was first introduced by Parnas in his work on *program families* [Par76]. Family-based design is established around two complementary concepts: *commonality* and *variability* [CHW98]. Commonality is the basic grouping criterion, so entities that share common aspects considered relevant by the designer are grouped together to shape *families*. Conversely, variability brings about the differences that identify each of the *members* of a family. A family arises when the commonalities between the members are more important than the variations. In this context, Parnas defined a program family as follows:

> "We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them."
>
> (David Lorge Parnas [Par76])

Subsequently, Weiss extended this concept from "set of programs" to "collection of abstractions", giving origin to the *Family-oriented Abstraction, Specification, and Translation* (FAST) method [Wei95, WL99]. This new concept encompasses, but is not restricted to, class hierarchies in object-oriented design, with a *base class* characterizing the family (abstraction) and *subclasses* capturing the variations that distinguish family members. In the FAST method, commonalities are regarded as "design secrets" that are hidden as soon as they are acquired in detriment of variability, which effectively guides the design process. This design strategy is ideally supported by *Application-Oriented Languages* (AOL) that feature constructs to easily and quickly express the commonalities that are typical of the corresponding domain. For example, an operating system could be designed relying on a `process` construct that would gather the intrinsics of the process abstraction such as identity, creation, destruction, and execution. The design would then concentrate on variations like scheduling policy, multithreading, grouping, coordination, etc. The result would be a family of "processes".

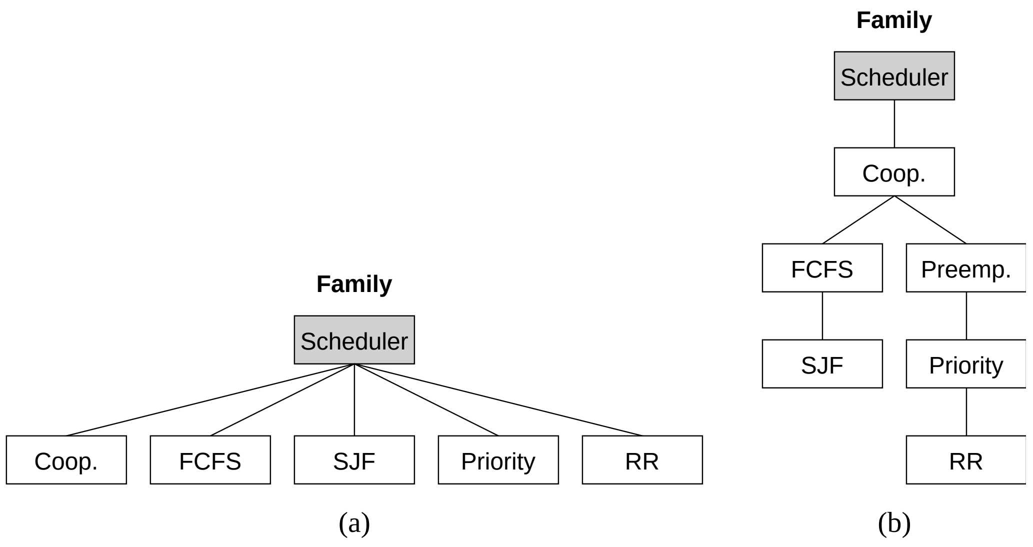

Figure 2.2: A family of scheduling algorithms modeled according to family-based design (a) and incremental system design (b).

Family-based design can be applied to the development of a customizable operating system with commonalities being accounted for the families of available system abstractions, and variability representing possible customizations. The system would thus be customized by selecting proper members of each abstraction family.

2.2.1.1 Incremental System Design

Incremental System Design was introduced by Habermann, Flon, and Cooprider [HFC76] to handle hierarchy in family-based design. Besides looking for commonalities and variations to shape families, they propose the problem domain to be organized in a hierarchical fashion. The most elementary functionality is gathered in a *minimal basis*, to which successive *minimal extensions* are applied. Family members, which in the original method were simply characterized by variations, are now organized in *levels of abstraction* [Dij68, PHW76], with each level being a substrate for the next, and the application being the final extension.

Figure 2.2 shows a family of scheduling algorithms modeled according to: (a) the original family-based design method, and (b) the incremental system design extension to that method. As it can be observed, incremental system design tends to generate deeper hierarchies, since variations are organized one upon another, with the most primordial closer to the root. Actually, the scheduler in this example is modeled as an abstraction and not as an algorithm; it is an agent that implements the operation "select next process to execute" in the realm of operating systems.

The *minimal basis* in figure 2.2(b) is a `Cooperative` scheduler, which indeed does not implement any policy, but give the means to schedule a process. In order to support different scheduling policies, members of the scheduler family enrich the abstraction accordingly. For example, the `Priority` member would probably tag processes with a "priority" to support priority-based scheduling, while the `Round-Robin` member would probably enrich the family with some sort of time keeping engine to implement the round-robin scheduling algorithm. Therefore, it is not the round-robin algorithm that is defined upon the priority-based, but the respective *abstractions*. Once more the similarities between family-based and object-oriented design become evident: incremental system design is for family-based design what subclassing is for object-oriented design.

Incremental system design is especially appealing for the operating system area because it gives the user a chance to select "how much he/she is willing to pay for a service". If performance is a major goal for an application, the programmer may decide to give up some advanced functionality for the sake of it. This demand can be easily accommodated in a system designed incrementally by selecting a family member closer to the "minimal base".

The PEACE parallel operating system [SP94a] developed at GMD-FIRST follows the guidelines of family-based and incremental system design. PEACE first version, developed for the SUPRENUM [BGM86] parallel computer, adopted a μ-kernel as the "minimal basis" for system extensions, which were accomplished by a collection of servers. A redesign for the MANNA [GBSP96] parallel computer produced a version of PEACE that no longer requires a μ-kernel: single-process-per-node configurations in which the operating system was completely embedded into the running application became possible. Both versions have been implemented as program families, with specialized family members for different classes of applications.

2.2.2 Object-Oriented Design

Object-orientation emerged simultaneously in several areas of computer science and was applied to software engineering in several approaches that culminated with the *object paradigm* and the respective disciplines of programming (OOP), design (OOD), and analysis (OOA). The object paradigm has been evolving for over 20 years and, notwithstanding constant improvements, is now well established. Because it is such an intensively studied subject, covered by an extensive bibliography[1], object-oriented design will only be summarized here, focusing the development of customizable operating systems.

Booch defines object-oriented design as follows:

> "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical as well as static and dynamic models of the system under design."

[1]Meyer [Mey88], Rumbaugh [RBLP91], Jacobson [JCJO93], and Booch [Boo94] cover object-orientation in depth and bring comprehensive bibliographies.

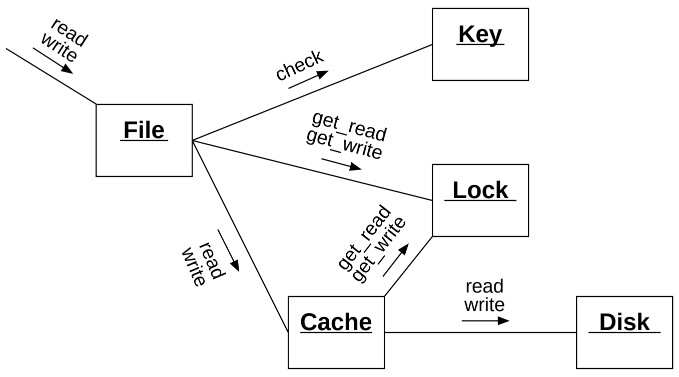

Figure 2.3: Object-oriented decomposition of a file system.

(Grady Booch [Boo94])

This definition emphasizes *object-oriented decomposition* as the most fundamental process in object-oriented design, by which the problem domain is decomposed in objects that abstract domain entities. These *objects* have well-defined behavior and can be viewed either as autonomous agents that do things, or as things upon which other objects act.

Figure 2.3 shows the object-oriented decomposition of a hypothetical file system. In this example, object `File` validates its operations with the aid of object `Key` and coordinate them with the aid of object `Lock`. Object `Cache` temporarily stores in memory part of `File`'s state, which is normally maintained by object `Disk`. Differently from what would happen in structured design, the algorithms that belong to the semantics of this file system are not represented in the early phases of design. They are implicitly designated by the operations of correspondent objects.

During decomposition, objects with similar responsibilities are grouped as to form *classes*. In the example above, all files in the file system present the same behavior and hence would be modeled by a single class. This grouping results from commonality analysis. Nevertheless, if a new kind of file is to be supported by this file system, for example files with data integrity verified via checksums, a new *subclass* of file would be defined. Subclassing is used to express variability in the design and naturally builds hierarchies [Weg86].

Commonality and variability are the main guidelines behind object-oriented decomposition, making the process of identifying objects in a given domain very similar to the processes of identifying families in family-based design. The resulting logical model of this process is a "collection of cooperating objects, in which individual objects are treated as instances of a class within a hierarchy of classes (Grady Booch [Boo94])". Classes are the constructs that specify the behavior and structure of objects in object-oriented languages and therefore are accounted for the static part of logical design, while their instances, the objects, correspond to the dynamic part.

In addition to the logical model, object-oriented design outputs a physical model for the system being modeled. This model is based on *modules* and *processes*. Modules constitute a second level of decomposition, in which classes are grouped according to the cohesion and coupling principles of structured design. A process model completes the design with information about the coordination of objects at run-time, including concurrency and synchronization. These four dimensions of object-oriented design are represented in figure 2.4.

Object-oriented design constitutes an important advance for the development of customizable operating systems. Being able to partition the operating system domain in reusable abstractions, designers can construct a variety of systems simply by arranging the proper components together. Furthermore, the task of arranging them together can be largely facilitated by capturing a system architecture in an object-oriented framework (see section 2.3.3.4). In practice however, object-oriented operating systems have difficulties to leave the design level and reach acceptable implementations. This happens mainly because of the following two factors:

1. Complexity due to high variability: although the operating system domain can be decomposed into a relatively small set of abstractions, most of them well-known conventions created by computer scientists, these abstractions usually present a high level of variability. For example, one could easily think of hundreds of variations for the communication abstraction, including aspects such as buffering, flow control, error detection and correction, encryption, and many others. High variability leads to complex class hierarchies that are difficult to manage without proper tools. Therefore, many designers opt for replacing a complex class hierarchy with a couple of generic abstractions that encompass the most traditional features. Such arbitrary simplifications often compromise customizability.

2. Programming languages: most object-oriented programming languages are not adequate to build operating systems. Languages that automatically manage the memory or that suppose all objects to be polymorphic are certainly not a choice, not only for performance reasons, but also for lack of determinism. Every operating system has to deal with asynchronous events generated by the hardware, and it is very unlikely that such events could wait, for example, for the completion of a garbage

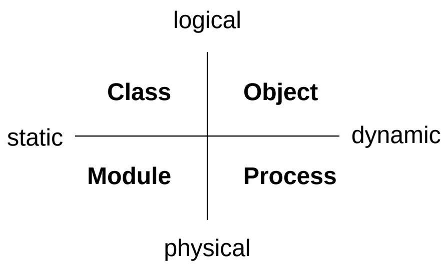

Figure 2.4: The models of object-oriented design.

collection operation. The lack of proper object-oriented programming languages for the operating system domain discourages the use object-oriented design.

The CHOICES object-oriented operating system [CJR87] from the University of Illinois at Urbana-Champaign has been designed as a hierarchy of object-oriented frameworks and implemented in C++. Each framework corresponds to a subsystem (e.g. virtual memory, file system, etc) that can be customized through the use of inheritance [CIM92]. System resources, policies, and mechanisms are represented as objects in the context of a framework. Several experiments have been conducted with the file system framework, enabling CHOICES to mimic several commercial file systems and corroborating its extensible design [MCRL89].

The ETHOS operating system [Szy92] developed at the Swiss Federal Institute of Technology covers extensible objected-oriented programming from the hardware up to the applications. The main goal of the project was exactly to experiment with object-orientation in the field of operating systems. The system has been modeled as a strongly typed hierarchy of abstractions, for which default implementations exist. High extensibility is achieved by restricting the use of inheritance in favor of forwarding. *Directory objects* act as proxies to access extensions, enabling modules to be dynamically loaded.

2.2.3 Collaboration-Based Design

Collaboration-Based Design (CBD) [BC89, RAB$^+$92, BO92, Hol93] *Role-Based Design*, is a design method that extends object-oriented design to express that an object may play different roles in a system, and that a cooperating suite of roles (collaboration) can be a better unit of reuse and composition than a class. A *collaboration* is thus defined by a set of objects and an interaction protocol that specifies their roles in the collaboration. Likewise, a *role* can be interpreted as the part of an object that enforces the interaction protocol. Collaborations can be expressed in *collaboration diagrams* similar to the one shown in figure 2.5, with roles in the intersection of classes and collaborations.

In the hypothetical collaboration diagram depicted in figure 2.5, class `queue` plays the role of a thread queue in the `thread` collaboration, which also leans on class `memory` for the `stack` and on class `timer` for the preemption mechanism. Class `memory` also plays the roles of a `buffer` in the `mailbox` collaboration and of a `cache` in the `file` collaboration. Class `timer` does not collaborate in `file`, while class `memory` does not play a role in `semaphore`. Besides playing the `preempter` in collaboration `thread`, class `timer` also plays the role of a `time-out` engine for collaborations `mailbox` and `semaphore`.

In a collaboration-based design, the system is expressed as a composition of independently definable collaborations. In this way, collaboration-based design has the potentiality to guide the development of longed-for reusable components. However, as a design discipline, it does not dictate any particular strategy to implement collaborations, neither to compose them. Implementation disciplines that promote composition in collaboration-based design will be discussed in section 2.3.3.5.

2.2.4 Subject-Oriented Programming

Subject-Oriented Programming (SOP) was introduced by Harrison and Ossher [HO93, OKH$^+$95] as an extension of the object-oriented paradigm to handle a "multiplicity of subjective views" of objects been modeled. Subjective perspectives of an object originate from the fact that some of its properties may be more interesting to some programs than to others. For example, the process abstraction in figure 2.6 could be viewed as a "schedulable subject" by the process scheduler, with properties such as execution time and priority. It could also be viewed as an "authenticable subject" by the security monitor, which would probably be more interested on keys and protocols. Subject-oriented programming avoids this kind of clash by allowing both perspectives to be independently, yet consistently, developed.

In subject-oriented programming, *subjects* are collections of classes, or class fragments, that model a subjective view of a domain. These subjective views are later reconciled during subject composition. The process of *subject composition* combines class hierarchies to produce new subjects that incorporate functionality from existing ones. In this way, subject-oriented programming enables decentralized development and supports system extensions without requiring modifications in the original source code.

Besides a design method, subject-oriented programming also features an implementation discipline that can guide C++ implementations of subject-oriented designs. It relies on an extended compiler to automatically derive abstract descriptions of subjects, called *subject labels*, directly from their source code. Subject labels are then composed, according to composition rules written in a special declarative language [OKK$^+$96], to produce *result labels* that describe the composed subject. Result labels can be used to guide the generation of the final system, or as input to further composition.

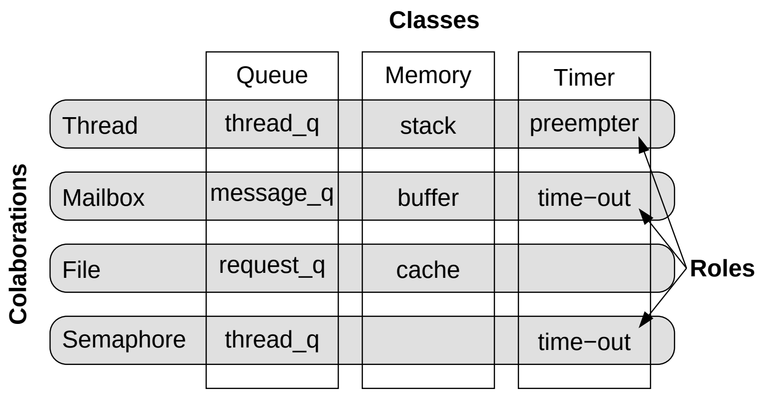

Figure 2.5: A collaboration diagram.

The use of subject-oriented programming at the operating system level can bring several benefits in regard to the subjective views it sustains. However, it may be a problem to express relationships that crosscut subjects in this model. For instance, the definition of priorities for the process subject may have implications for the management of messages in the communication subsystem. This kind of collateral effect, or system-wide property, is not easily expressible by means of subject-specific composition rules.

2.2.5 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) was introduced by Kiczales [KLM$^+$97] to deal with non-functional properties of component-based systems. Most of the current component-based development strategies concentrate on defining and composing functional units, and do not properly address the representation of non-functional properties captured during system design. In these strategies, properties such as synchronization, error handling, and security are usually expressed as small code fragments scattered over several components. Developing and maintaining such a kind of entangled code constitutes one of the biggest problems in component-based software engineering, for it ruptures with important concepts like encapsulation and cohesion and compromises the software quality. Furthermore, a component that is hardwired to a specific environment will hardly be reused in another.

Aspect-oriented programming captures non-functional properties in reusable units called “aspects”. Aspects are specified in *aspect-oriented languages* and *woven* with components by *aspect weavers* to generate the target system (figure 2.7). This process

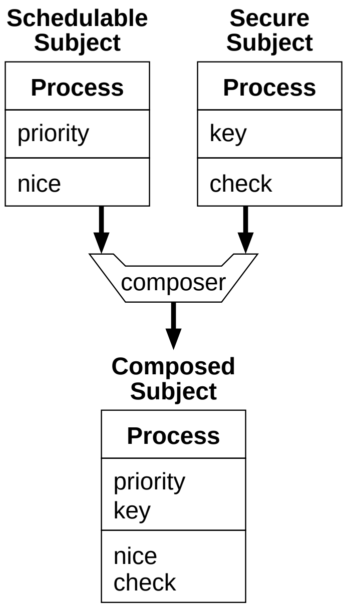

Figure 2.6: Subject-oriented composition.

of combining aspects and components is instrumented by *join points*, which are elements of the component language semantics understood by the aspect program and used for coordination.

One of the major problems with aspect-oriented programming is the difficulty of separating aspects from components during design, and subsequently implementing aspect-independent components. As mentioned before, some non-functional properties spontaneously emerge during design, but a true aspect-oriented program is supposed to go much further in order to reach aspect independence. Moreover, an aspect-oriented design method, that besides aspects also addresses the traditional issues of functional components, is yet to be unveiled. Another open issue regards the absence of mechanisms to check for semantic preservation during the weaving process—depending on the complexity of join points, it is possible that accidental semantic modifications occur [Szy97].

The PURE [BSPSS00] system at the University of Magdeburg uses aspect-oriented programming to streamline the operating system, firstly getting rid of unnecessary components, but also modifying the source code of components in order to optimize them for the conditions established by users. For example, dynamically bound references to objects that are known not to vary are replaced by static binding.

2.2.6 Multiparadigm Design

More than a design method, *Multiparadigm Design* is the realization that no single paradigm can cover all peculiarities of all domains: sometimes distinct paradigms have to be combined in order to achieve a successful design. Innumerable combinations of paradigms have been proposed for different domains [Hai86]; however, the combinations that involve object-orientation are of special interest for this thesis. Object-oriented design methods are suitable to model most elements in an operating systems, but a few elements simply do not fit correctly in the object model.

Each designer has a particular opinion about which elements fit in the object model

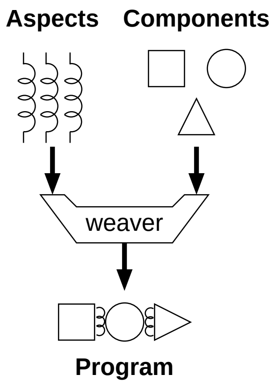

Figure 2.7: Aspect-oriented composition.

and which are better represented in some other way. A few even insist that "everything" fits in the object model. However, the representation of some operating system elements is quite controversial. For example, when one considers accessing the CPU's control registers, sending a message to the CPU object sounds just natural, but for the more generic perspective, the CPU is one of those ubiquitous elements, that exists without needing to be represented. Another contentious element is the process scheduler. Some designers model it as an object, some as a class operation of class process, and some simply prefer to see it as an algorithm outside the object world. Multiparadigm design gives designers a chance to combine paradigms in order to solve this kind of conflict.

Some programming languages have been conceived to support the implementation of multiparadigm designs, the most famous of them being C++. Although commonly defined as an object-oriented language, C++ was introduced by Stroustrup [Str86] as a multiparadigm language. Besides accommodating most object-oriented concepts, multiple inheritance inclusive, C++ supports a variety of other paradigms, including structured, family-based, generic programming, and static metaprogramming.

Coplien [Cop98] explored the multiparadigm characteristics of C++ to define a multiparadigm design method that fuses object-oriented design with family-based and structured design. The method guides an implementation-aware design, revealing the language constructs that better match each paradigm.

2.3 Implementing Customizable Designs

The development of software as an assemblage of reusable components has been sought for decades. Successive announcements have been made of methods that could produce the desired reusable components, including modular programming in the seventies, object-oriented programming in the eighties and uncountable extensions to both paradigms in the nineties. Only recently, however, the construction of software systems based on reusable, sometimes configurable, components began to prove effective. Even so, software engineers still have to agree about a common *component-based software development methodology*. For the time being, they have to rely on a combination of methods and techniques, each contributing with elements that apply better to this or that situation.

This section considers the implementation of customizable operating systems based on reusable components, trying to identify the pros and cons of each method.

2.3.1 Software Components

The growing interest on the development of software systems as arrangements of reusable components has recently raised an intense discussion about what a software component is, and what it is not. Similar discussions seem to arise among the software engineering community every time a technology achieves success, and forces whoever adopts the term to state his/her opinion about the controversy. Next, definitions for *software component* proposed by prominent authors are presented in order to render a brief snapshot of the controversy.

> "A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction."
>
> (Grady Booch [Boo87])

> "By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming and correspond to the components in the building industry."
>
> (Ivar Jacobson [JCJO93])

> "Reusable software components are self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status."
>
> (Johannes Sametinger [Sam97])

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."
>
> (Clemens Szyperski [Szy97])

Booch extends the traditional concept of module to include the object-oriented notion of abstraction. For components in the realm of object-orientation, his definition also recognizes that a class is often an inadequate construct to encapsulate a reusable abstraction (a module is adequate). Jacobson's definition is quite vague and concentrates on composition during system programming, at language level. Sametinger's definition is wider and includes not only pieces of software implementation, but of software description too. He also stresses the importance of clear interfaces and proper documentation. Szyperski, among the selected authors, has the most restrictive definition, whereas it requests a component to be independently deployable and to be subject to composition by third parties. He also states that "a corollary of this definition is that software components are 'binary' units that are composed without modifications" [Szy97]. This excludes systems that adopt source code generation techniques to configure and/or adapt components at deployment time.

The discussion about the use of software components in the construction of customizable operating system, as addressed in this dissertation, does not intend to exclude this or that approach. Therefore, restrictive concepts are of little interest. Booch's and Sametinger's definitions of reusable software component are in tune with the author's personal opinion, but, because they may also exclude some approaches addressed by the operating system community, a wider dictionary definition will be taken. The Oxford English Dictionary [Oxf92] defines a component as:

> "any of the parts of which something is made"

Despite the conceptual clash, the absence of a consensual definition for software component does not seem to prevent the technology from being successfully used. Other open questions seem to have greater impact, amongst them:

- What is the ideal size for a component?
- How should components be composed?
- How should components be adapted?
- How to grant that a component preserves its individual semantics after being composed?
- How to grant that a component composition matches the requisites defined for the system as a whole?

Notwithstanding several recent advances, none of these questions has been conclusively answered. They will be considered next in the realm of customizable operating systems.

2.3.2 Component Granularity

Independently from the development methodology adopted, component granularity always plays an important role, since it directly affects reusability. In order to be successfully reused, a component must fulfill the user's expectations about it, primarily delivering the expected functionality, but also observing requirements such as performance, usability, maintainability, and configurability. Component granularity directly influences all these metrics. On the one hand, a coarse-grained component will very likely include functionality that will not always be used, affecting performance and configurability (unused functionality often becomes overhead and yet has to be configured for proper component deployment). On the other hand, a large set of fine-grained components may escape the user's ability to conduct composition, affecting usability and maintainability [SP94b].

There is not, and probably there will never be, a rule to define the ideal granularity of software components, but being aware of its implications may help to reach a compromise

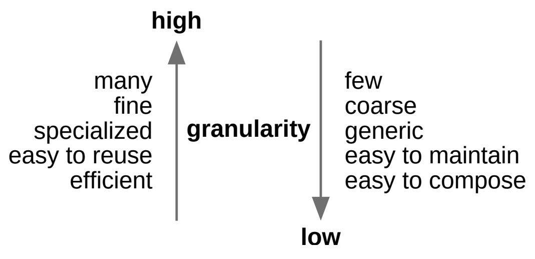

Figure 2.8: Component characteristics versus granularity.

between size and amount that will enable effective reuse[2]. The way granularity influences other component characteristics is illustrated in figure 2.8.

Alternatives to handle the granularity compromise involve hierarchy and automation. Sometimes complex components can be built on simpler ones, leading to hierarchical implementations and making the granularity issue more manageable [Szy97]. Moreover, the advent of tools that support automatic composition will enable components to shrink in size and grow in number without affecting usability and maintainability.

In the operating system scene, device driver modules and servers are still the most common fashion of component deployment. Such components are usually coarse and, despite of some configurability, are not specialized to any particular operation mode. In other words, there are no special versions of these components that implement specific functionality subsets. For example, if one wishes to use an input/output device for output only, one is not only forced to take unnecessary input functions, but also a non-optimized version of output functions (they are not optimized because they have been implemented to operate in the presence of input) [CSP91, And92, PAB$^+$95].

As of today, there are few component-based operating systems. From them, two have quite divergent opinions about component granularity. The component libraries in the FLUX operating system toolkit, developed at the University of Utah [FBB$^+$97], comprise relatively coarse components, basically device drivers and complete subsystems. Several of these coarse components have recognized performance flaws and are aimed at "take off" support for new systems. Proper components are expected to replace them as the system matures. In contrast, the PURE system at the University of Magdeburg [SSPSS98] defines a large set of fine-grained components that can be composed to form low-level abstractions. PURE recognizes that configuring such a large set of components is beyond most users' grasp and is currently working on tools to automate the process [BSPSS00].

[2]Coplien [Cop98] remarks that extremely large components usually result from poor application domain partitioning.

2.3.3 Component Composition

Software components are of no value if they cannot be composed to yield a properly functioning system. Achieving a properly functioning composition, in turn, demands a predictable component interaction mechanism that ensures the preservation of each component's semantics in the presence of others. Several such mechanisms have been proposed for applicative software, but few of them explicitly consider the peculiarities of composition at system-level, so adaptations are usually required. Because the operating system constitutes the most elementary form of run-time support, relying on advanced run-time language constructs for system composition is usually not possible. Besides, the intrinsics of an operating system challenges for composition strategies that are able to handle special circumstances such as the enforcement of properties that cross-cut component boundaries, the interaction of components that reside in different domains, the handling of asynchronous events generated by the hardware, etc.

Subsequently, the most relevant composition mechanisms traditionally deployed at the application-level will be described together with the implications of adopting them at system-level.

2.3.3.1 Interfaces

The *interface* of a component gathers the signatures of the operations that can be invoked on it, serving as a kind of "service contract" between its clients and providers. On one side, such contracts define what clients can expect from a component and how they can use it. On the other side, they define what providers have to implement to meet the contracted services. Therefore, interfaces constitute the most elementary mechanism to support component composition, upon which more complex mechanisms can be built.

Interfaces on their on, however, are not enough to grant a properly functioning composition, since they do not ensure semantic preservation in relation to external dependencies. For example, a component conceived to interact in a single-threaded environment might perform incorrectly in the presence of multiple threads, even though the interface has been properly observed. Furthermore, the operations in an interface are restricted to specify functional aspects of components. Concerns about performance, availability, and other non-functional aspects are not adequately expressible via interfaces.

2.3.3.2 Contracts

In the sense of object composition, a *contract* is a kind of extended interface that, besides the syntactic part corresponding to operation signatures, also includes a semantic part concerning behavioral aspects of components. The semantic annotations in a contract may be formal or informal. Informal annotations are restricted to document the semantics of a component, while formal annotations can be used to validate a composition.

Helm [HHG90] and Holland [Hol92] consequently explored the utilization of formal contracts as means to support object composition. In their approach, a contract is defined as follows:

> “... a contract defines the behavioral composition of a set of communicating participants. Each contract specifies the following important aspects of behavioral compositions. Firstly, it identifies the participants in the behavioral composition and their contractual obligations. Contractual obligations consist of *type obligations*, where the participant must support certain variables and external interface, and *causal obligations*, where the participant must perform an ordered sequence of actions and make certain conditions true in response to these messages. Through causal obligations, contracts capture the behavioral dependencies between objects. Secondly, the contract defines invariants that participants cooperate to maintain. It also defines what actions should be initiated to resatisfy the invariant, which as a matter of course during program execution will become false. Lastly, the contract specifies preconditions on participants to establish the contract and the methods which instantiate the contract.”
>
> (Richard Helm and Ian M. Holland [HHG90])

Although contracts have the benefit of generating object compositions that can be formally verified, the effort demanded to formally specify components mostly restricts its use to safety critical applications.

2.3.3.3 Design Patterns

Design patterns are catalogs of solutions to recurring problems in the object-orientation scene, among them object composition. The widely accepted catalog introduced by Gamma et al. [GHJV95] describes each design pattern with four attributes: name, problem description, solution description, and the consequences of adopting the respective solution. One important achievement of this catalog was the establishment of a ”taxonomy” for object-oriented elementary architectures that may ease the interaction between software developers. Patterns are also used in combination with other strategies, in particular with component frameworks [Joh92], to support composition.

The ADAPTER pattern (figure 2.9), for instance, can be used to adapt an existing component (`Adaptee`), which has the expected functionality but an incompatible interface, to match the interface defined in a given framework (`Target`). `Adapter` intercepts the messages exchanged between `Client` and `Adaptee` and makes the necessary adjustments to support the correct interaction between them. Components that otherwise would be incompatible can be properly composed in this way.

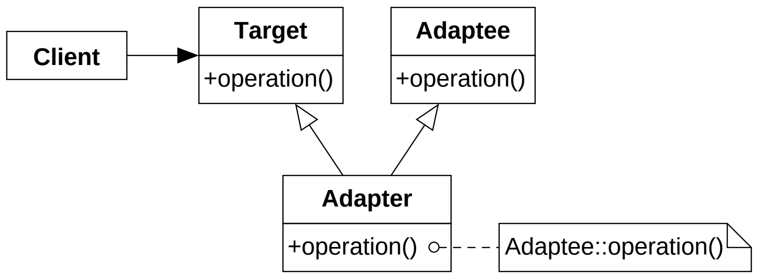

Figure 2.9: The ADAPTER design pattern.

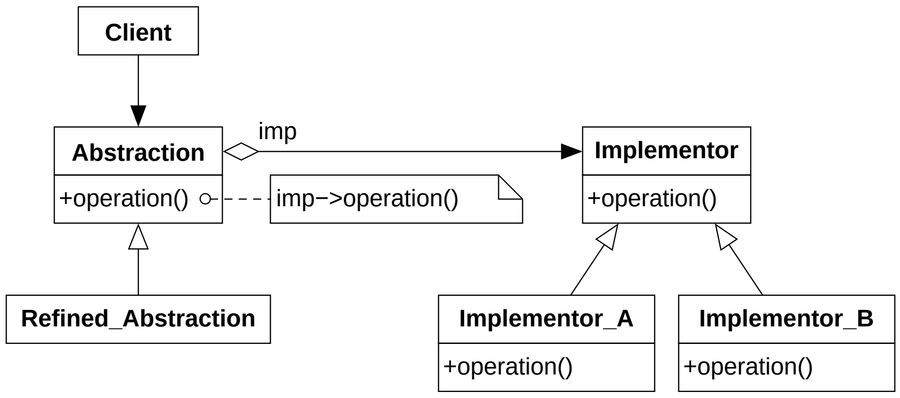

Figure 2.10: The BRIDGE design pattern.

Another example is the BRIDGE pattern (figure 2.10), which can be used to decouple abstraction from implementation so that both can be independently extended by subclassing. Bridges can be used to plug components (`Implementor_A` and `Implementor_B`) to a framework defined in terms of abstractions (`Abstraction`).

2.3.3.4 Frameworks

An *object-oriented framework* [JF88, Deu89] is an arrangement of classes that captures a reusable, usually domain-specific, design. Some of the classes in a framework are abstract and open for implementation inheritance, while others are concrete and encapsulate reusable implementations. Some frameworks also supply default implementations for abstract classes, so clients only have to implement the classes (or methods) that do not fit.

In a *whitebox framework*, existing functionality is reused and extended by inheritance and overriding. It is so called because clients need to know the internal structure of the framework in order to use it. Conversely, in a *blackbox framework* [Joh97, Lar00], or *component framework*, the relationships between components are defined exclusively in terms of their interfaces, so that components can be reused without modifications. Composition is done by selecting and plugging components to the framework (figure 2.11).

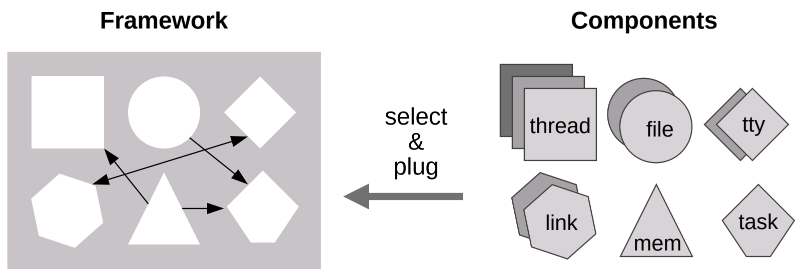

Figure 2.11: A component framework.

As Szyperski [SV98] observes, when compared to mechanisms in which components are simply "wired" together, a component framework has the important advantage of being able to enforce system-wide properties like reliability, availability, security, and scalability. Even if individual components present such properties, arbitrary compositions will seldom preserve them. However, the inclusion of specific elements in the framework can sustain this kind or property.

A component framework is an attractive alternative to support composition at system-level. However, some component frameworks, especially those defined in terms of design patterns, make extensive use of dynamic binding. In order to avoid severely compromising performance, an operating system framework must rationalize the use of expensive language constructs, reserving them to imperative cases.

The CHOICES [CIM92] system at the University of Illinois at Urbana-Champaign defines a collection of *whitebox* frameworks, one for each of its subsystems. Default implementations for the abstractions on each framework are provided, facilitating the construction of new systems. Default abstractions can be overridden or specialized through inheritance. The X-KERNEL [HP91] at the University of Arizona defines a framework for implementing network protocols in a similar way. A *blackbox* framework is used by the FLUX project [FBB^{+}97] at the University of Utah to guide the linking of object files (components). A more elaborate kind of component framework, that uses generative programming techniques, will be presented in section 4.5.1.

2.3.3.5 Collaborations

Collaborations are the components in the realm of *collaboration-based design* [section 2.2.3]. A *collaboration* is defined by a set of objects and an interaction protocol that specifies the roles of each object in the collaboration. A collaboration-based system is thus a composition of independently definable collaborations.

Several strategies have been proposed to implement collaborations in such a way as to preserve the original design. VanHilst and Notkin [VN96b] express roles as parameterized classes (C++ class templates), which are in turn composed to yield the collaborating classes. The process is guided by class definitions analogous to the type equations used by Batory in the GenVoca model [BO92]. The use of templates for composition considerably reduces the occurrence of dynamic binding in the resulting system, hence enhancing performance. Scalability, however, is a weak point in this approach, since the composition of each individual role has to be specified.

Smaragdakis [Sma99] extended VanHilst and Notkin approach by mapping each collaboration into a single parameterized class with a nested class for each of its roles. Collaborations are composed in a hierarchical way by successively specifying one as parameter to the next. This yields a composition scheme that directly matches the type equations of GenVoca and requires less effort to specify compositions. Mezini [ML98] work on *adaptive plug-and-play components* focuses on implementing collaboration-based designs as component frameworks.

2.3.3.6 Static Metaprogramming

Metaprograms[3] are programs that represent and manipulate other programs or themselves. A *static metaprogram* is a metaprogram that runs before the code it produces is set to run. The most classical static metaprograms are compilers and preprocessors, which manipulate an input program in order to translate it into another language or to modify its structure. Nevertheless, static metaprogramming is also used in programming languages that support parts of the input program to be evaluated at compile-time, the so-called multilevel languages [GJ97]. The C++ programming language, for instance, supports static metaprogramming through mechanisms such as class and function *templates*, expression evaluation in constant initialization, and function *inlining* [Str97, Vel95, Pes97].

In order to illustrate the case for static metaprogramming using C++ templates, the natural number factorial calculator used by Czarnecki [CEG+98] is reproduced in figure 2.12. The template `Factorial` recursively multiplies its argument, while the template specialization `Factorial<0>` finishes the recursion. When invoked, this factorial function is executed by the compiler, which simply includes the function result (an integer) in the generated code. Therefore, the code generated by the compiler for the following two lines is identical:

```
cout « Factorial<7>::RET « '\n';
cout « 5040 « '\n';
```

In the context of component-based systems, static metaprogramming can be used to support efficient composition when deployed in combination with some of the strategies described previously in this section. A static metaprogram can take components and composition rules as input and generate the corresponding program during the compilation process. When compared to composition assisted by external tools, i.e., tools that are loosely integrated with the language used to specify components, a static metaprogram has two main advantages: performance and correctness. A composition produced

[3]The Greek prefix “meta” stands for “after” or “beyond” and is used to denote a shift in level.

```
template<int n>
struct Factorial { enum{ RET = Factorial<n − 1>::RET ∗ n }; };

template<>
struct Factorial<0> { enum { RET = 1 }; };
```

Figure 2.12: A statically metaprogrammed factorial calculator.

by a static metaprogram can perform better than a tool-based composition because it is carried out along with the compilation of components, extending the possibilities for optimizations. A carefully designed composition scheme can avoid most run-time overhead. Composition conducted by a static metaprogram also present benefits regarding correctness, whereas the metaprogram is written in the same language as the components and is subject to syntactic and semantic verifications by the same compiler.

As an example of how a static metaprogram can manipulate the input program, consider the profiler in figure 2.13. It can be used to measure the lifetime of any object without modifying the respective classes. The parameterized class `Profiler` takes class `Target` as parameter and specializes it in such a way that a timer is started when objects are created and stopped when they are destroyed. One specialization of `Profiler` is created for each target class, but as it is handled at compile-time, no extra code is generated but the necessary to measure the lifetime.

```
#include <package.h>

namespace Profiled_Package
{
    template<class Target>
    class Profiler : public Target
    {
    public:
         Profiler () {  live (); }
        ~Profiler () {  die (); }
    private:
        void live ();     // t0 <− current time
        void die ();      // t <− current time − t0
        Time t0;
    };

    typedef Profiler<Package::Class_A> Class_A;
     // ...
    typedef Profiler<Package::Class_Z> Class_Z;
}
```

Figure 2.13: A statically metaprogrammed profiler that measures the lifetime of objects.

Modifications on the clients' side could be avoided as well, for instance, with the use of name spaces and conditional compilation[4]. If the original set of classes is defined in one name space and the profiled one in another, then clients can select the desired version by including the proper header file and "using" the proper name space as shown bellow:

```
#ifdef PROFILED
#include "profiled_package.h"
using namespace Profiled_Package;
#else
#include "package.h"
using namespace Package;
#endif
```

Defining a component framework based on static metaprogramming techniques could eliminate much of the overhead in conventional frameworks [SV98]. Indeed, static metaprogramming has been explored to support component composition in several approaches other than frameworks, including collaboration-based design [VN96a, ML98, Sma99] and subject-oriented programming [OKH$^+$95]. It has not been intensively explored in aspect-oriented programming probably due to the presumption that aspects are written in aspect languages, but it could certainly be an aid for an aspect weaver.

2.3.4 Component Configuration

The previous sections addressed the customization of operating systems by selecting specific components to take part in a system instance. This section considers the possibilities of using generic programming techniques to customize every single component in order to achieve the expected global configuration. Real systems will more likely combine both approaches, first selecting a set of components and then configuring each of them accordingly.

Generic Programming (GP) [MS89, Gog96, JLMS98, Aus99] is a programming discipline that promotes reusability by means of parameterization. Perhaps the most notorious example of generic programming is the C++ *Standard Template Library* (STL) [Pla95], which decouples algorithms from the data structures on which they operate. STL orthogonal design allows programmers to use STL data structures with their own algorithms, and to use STL algorithms with their own data structures. As a template library, STL is mostly handled at compile-time, thus profiting from numerous compiler optimizations.

Generic programming can also be used to make ordinary software components more configurable. These *generic components* are able to adjust their behavior according to externally defined parameters. Because generic components are instantiated at compile-time, customizing them usually does not affect their efficiency. When a generic component is instantiated with a particular set of parameters, a concrete component is generated.

[4]A class "wrapper" able to make `Profiler` fully transparent will be discussed later in section 3.8.3.

```
template<int n_res, class Resource>
class Allocator
{
public:
    Allocator ()
        { for(int i = 0; i < n_res; i++) used[i] = false ; }

    Resource* alloc() {
        int i ;
        for(i = 0; ( i < n_res) && used[i]; i++);
        return (i == n_res ) ? 0 : ( used[i] = true, &resource[i ]);
    }

    void free(Resource* res) {
        int i ;
        for(i = 0; ( i < n_res) && (&resource[i] != res ); i++);
        if(i != n_res) used[i] = false;
    }

private:
    bool used[n_res];
    Resource resource[n_res];
};
```

Figure 2.14: A simple generic resource allocator.

An example of generic component is depicted in figure 2.14. This simple resource allocator is able to allocate and reclaim any kind of resource, as long as information about the maximum number of units of that resource is supplied[5]. The parameterized class `Allocator` takes two parameters: `n_res` and `Resource`. The first stands for the maximum number of resources of the corresponding type, while the second designates the resource type. A statically allocated array of resources `resource` is created for later allocation, and the `used` bitmap tracks resource utilization.

Indeed, this example goes beyond parameterized programming and uses C++ static metaprogramming features to completely embed the allocator into the client (no function calls are generated at all)[6]. A concrete allocator that allocates up to 16 buffers of 4 Kbytes could be instantiated as follows:

```
Allocator<16, char[4096]> allocator ;
```

[5]An allocator with a fixed number of resources may look strange for an application programmer, but is certainly in order for a system programmer. In an operating system there are plenty of cases in which the maximum number of resources is known in advance (e.g., SCSI devices in a bus), or in which the cost of dynamic allocation is not worth paying (e.g. `i-nodes` in a file system).

[6]If code replication becomes a problem, a set of allocation functions that ignores the type of resource being allocated and operates solely on pointers and size information can be gathered in a base class for the generic allocator.

Although not the only language to support generic programming, C++ has been the choice for many generic component developers, which rely mostly on the template mechanism. Template parameters in C++ are not restricted to types; they can also be constants, data structures, or even stretches of code. This makes component construction very flexible, but not without its price. The resulting source code may become incomprehensible if many parameters are used, or if unconventional parameters are used.

2.4 Summary

A customizable operating systems is constructed to be configured in accordance with specific requirements dictated by the hardware, users, or applications. In such a system, configuration must be explicitly controllable, so that clients can designate the features that will be active in the system at a given time. This is usually achieved with mechanisms that adjust system control parameters and/or manipulate system parts.

The configuration mechanisms available to an operating system to accomplish customizability can be classified as static or dynamic. Static configuration mechanisms are deployed to modify system characteristics before it begins to execute, influencing the process of link edition, compilation, or source code generation. In contrast, dynamic configuration mechanisms are deployed while the system is running. Common mechanisms include process creation, kernel extension, and reflection.

Static configuration mechanisms have advantages on performance and resource utilization, for they are not included on the final system. On the other hand, dynamic configuration mechanisms confer the system a higher degree of flexibility, for they support the system in reconfiguring itself to deal with unpredicted situations. In general, dynamic configurability agrees with generic computing systems, in which applications are not known in advance, while static configurability matches up dedicated computing systems, which comprise a small set of applications (often a single one) whose requirements are understood in advance. Knowing applications in advance allows dedicated systems to profit from static configuration mechanisms without degrading flexibility, since the system can be pre-configured to include all the features that will be needed by the application(s).

An operating system modeled as an arrangement of software components can reach a high degree of customizability. Configuration is thus accomplished through the selection, adaptation, and combination of software components. Methodologies such as family-based design, object-oriented design, collaboration-based design, subject-oriented programming, aspect-oriented programming and multiparadigm combinations of these have the potentiality to guide the design of customizable operating systems by means of concepts like variability and commonality analysis, domain decomposition, subjective views of domain, and aspect separation.

During the development of component-based software, the definition of component interfaces is a fundamental activity. A component interface specifies the services provided by a component, yielding a kind of service contract between its clients and suppliers. Such software components can be composed with a variety of techniques, including design patters, frameworks, and generative programming, while generic programming techniques can assist internal configuration. Nevertheless, the development of customizable operating systems as collections of components introduces a series of new obstacles. Engineering software components of adequate granularity, adaptability, and composability, as well as mechanisms to assemble them efficiently and correctly, remains a challenge, indicating that more adequate design and implementation techniques are necessary.

Chapter 3

Application-Oriented System Design

In chapter 2, several mechanisms to support the construction of customizable operating systems have been described. From that study, one could conclude that, by adopting state-of-the-art software engineering techniques, it is possible to construct run-time support systems with a high degree of customizability. In particular, dedicated systems could benefit from statically configurable components to achieve the desired customizability without having to pay the high price of dynamic reconfiguration.

Nevertheless, a repository of system-level components and a mechanism to arrange them together may not be enough to comply with the requirements of dedicated computing systems. Ordinary run-time support systems frequently fail to deliver the expected services, or the expected service quality [And92, Mah94, DBM98]. Slicing one such a system in a set of components will certainly not improve the case for applications.

This chapter presents a novel strategy to build component-based run-time support systems that can be tailored according to the requirements of particular applications. This strategy is defined around the *Application-Oriented System Design* method, which covers the development of application-oriented operating system from domain analysis to implementation.

3.1 The Case for Application-Orientation

Historically, operating systems have been constructed aiming at abstracting physical resources in a way that is more convenient for the hardware than for applications. Undoubtedly, the monolithic structure of early operating systems contributed to this scenario, for it must have been very difficult, if not impracticable, to customize such systems in order to accomplish the demands of particular applications. The notion that applications have to be adapted to the operating system was so established. Since then, a succession of standardizations has been freezing application program interfaces, thus helping to consolidate the situation. Contemporary operating systems are suffocated by thick layers of standards that keep the majority of the conceivable improvements unreachable for applications [Pik00].

Besides failing to accompany the natural evolution of applications, many operating systems also fail to keep updated as regards software engineering. By comparing articles published in acclaimed scientific journals of both fields, operating systems and software engineering, one could even conclude that they do not concern the same science. Perhaps this is also a consequence of extreme standardization, whereas there is little room for new software engineering techniques in the constrained scenario of operating systems. Astonishingly, this is a very complex software scenario, which spans from hardware to applications, and would greatly profit from modern software techniques. However, in reality, the obsolescence of the techniques deployed in some systems comes out to impact applications.

Even modern operating systems that support customization have difficulties to match up with application requirements. Mainly because they usually target the design of configurable features, the heart of any customization strategy, on standard compliance and on hardware aspects, and do not adequately address application requirements. Hence, an application programmer may be invited to select features such as POSIX or TCP/IP compliance, or to select drivers for a certain hardware device, but seldom will have the chance to express that traditional features such as process and memory management are not needed. Interestingly, this is one of the most typical operating modes for a dedicated computing system.

Building an operating system as an aggregate of reusable components has the potentiality to considerably improve the situation, diminishing the gap between system and applications. Nevertheless, component-based software engineering is just a means to construct systems that can be customized to fulfill the demands of particular applications. Inadequately modeled components, or inadequate mechanisms to select and combine components, may render the extra effort of building reusable components unproductive. The goal of application-driven customization can only be achieved if the system as a whole is designed considering the fulfillment of application requirements.

Furthermore, the way customization is typically carried out in component-based operating systems makes it difficult to pair with application requirements. As a rule, customization in these systems is delegated to end users, which are assisted by some sort of tool in selecting and combining components to produce an executable system. In this case, successfully customizing the operating system becomes conditioned to the knowledge the user has about the system. Hence, user-driven customization is entangled in the balance of component granularity discussed in section 2.3.2:

- If components are *coarse-grained*, the chance of an ordinary user, i.e., a user without deep knowledge about the operating system, to successfully conduct system customization grows, but the probability that components will meet application requirements decreases proportionally.

- If components are *fine-grained*, the chance that the system will match application requirement grows, but it is likely that users will not be able to understand the peculiarities of such a large collection of components, and will probably miss the proper configurations.

Improvements in user-driven configuration have been pursued by enabling components to be selected indirectly. The LINUX system, for instance, utilizes a mechanism to select kernel components through the features they implement. Instead of pointing out which components shall be included in the system, users can select the desired system features. Features, in turn, are interrelated by dependencies and mapped into components. Nevertheless, even if LINUX kernel components are coarse-grained (they are mainly device drivers and subsystems) and will seldom satisfy the specific requirements of individual applications, selecting features from a list with approximately 700 options[1] would be considered a sordid activity by most users. A mechanism that allows applications themselves to guide the configuration process would be more appropriate.

The considerations made so far relate to applications in general; the case for dedicated computing systems is even worse. Dedicated systems execute specific sets of applications that are defined in advance, so the requirements for the operating systems are also known beforehand. Theoretically, this should enable the operating system to be rightly customized to support the application. However, when a general-purpose operating system is in scene, what usually happens is that applications get uncountable services that are not needed, but still have to implement much of what is needed.

Notwithstanding, software engineering seems to be now in such an advanced stage that it should be possible to produce an operating system that, besides scaling with the hardware, also scales with applications; that delivers all the functionality required by applications in a form that is convenient for them; and that deduces application requirements to automatically configure itself. Many of the related issues have already been addressed in the context of all-purpose computing by *reflective systems*. In order to comply with the requirements of high-performance dedicated systems, the subject is approached in this thesis from the perspective of statically configurable component-based systems.

3.2 Application-Oriented Operating Systems

The expression *application-oriented* is being introduced here to characterize operating systems that are strongly compromised with applications. A customizable operating system is said to be application-oriented if it can be customized to match the requirements of particular applications, rather than simply being customized to match standards or hard-

[1]The number of LINUX configurable kernel features has been estimated by executing the following command in a system based on kernel version 2.2.14: `grep CONFIG /usr/src/linux/configs/kernel-2.2.14-i386.config | wc -l`.

ware aspects. The configurable features of such a system are directly derived from application requirements, so that applications themselves can drive the customization process.

The following enunciate unveils the foundations of an application-oriented operating system:

An application-oriented operating system is only defined with regard to the corresponding application(s), for which it implements the necessary run-time support that is delivered as requested.

It also yields the following corollaries:

- An application-oriented operating system, in contrast to an all-purpose one, is always associated with a particular application (or with a particular set of applications). This intentionally restricts the scope of the proposed definition to dedicated computing systems, for which the corresponding applications are known in advance. It also favors static configuration mechanisms in detriment of dynamic ones.
- An application-oriented operating system has to implement the run-time features that are necessary to support the application, neither less nor more. It is straightforward to understand why the system should not implement less than the necessary features, but for an application-oriented system, delivering more than what is necessary should be avoided as well. Included features that are not used by the application, besides affecting the average system quality (performance, resource utilization, configurability, etc), may complicate the matching with application requirements.
- The features implemented by an application-oriented operating system must be delivered to the corresponding application as they have been requested. Hence, externally visible interfaces must be defined in the context of applications to be realized in the context of the operating system with the necessary internal adjustments.

The main contribution of this dissertation is a design method that addresses the issues involved in the construction of application-oriented operating systems. The proposed design method, namely *Application-Oriented System Design* (AOSD), covers the development of system-level software from domain analysis to implementation, reusing state-of-the-art software engineering concepts and techniques when possible, and defining novel ones when necessary. A new design notation has been suppressed in favor of the *Unified Modeling Language* (UML) [BRJ99], though extensions and simplifications to this language will be frequently practiced.

3.3 Domain Analysis and Decomposition

Although no design can go further than its perception of the corresponding problem domain, domain analysis and operating systems are subjects that seldom come together. The fact that the operating system domain is basically made of conventions, many of which established long ago in projects such as THE [Dij68] and MULTICS [Org72], seems to have fastened it to a "canonical" partitioning. This partitioning includes abstractions such as process, file, and semaphore, and is taken "as-is" by most designers. Indeed, it is now consolidated by standards on one side and by the hardware on the other, leaving very little room for new interpretations.

Notwithstanding this, revisiting the problem domain during the design of a new operating system would probably reveal abstractions that are better tuned with contemporary applications. For example, the triple (process, file, message passing) could be replaced by persistent communicating active objects. Actually, most run-time platforms feature this perspective of the operating system domain through a middleware layer such as CORBA [OMG01] and JAVA [SUN01]. However, the middleware approach goes the opposite direction of application-orientation, whereas it further generalizes an already generic system.

Nevertheless, even if one endures domain analysis knowing that decomposition will have to be carried out respecting the boundaries dictated by standards, programming languages, and hardware, there is at least one important reason to do it: to avoid the monolithic representation of abstractions. If an application-oriented operating system is to be the output of design, capturing application-specific perspectives of each abstraction and modeling them as independently deployable units, as suggested by *subject-oriented programming* [section 2.2.4], is far more adequate than the monolithic approach. After all, the product of domain engineering is not a single system, but a collection of reusable software artifacts that model domain entities and can be used to build several systems.

3.3.1 Application-Oriented Domain Decomposition

An application-oriented decomposition of the problem domain can be obtained, in principle, following the guidelines of *object-oriented decomposition* [section 2.2.2]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the famous "fragile base class" problem [MS98], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *family-based design* [section 2.2.1] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, as in *incremental system design* [section 2.2.1.1], but this is no longer an imperative rule. For example, instead of modeling connection-oriented as a specialization of connectionless communication (or vice-versa), what would misuse a network that natively operates in the opposite mode, one could model both as autonomous members of a family.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from the execution scenarios being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented operating system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *aspect-oriented programming* [section 2.2.5], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects[2]. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

The phenomenon of mixing scenario aspects and abstractions seems to happen spontaneously in most other design methods, so learning to avoid it may require some practice. Perhaps the most critical point is the fact that the majority of operating systems are conceived with an implementation platform in mind, which is often better understood than the corresponding problem domain. In principle, there is nothing wrong in studying the target platform in details before designing the system, actually it may considerably save time, but designers tend to misrepresent abstractions while considering how they will be implemented in the chosen platform. In an application-oriented system design, this knowledge about implementation details should be driven to identify and isolate scenario aspects. In general, aspects such as identification, sharing, synchronization, remote invocation, authentication, access control, encryption, profiling, and debugging can be represented as scenario aspects.

Building families of scenario-independent abstractions and identifying scenario aspects are the main activities in application-oriented domain decomposition, but certainly not the only ones. The primary strategy to add functionality to a family of abstractions is the definition of new members, but sometimes it is desirable to extend the behavior of all members at once. Specializing each member would double the cardinality of the family. Application-oriented system design deals with cases like this by modeling the extended functionality as a *configurable feature*. Just like scenario aspects, configurable

[2]The representation of scenario aspects will be discussed later in section 3.5, for now it is only important to avoid modeling unnecessary family members.

features modify the behavior of all members of a family when activated, but, unlike those, are not transparent. One could say that scenario aspects have “push” semantics, while configurable features have “pull”.

A configurable feature encapsulates common data structures and algorithms that are useful to implement a family’s feature, but leave the actual implementation up to each family member. Abstractions are free to reuse, extend, or override what is provided in a configurable feature, but are requested to behave accordingly when the feature is enabled.

The case for configurable features can be illustrated with a family of networks and features such as multicasting, in-order delivery, and error detection. If new family members were to be modeled for each such a feature, a family of 10 networks subjected to 10 features could grow up to 10^{10} members. Modeling this kind of feature as a scenario aspect is usually not possible either, since its implementation would have to be specialized to consider particular network architectures.

Another relevant issue to be considered during domain decomposition is how abstractions of different families interact. Capturing ad-hoc relationships between families during design can be useful to model reusable software architectures, helping to solve one of the biggest problems in component-based software engineering: how to tell correct meaningful component compositions from unusable ones. A reusable architecture avoids this question by only allowing predefined compositions to be carried out. For example, one could determine that the members of a family of process abstractions must use the family of memory to load code and data, avoiding an erroneous composition with members of the file family. In application-oriented system design, reusable architectures are captured in component frameworks that define how abstractions of distinct families may interact. Although such frameworks are defined much later in the design process, taking note of ad-hoc relationships during domain decomposition can considerably ease that activity.

An overview of the process of application-oriented domain decomposition is presented in figure 3.1. In summary, it is a multiparadigm domain analysis method that promotes the construction of application-oriented operating systems by decomposing the corresponding domain in families of reusable, scenario-independent abstractions and the respective scenario aspects. Reusable system architectures are envisioned by the identification of inter-family relationships that will later build component frameworks.

3.3.2 An Example of Domain Decomposition

In order to illustrate the process of application-oriented decomposition, the domain of high-performance communication in clusters of workstations will be decomposed next. This experiment has been realized with the collaboration of RWCP Laboratory for Parallel and Distributed Systems at GMD that currently investigates means to facilitate the construction of grid and mesh-based scientific applications [GGD01]. The members of this group played the role of domain experts for a simplified use case analysis [JCJO93]

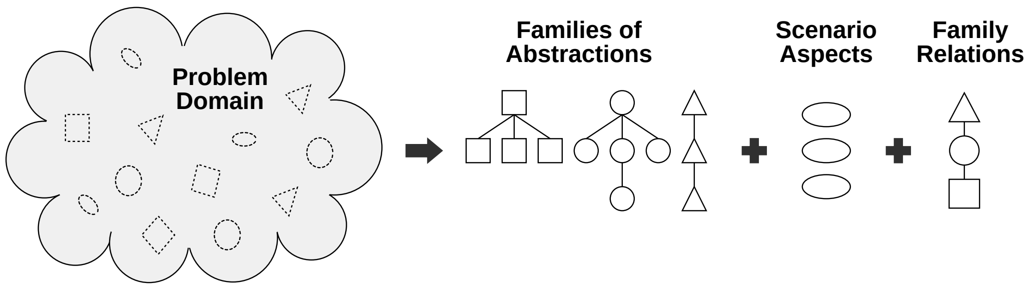

Figure 3.1: An overview of application-oriented domain decomposition.

that produced the vocabulary in table 3.1.

active messages	ARMR/ARMW	asynchronous send
ATM	capability	channel
collective operations	datagram	debugging support
DSM	end-point	Ethernet
group communication	high bandwidth	link
location transparency	low latency	mailbox
MPI	multithreading	Myrinet
port	protection	PVM
reliability	SCI	sharing
stream	synchronous send	user-level

Table 3.1: A vocabulary regarding the domain of high-performance communication in clusters of workstations.

Based on this perception of the problem domain, application-oriented decomposition begins by applying commonality/variability analysis to identify the families of abstractions that build up the domain. For example, `Myrinet`, `SCI`, `Ethernet`, and `ATM` could be gathered in a family of networks, with a member for each network technology. The family would abstract what these networks have in common (e.g. the ability to transmit and receive data), while members would make evident the technological differences among them that are relevant to applications. Similarly, `port` and `mailbox` could start a family of communication end-points, while `datagram` and `stream` could be gathered in a family of communication strategies.

Some of the keywords in table 3.1, however, clearly do not designate abstractions. `High bandwidth` and `low latency`, for instance, are non-functional requirements that have to be taken in consideration during the design of the system as a whole. This kind of *global property* is quite different from those that can be enabled and disabled at users' wish. The support for `multithreading`, for instance, is a property that affects all abstractions in the domain, but not permanently. It could be modeled as a *scenario aspect*. Configurable properties that originate from an execution scenario but that are only meaningful to specific families can be modeled as *private scenario aspects* for those families. For example, an `asynchronous` communication system could be obtained exclusively by requiring the family of communication strategies to join the "asynchronous scenario". Properties such as `reliability` and `multicast`, which regard the behavior of abstractions in the family of networks, are too dependent from the physical network to be implemented as part of a scenario. They are better modeled as *configurable features*.

From the keywords in table 3.1, three do not fit in the cases considered above: `user-level`, `MPI`, and `PVM`. The first reveals the knowledge the contributing group of experts has about the advantages of an implementation technique, while the other two designate desired *Application Program Interfaces* (API). As long as possible, these two last should be considered as what they really are: APIs. It should be possible to design an adaptation layer to support these APIs on top of any well-designed communication system, which could also offer more sophisticated native APIs.

A schematic representation of the decomposed domain is presented in the following diagrams. Figure 3.2 shows the topmost family diagram, including the families of abstractions and scenario aspects that were identified during decomposition. Each family of abstraction will be described subsequently. The multithread (`m-thread`), multitask (`m-task`), and multiprocessor (`SMP`) scenario aspects concern the synchronization of concurrent object invocation in the respective scenarios. The `protected` scenario aspect enforces access control to abstractions, while the `shared` aspect coordinates simultaneous access. The location transparent (`loc.transp.`) scenario aspect imposes an indirect invocation mechanism that hides the location of abstractions. The `debug` scenario aspect produces run-time information about abstractions that can be used to debug the application.

The family of networks is depicted in figure 3.3. It has four members: `Ethernet`, `ATM`, `Myrinet`, and `SCI`. Two configurable features have been considered: `multicast` and `reliability`. The first is enabled when the application requests for collective operations, while the second is enabled to build a reliable scenario. It is important to notice that, whatever construct is used to implement these configurable features, it will likely have to be specialized for each family member, since architectural differences between them may render portable solutions inefficient.

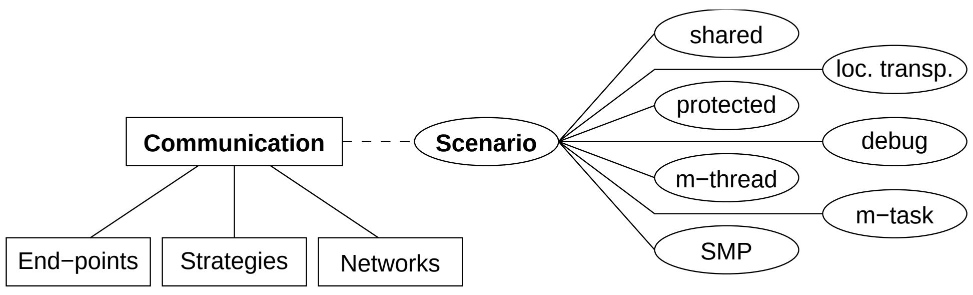

Figure 3.2: Families of abstractions and scenario aspects in the domain of high-performance communication.

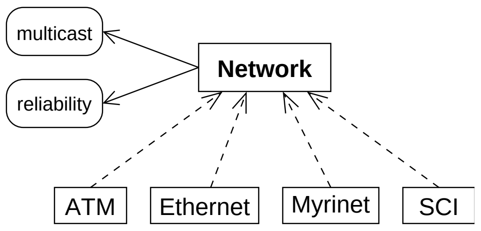

Figure 3.3: A family of networks in the domain of high-performance communication.

The family of communication strategies is shown in figure 3.4. It features six members that support the following strategies: connectionless communication (`Datagram`), connection-oriented communication (`Stream`), asynchronous write to a memory region on a remote host (`ARMW`), the respective read operation (`ARMR`), active messages (`AM`), and distributed shared memory (`DSM`). Both `AM` and `DSM` are modeled in terms of `ARMW`. `DSM` also reuses `ARMR`. Two private scenario aspects have been modeled for this family: `synchronous` and `asynchronous`. The first scenario modifies the semantics of the communication mechanisms in such a way that the send operation (or equivalent) only concludes when the destination application completely receives the data, while the second implies in the send operation returning as soon as possible. Besides, two configurable features have been modeled: group communication (`group comm.`) and `buffering`. The first is used to implement collective operations, while the second supports the family in implementing the `asynchronous` scenario.

The third family of abstractions is depicted in figure 3.5. It concerns end-points for the family of communication strategies, thus being the family that will be used more often by applications. It includes the following members: 1-to-1 channel (`Connection`), n-to-1 channel (`Port`), n-to-n channel (`Mailbox`), active message handler (`AM Handler`), and memory segments for asynchronous remote operations (`ARM Segment`) and distributed shared memory (`DSM Segment`). No private scenario aspects or configurable features were modeled.

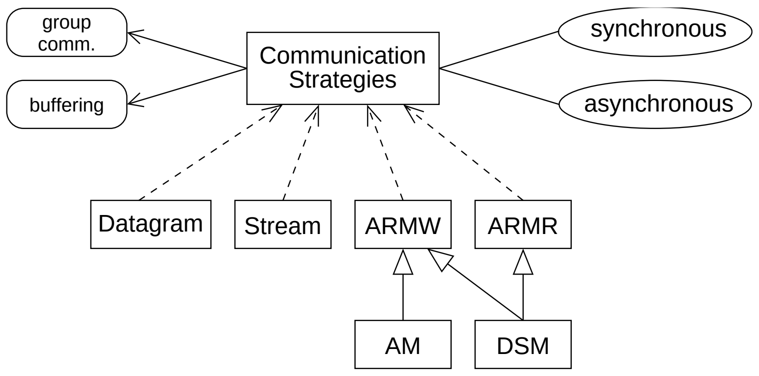

Figure 3.4: A family of communication strategies in the domain of high-performance communication.

During the decomposition of the domain of high-performance communication in clusters of workstations, some fragments of reusable system architecture became evident: the family of communication strategies *uses* the family of networks, and both are *interfaced* by the family of end-points. This would be recalled when defining a component framework for the high-performance communication system in question.

The following sections describe how a preliminary design specification produced during application-oriented domain decomposition can be refined to yield a detailed specification that can be used to guide the implementation of an application-oriented operating system.

3.4 Families of Scenario-Independent Abstractions

During application-oriented domain decomposition, scenario-independent abstractions are identified and grouped in families according to what they have in common. The subsequent phase in application-oriented system design is the refinement of these abstractions in order to define the software components that will implement them.

The first refinement to be performed is the adjustment of the magnitude of components in relation to abstractions. The basic idea is to keep a 1-to-1 relation, preserving conformity with the domain. However, it may happen that some entities in the domain are too coarse-grained to be modeled as a single component (e.g. a file system), or to primitive to be directly exposed to application programmers (e.g. an FPU). As explained earlier, excessively large components will likely miss the target on application-orientation, while excessively small components will likely flop on user-friendship.

The balance "as simple as possible, but still application-ready" is the goal to be pursued in this phase of design. Larger, more complex abstractions can be implemented as an assemblage of components, while elementary entities, that do not characterize abstractions, can be embedded in other components according to the criterion of functional cohesion. Data structures and algorithms that are common to several families of abstractions can be collected in *utility classes* to be reused during the construction of actual components. Although a precise "algorithm" to reach such a balance does not exist, chapter 4

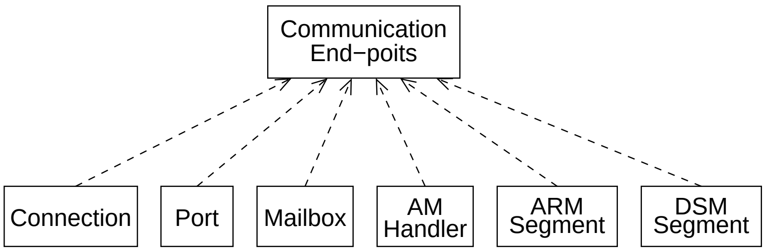

Figure 3.5: A family of communication end-points in the domain of high-performance communication.

brings an extensive case study on application-oriented system design that shall elucidate many of the issues related to abstraction granularity.

For all practical effects, components in an application-oriented system design will always be seen as direct emanations of abstractions, hence some granularity adjustments may have to be propagated back to the documents of domain analysis. This correspondence between abstractions and components helps to assure that an application programmer will never be called to interact with abstractions that are not directly being used, nor confronted with the fact that unnecessary abstractions have been included in the system. If a component encapsulates several abstractions and the application makes use of a single one, it is likely that the programmer will still have to configure the remaining abstractions, and that these extra abstractions will consume resources that otherwise could be used by the application.

After components and abstractions have been matched, each family of abstractions can be refined considering details about the structure and behavior of its members in order to ratify their interfaces. Ideally, application-orientated abstractions should be delivered as *Abstract Data Types* (ADT) [LZ74]. In this case, each abstraction would be exported through an interface that clearly identifies its responsibilities. The definition of these interfaces is fundamental for a successful design, whereas they constitute the main interaction point between system and application programmers.

Some designers defend that interfaces should not specify constructors, since in principle a class can realize several interfaces, possibly leading to conflicts. However, the class that represents an abstraction in the realm of application-oriented system design is a rather special one: it is conceived to collect the classes that implement the abstraction in a construct that realizes the interface of that abstraction. These classes always realize a single interface. Furthermore, some abstractions become considerably easier to understand and to use when associated with distinct initialization semantics, which could be expressed by constructors in their interfaces.

The interface of a member of a family of threads outlined in figure 3.6 illustrates the case for constructors in interfaces. When creating an object of type `A_Thread`, the programmer can choose from a variety of initial states that differ on aspects such as whether the thread is immediately eligible for execution or not, where the execution will begin, and what priority it will have. In situations like this, application-oriented designers are encouraged to specify constructors in abstraction's interfaces.

Regarding the organization of abstractions in families, one of the first aspects to be observed is whether the commonalities that substantiated the creation of the family can be modeled as a basic abstraction from which all other abstractions derive. If so, the family becomes a traditional object-oriented class hierarchy, with the basic abstraction as a *base class* and the remaining members as specializations of that. In this case, each family member defines a *subtype* of the basic abstract data type, thus enabling the entire family to be handled as a single *polymorphic* abstraction.

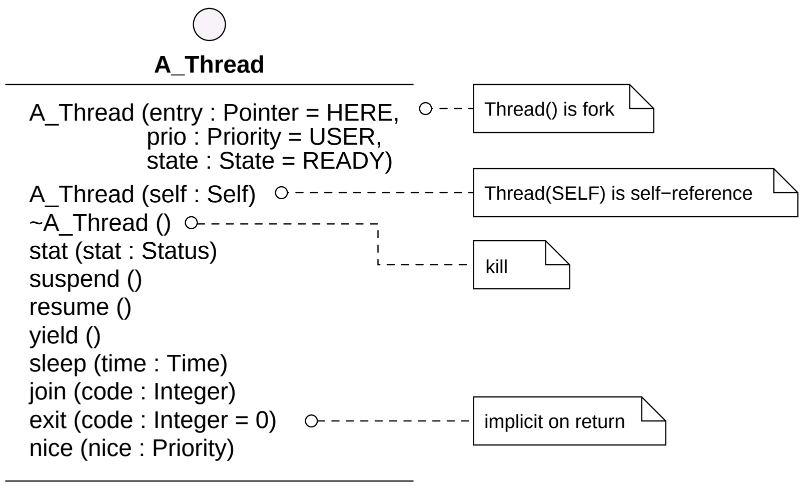

Figure 3.6: The interface of a member of a family of thread abstractions.

Contrarily to what a “pure” object-oriented designer might suppose, this is not the only (and sometimes not even an adequate) alternative to represent a family of abstractions. Consider for instance a family of synchronization mechanisms comprising two well-known abstractions: *condition variables* and *semaphores*. Does what they have in common characterize a basic abstraction? Alternatively, should one of them be taken as the basic abstraction? In any case, how would the drastic differences between them be accommodated in the base class? As the term suggests, *polymorphism* concerns the reconciliation of the multiple “forms” of an abstraction, providing a single interface for entities of different types. However, what happens if these multiple forms, though sharing semantic and functional aspects, do not fit under a common type?

As a multiparadigm design method, application-oriented system design does not presuppose abstractions in a family to be polymorphic. If the commonality of a family does not spontaneously characterizes a basic abstraction, then it can be represented by classes collected in a “common package” and made available to be reused by independently defined family members via *aggregation* or *subclassing* instead of *subtyping*.

Some families of abstractions may feature members that are mutually exclusive, i.e., that cannot be deployed at the same time. For example, a family of process abstractions could feature a member to support a single thread per process and another to support both single and multiple threads. These two abstractions would likely be mutually exclusive, since the cost of dynamically switching between them would be much higher than the cost of selecting the abstraction that supports multithreading and leaving the feature inactive for most of the time. This kind of family excludes polymorphism even if the family derives from a common basic abstraction, since only one of its multiple “forms” can be used at a time.

The construct used to encapsulate the common elements of a family is also the ideal place to store the classes used to implement configurable features, since it is always in-

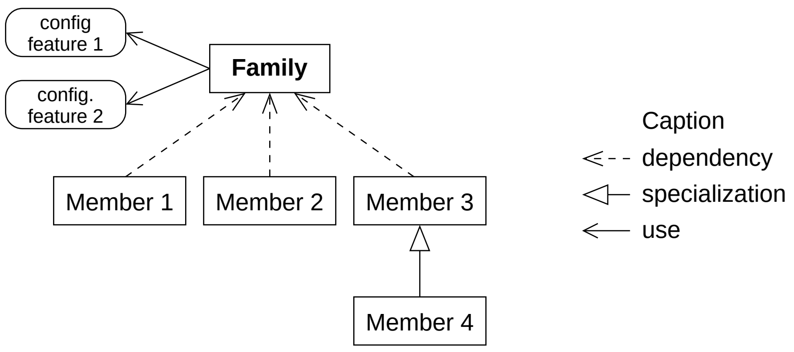

Figure 3.7: Notation to represent a family of abstractions.

corporated by all members. Configurable features dictate particular conditions to abstractions, but unlike scenario aspects, are not transparent. When an abstraction is notified that a configurable feature has been enabled, it has to modify its behavior accordingly. Nevertheless, the implementation of such configurable features across the members of a same family can usually be accomplished deploying common data structures and algorithms, which can be modeled as classes in the family's common package. These classes can be reused, specialized, or overridden by each family member.

A typical family of abstractions can be represented using the notation illustrated in figure 3.7. The correspondence of abstractions and components in an application-oriented system design allows *family diagrams* to fuse elements from both logical and component view. A family is represented as a tree, with members connected through dependency relationships, and themselves depending or specializing each other. The family's common package is implicitly represented, unless configurable features have been modeled for the family. In this case, they appear connected to the root of the tree by means of use relationships.

While refining the specification of families and their members, it is also important to consider inter-family relationships. Families often rely on abstractions from other families to deliver the contracted services. For example, a family of process abstractions may rely on a family of synchronization abstractions to coordinate concurrent execution. Whenever possible, inter-family relationships should be expressed without making the case for a particular member of the supplier family, so that configurability is not constrained. If a family makes a superfluous choice for a mutually exclusive member of another family, it implicitly configures that family, preventing the application programmer from doing so, and reducing the number of valid system configurations.

Suppose for instance that family `B` of figure 3.8 is mutually exclusive (i.e. only one of `B1`, `B2`, and `B3` can be used at a time). Defining relationship `R2` would implicitly exclude members `B2` and `B3` from been used when family `A` is deployed. Evidently, this is not an issue if family `A` really depends on member `B1`. Indeed, failing to represent this relationship would make room for erroneous configurations in this case. However,

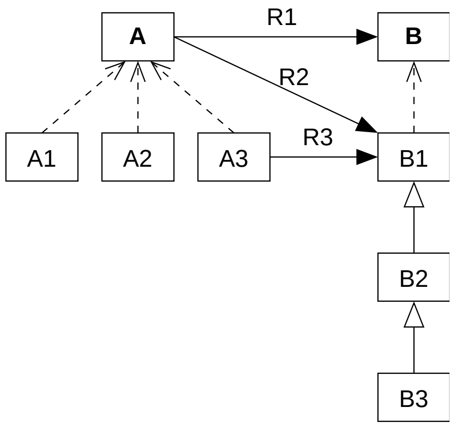

Figure 3.8: Inter-family dependencies.

it is important to carefully consider if a restrictive relationship like this really holds for the whole family. For example, if the dependency on member `B1` concerns exclusively to member `A3`, while other members of family `A` would behave correctly with any member of family `B`, a relationship `R1` could express the inter-family dependency and be overridden by `R3` for member `A3`.

Another situation that could bring families to interact is the identification of common software artifacts that are of interest to several families. Containers such as lists and queues, for instance, appear in a large number of system-level abstractions. Modeling such artifacts in the context of one abstraction and establishing inter-family relationships just for the sake of code sharing is certainly not an option: the loss of quality due to improper coupling is eminent. Defining additional “utility abstractions”, where common artifacts are collected for posterior reuse, is not an alternative either, since such abstractions would break fidelity with the domain. A satisfactory answer to the question would be to model these common artifacts as utility classes that are stocked up in a library. In this way, they can be shared among clients without setting up any relationship between them.

At this point, application-oriented system design converges into object-oriented design to produce detailed specifications of each family’s common package, its members, its configurable features, and class utilities. Among others, the methods proposed by Rumbaugh [RBLP91], Jacobson [JCJO93], and Booch [Boo94] could be used for this purpose.

3.5 Scenario Aspects

In the process of application-oriented domain decomposition, abstractions are specified avoiding dependencies from envisioned execution scenarios, while scenario aspects are

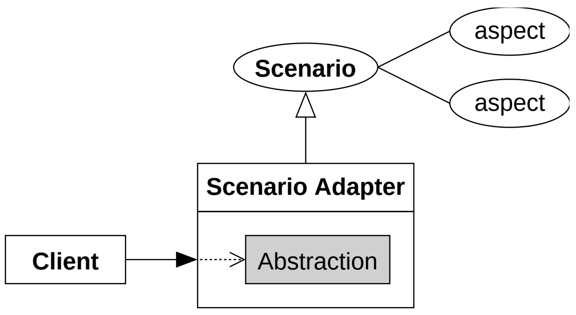

Figure 3.9: The general structure of a scenario adapter.

captured in separated constructs. By doing so, an explosion of scenario-dependent abstractions is avoided at the same time the degree of reusability of abstractions is increased. However, for this scheme to be effective, the constructs used to capture scenario aspects must be modeled in such a way that it becomes possible to enforce abstractions the conditions dictated by a given scenario without having to explicitly modify them.

One can think of a *scenario* as a construct that incorporates several scenario aspects in the same way abstractions do with configurable features. Once the desired scenario aspects have been selected, the scenario can be applied to abstractions with a *scenario adapter*. A scenario adapter is a kind of agent that engulfs a scenario-independent abstraction in order to mediate its interaction with a scenario-dependent client (figure 3.9). In this way, abstractions acquire the properties needed to perform in a given scenario without having to be modified.

In application-oriented system design, scenarios and the respective aspects are represented in *scenario diagrams*. In principle, scenario adapters are defined on a per-scenario/per-family basis, so it is unnecessary to represent them either in scenario diagrams or in family diagrams. However, if some exotic families (or family members) require scenario adapters to be specialized, then it is convenient to represent them in the corresponding family diagram. A scenario that only regards to a single family is represented in the corresponding family diagram too.

Scenario aspects can be either structural or behavioral. The first kind modifies the structure of abstractions, appending some scenario-specific data structure to them. The unique global identifier that is assigned to abstractions in order to support remote invocation in a distributed scenario is an example of structural aspect. Such a kind of scenario aspect can be modeled as shown in figure 3.10, with the structural aspect being first incorporated by the corresponding scenario via aggregation and later by the scenario adapter via inheritance.

The second kind of scenario aspect modifies the behavior of abstractions, enforcing scenario-specific semantics on them. Attaching a lock to abstractions so that concurrent invocations of their operations get coordinated is an example of behavioral aspect. Abstractions would be locked just before an operation is invoked and released just after it is concluded. A generic representation of behavioral aspects is shown in figure 3.11. Just like structural aspects, behavioral aspects are first incorporated by the corresponding scenario via aggregation and later by the scenario adapter via inheritance. However, unlike the former, behavioral aspects implement operations that are automatically invoked by the scenario adapter to establish the conditions required by the scenario before the operation is invoked on the abstraction. The corresponding operations of all behavioral aspects are usually called in-order when entering and leaving the scenario, but more sophisticated schemes can be devised.

A new scenario, and consequently a new scenario adapter, has to be defined for a family of abstractions only if incompatibilities among scenario aspects arise. Otherwise, a set of scenario aspects can be simultaneously activated to shape an execution scenario for the abstractions in the system.

3.6 Inflated Interfaces

In application-oriented system design, families of abstractions are sometimes handled as single entities. Configurable features and scenario adapters, for instance, are defined in terms of families and not of their members. Viewing a family as a single entity may be convenient to application programmers as well, since it would enable them to postpone the decision of which family member will be used until enough arguments have been collected. Assigning each family a single interface, that represents all its members at once, would produce the desired single-view and would enable programmers to write their applications with a higher degree of abstraction. Adequate realizations of these interfaces could be selected just before generating the application-oriented operating system.

Consider, for example, the case in which an application programmer first identifies a given member A of a family of abstractions as being the most adequate for an application, but during implementation realizes that writing the application in terms of a member B would have been more appropriate. Replacing each appearance of A for B in the source code could be avoided if the programmer had written the application in terms of an interface that congregates the services of both member A and member B, and if both members had been implemented respecting the semantics of this interface. In this case, binding the interface to member B would be enough to produce the desired effect.

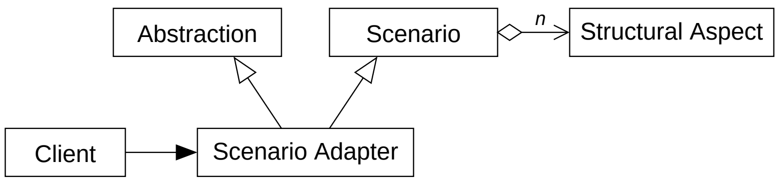

Figure 3.10: The representation of a structural scenario aspect.

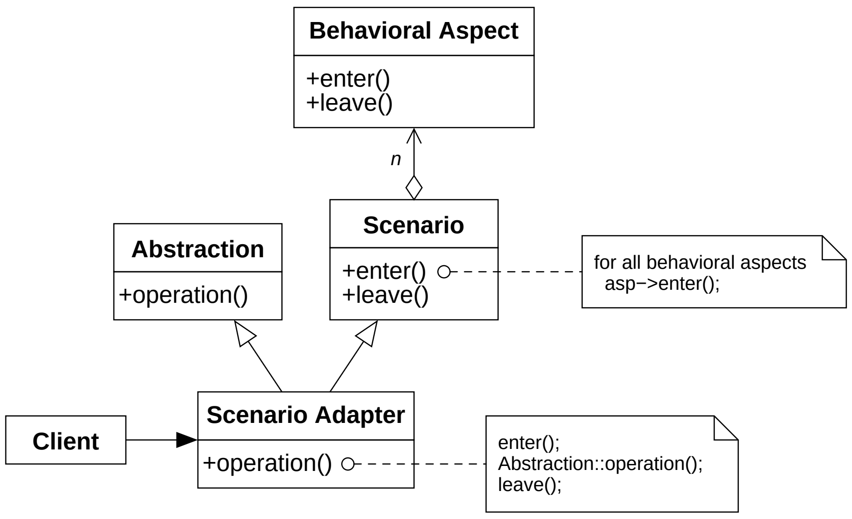

Figure 3.11: The representation of a behavioral scenario aspect.

Therefore, besides exposing the individual interface of each member of a family of abstractions, an application-oriented system also delivers an *inflated interface* that exports the family as though a “super” component, that implements all responsibilities assigned to the family, was available. Theoretically, such an interface can be obtained by merging the interfaces of individual family members. However, if programmers are invited to write their applications based on these inflated interfaces, a strategy to transparently bind them to one of their realizations (i.e. a family member) has to be devised.

In order to support design based on inflated interfaces, two new relationships are being proposed: *partial* and *selective* realization. Both relationships take place between an inflated interface and its realizations, being that a realization participating in a partial realization relationship implements only a specific subset of the corresponding inflated interface, and selective realization means that only one of the realizations can be bound to the inflated interface at a time. Both relationships are depicted in figure 3.12.

Ideally, application programmers should be able to write applications entirely in terms of inflated interfaces, delegating the burden of system configuration to an automatic tool. Such a tool would syntactically analyze the source code of the application in order to determine the inflated interface subsets that have been effectively used. It could then bind each inflated interface to the lightest family member that realizes the required subset[3], thus producing an application-oriented operating system. In order to achieve this scenario, inflated interfaces must be carefully specified, taking in consideration the internal organization of families. In this regard, there are four basic categories of families to be considered:

[3]The complex task of ordering family members according to a cost model would have been previously accomplished by the operating system designer.

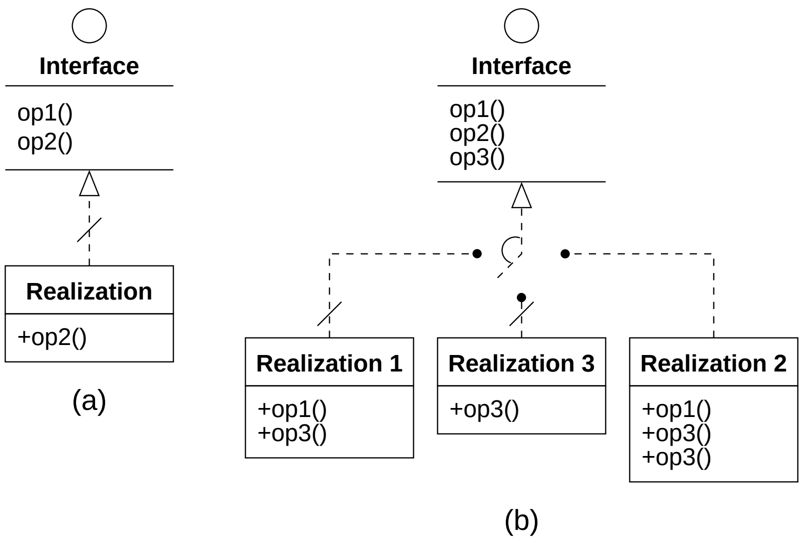

Figure 3.12: Partial (a) and selective (b) realization relationships.

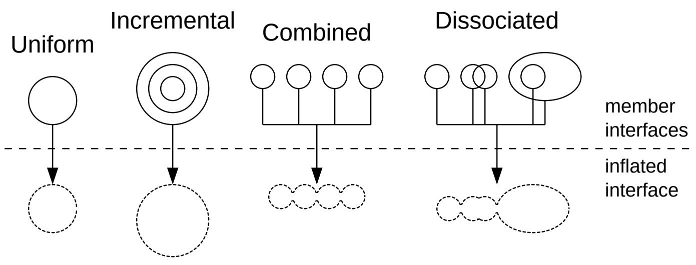

Figure 3.13: Categories of families of abstractions with regard to their inflated interfaces.

- Uniform: a family of abstractions in which all members share the same interface;
- Incremental: a family of abstractions that has undergone an incremental design, with each member being an extension of the previous;
- Combined: a family of abstractions in which members show no intersection at all, but that allows members to be automatically merged (e.g., via multiple inheritance) to produce new members with the combined functionality;
- Dissociated: a family of abstractions that does not fit in one of the preceding categories.

Figure 3.13 illustrates how the inflated interfaces of families of these four categories are produced. The inflated interface of a *uniform* family can be directly derived from the interface of any of its members, since they are all equivalent. That is, an inflated interface already exists by definition for this kind of family. Figure 3.14 shows a uniform family of CPU scheduling policies in which threads candidate to execution are registered with the

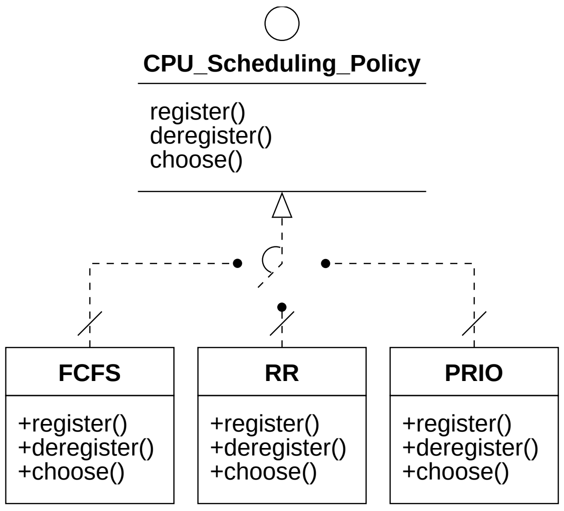

Figure 3.14: The inflated interface of a uniform family of CPU scheduling policies.

policy implementer by the thread manager. In order to select a new thread to occupy the CPU, the method `choose` is invoked. The information necessary to make a scheduling decision is obtained directly from the registered threads by invoking appropriate methods, thus enabling a uniform interface for the policy implementers. If policies are allowed to be changed at run-time, then the family should be implemented using polymorphism, making changes transparent to the thread manager. In fact, uniform families are usually polymorphic, since a common interface for the different types defined by each member abstraction already exists.

A family of the *incremental* category has its inflated interface derived from the member with the broadest interface, which encompasses all other members. An example of this kind of family is presented in figure 3.15. Each member of this family of thread abstractions adds functionality to the family by defining new operations. Operations with the same signature also have the same semantics for all family members. Member `Priority` is the most comprehensive one in the example, thus characterizing the inflated interface of the family.

The inflated interface of a *combined* family can be produced by merging the individual interfaces of all family members. This is the most flexible family organization in what concerns configurability, since it enables an eventual binding tool to perform arbitrary combinations of family members in order to match the required inflated interface subsets. Nonetheless, this kind of family is extremely difficult to model, since each member has to be designed considering the consequences of arbitrarily combining it with all other members. During the design of EPOS, the prototype system developed to validate the concepts and techniques of application-oriented system design, no families of this type have been identified.

The inflated interfaces of the former three categories, when designated by an applica-

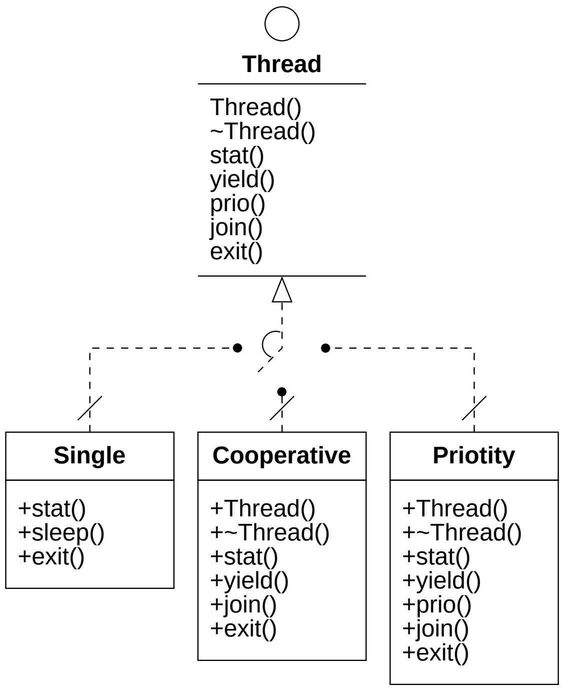

Figure 3.15: The inflated interface of an incremental family of thread abstractions.

tion, can always be bound, either to an existing realization or, in the case of a combined family, to an implicitly derived one. The inflated interface of a family in the *dissociated* category, however, may yield an adverse situation. Such an interface would also be produced by merging the interfaces of all family members, but binding to an appropriate realization may not be viable. Indeed, binding will only be possible if the interface subset designated by the application is entirely realized by a single member of the family, since members of this kind of family cannot be automatically merged. The situation of not being able to bind an inflated interface can be interpreted from two perspectives:

1. The application programmer misused the inflated interface, requesting for incompatible services that cannot be delivered by a single abstraction;
2. The abstraction that would realize the required inflated interface subset, though coherent, has not yet been implemented.

The first situation happens when an application makes the case for a family member and than requests an operation that cannot be performed on abstractions of that type—writing to a read-only file, for instance. The inability to bind the inflated interface, in this case, does not compromise the system, since it is a direct consequence of a semantic flaw in the application program. Even if one considers that the programmer was induced to error by the inflated interface, for example because it showed the constructor of a read-only file and the write operation under the same interface (figure 3.16), detecting such

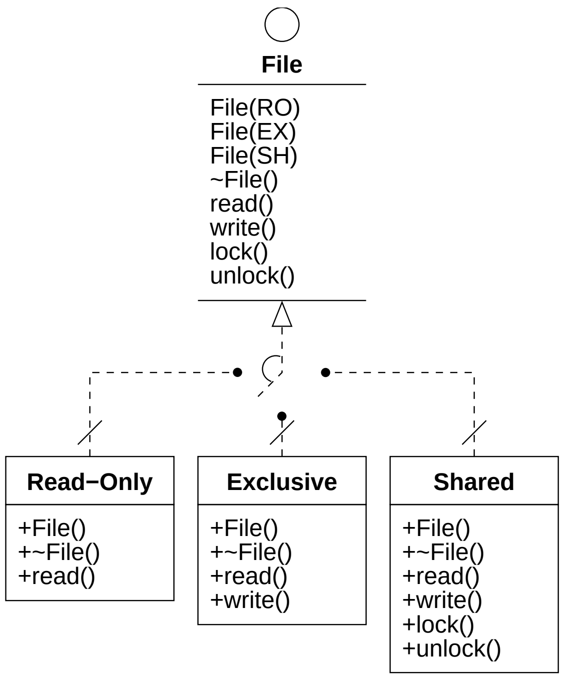

Figure 3.16: The inflated interface of a dissociated family of file abstractions.

an error during system configuration is an advance in regard to ordinary systems, which usually only detect this kind of error at run-time[4].

The second case, however, exposes a controversial aspect of inflated interfaces: an application programmer, after having been invited to write an application in terms of inflated interfaces, may be required to modify it because a needed (and announced) component is not available. This situation can be observed in the family of I/O buses depicted in figure 3.17. If an application invokes the method `scan` on an object of type `ISA`, it implicitly requests for a realization that does not (yet) exist. However, scanning an ISA bus is a logically correct operation that could be implemented either by the `ISA` member itself or by a new `Plug_and_Play_ISA` member. Situations like this have to be taken in consideration during system design so as to decide, together with users, whether the use of inflated interfaces for dissociated families is adequate or not.

Nevertheless, incidents that require applications to be modified will be more frequent in the early stages of system development, when many conceived abstractions have not yet been implemented, fading away with time. Furthermore, when such an incident happens, the operating system developers get a precise description of the missing component, what can considerably accelerate the development of the system as a whole. After all, this is not a peculiarity of application-oriented system design: the same problem would occur in a family-based design and in a pure object-oriented design. It is intrinsic to giving a set

[4]The POSIX system call `open` can be invoked to create a read-only file and nothing prevents the system call `write` to be invoked on the same file, resulting in a run-time error.

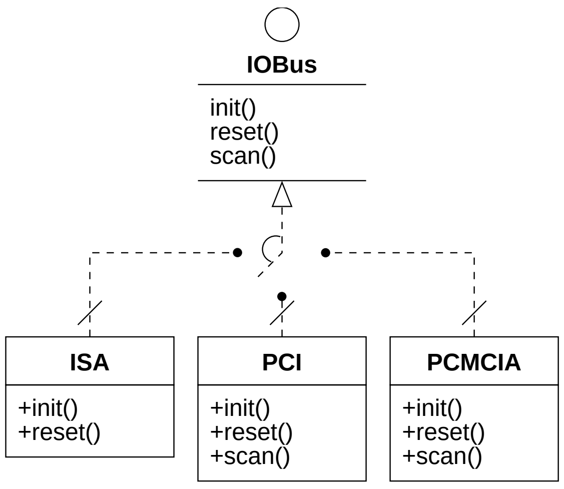

Figure 3.17: The inflated interface of a dissociated family of I/O buses.

of dissociated abstractions a common interface.

3.7 Reusable System Architectures

Along with the specification of abstraction families and scenario aspects, an application-oriented system design delivers specifications of reusable system architectures, which define how abstractions can be arranged together in a functioning system. Reusable system architectures are usually defined considering the past experience with the building systems of a certain class. After having developed some systems, or some versions of a system, for a certain problem domain, developers begin to agree on how to implement the abstractions that build up the domain, how they interact with each other, with the environment, and with applications, and how the implied non-functional requirements can be accomplished. Such an expertise can be captured in an architectural specification to be reused in upcoming systems.

Capturing reusable system architectures in a component-based system is fundamental, since a pile of components, by itself, is nothing but a pile of components. A component-based system is only achieved when components can be arranged together in an assemblage of predictable behavior. In application-oriented system design, reusable architectures begin to be modeled yet during domain decomposition with the identification of relationships between families of abstractions. These relationships are enriched by scenario constraints during the specification of scenario aspects and serve as input for this phase, which aims at producing a detailed specification of reusable system architectures in the form of component frameworks.

An application-oriented component framework captures a reusable architecture by specifying the families of abstractions that take part in a certain kind of system, as well

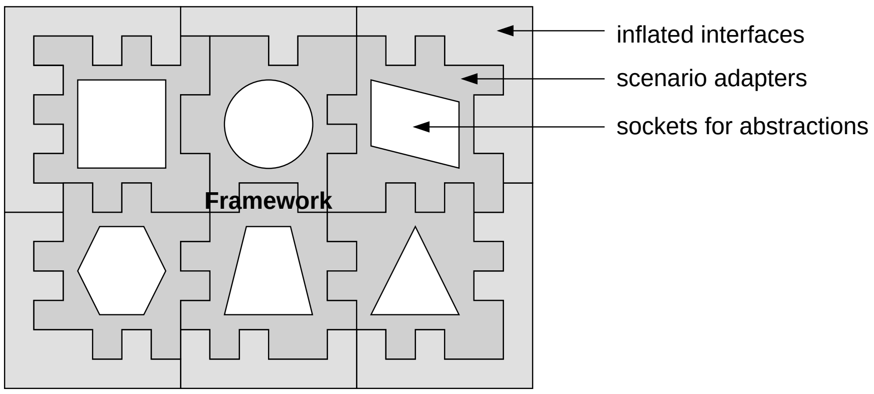

Figure 3.18: An application-oriented component framework.

as rules that guide their interaction. Systems produced by component frameworks, when compared to arbitrary arrangements of components, are less prone to misbehavior, since only compositions that have been predefined by system architects are allowed. Although component frameworks are not the unique alternative to capture reusable architectures—among others, *aspect programs*, *subjects*, and *collaborations* could also be used for this purpose—they fit perfectly with application-orientation's notion of isolating scenario aspects from abstractions by means of scenario adapters.

An application-oriented component framework could be defined as a collection of interrelated scenario adapters as shown in figure 3.18. Each scenario adapter would set up a "socket" for components of the corresponding family. Plugging components into the framework would be accomplished by binding the inflated interface of every used family to the desired family member. The way scenario adapters are arranged in the framework would define the basic architecture of resultant systems, while architectural elements that do not concern components could be hard-coded in the framework.

A component framework defined in terms of scenario adapters would also present advantages concerning *system-wide properties*, which could be modeled as scenario aspects to be enforced on components by the respective scenario adapters [SV98]. Moreover, a component framework of this kind does not require complex tools to manipulate the source code of components in order to generate a system. After all, the representation of a component framework as a socket board to which components can be plugged is well understood and accepted by users.

Figure 3.19 shows a schematic representation of a component framework that embodies a plausible system architecture for the domain of high-performance communication in clusters of workstations used as example in section 3.3.1. It illustrates the relationships between the three families of abstractions modeled: `Communication End-Point`, `Communication Strategy`, and `Network`. Firstly, it shows a mutual dependency between the families of strategies and end-points, i.e., by selecting a certain strategy, one automatically selects the corresponding end-point, and vice-versa. It also shows that a

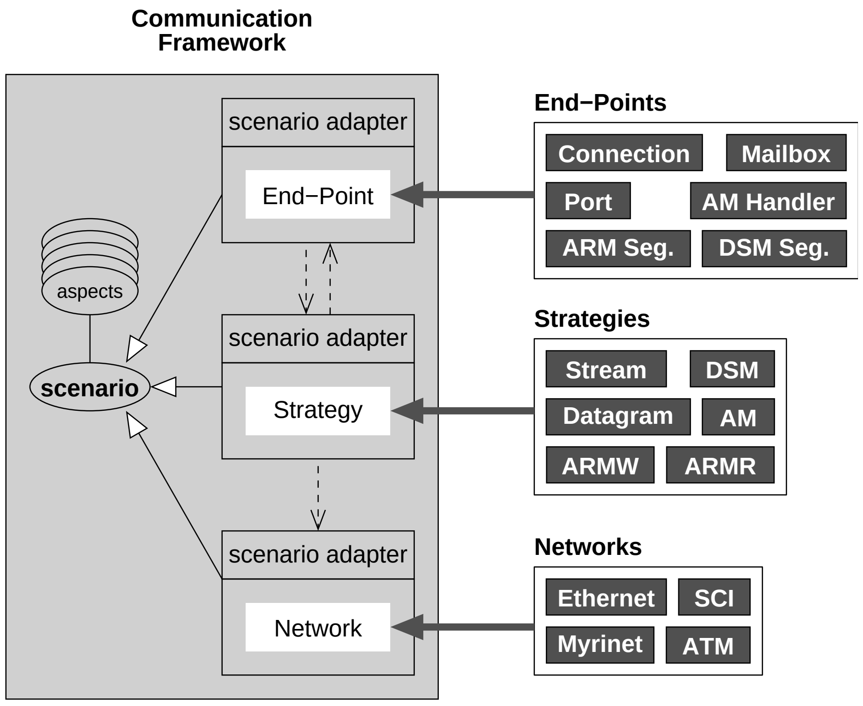

Figure 3.19: A component framework for the domain of high-performance communication.

network is *used* by the members of the communication strategy family. The respective sets of components are also shown to illustrate the "select-and-plug" organization of the framework.

An overview of the application-oriented system design method is presented in figure 3.20. In summary, it is a multiparadigm design method that promotes the construction of application-oriented operating systems by applying the process of *application-oriented decomposition* to engineer a domain as collection of *families of reusable, scenario-independent abstractions*. Scenario dependencies are modeled as *scenario aspects* that can be enforced on abstractions by means of *scenario adapters*. Families are made visible to applications through *inflated interfaces*, which export all members as though a comprehensive component was available. Reusable system architectures are captured in *component frameworks* defined in terms of scenario adapters.

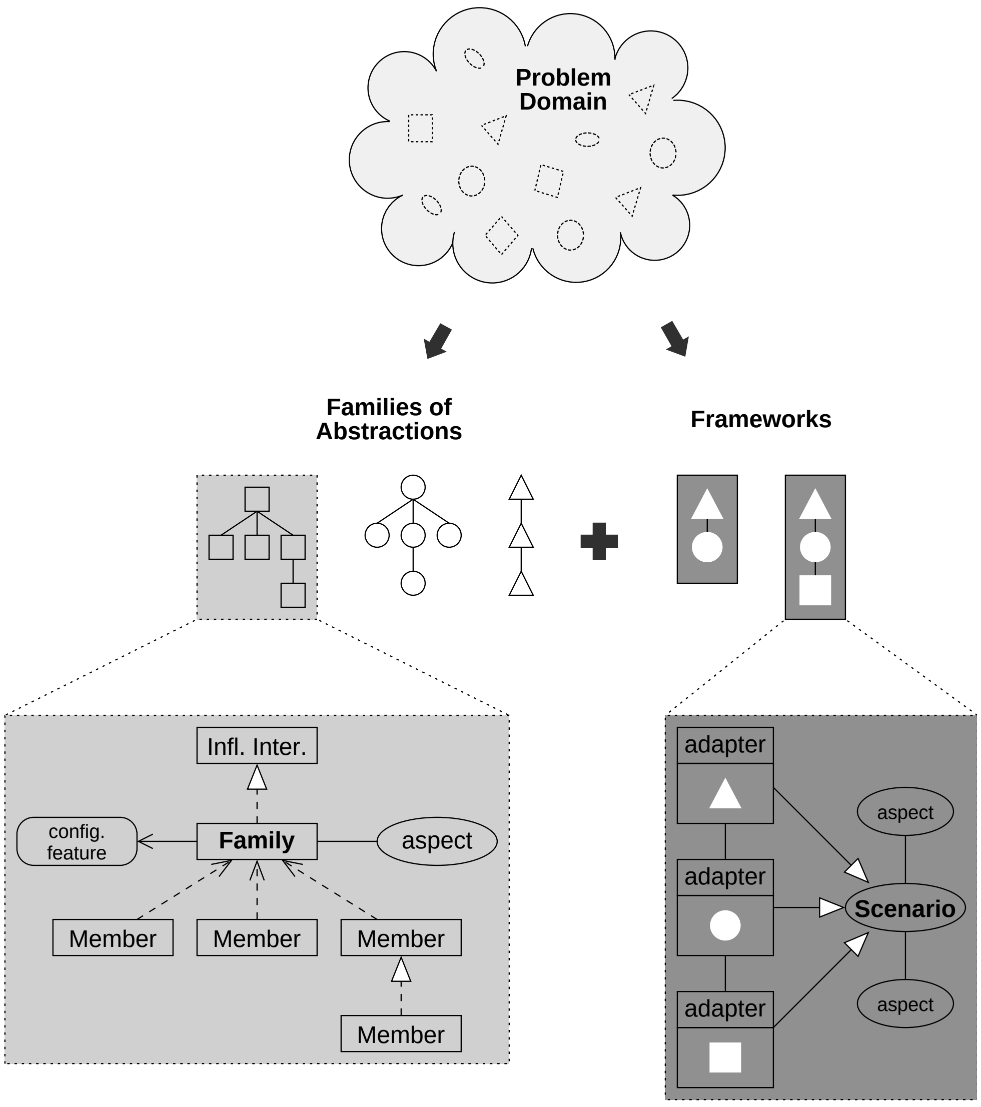

Figure 3.20: An overview of application-oriented system design.

3.8 Implementation Considerations

A system designed according to the directives of application-oriented system design can be implemented using a variety of techniques. An *application-oriented system implementation discipline* is not being proposed in this thesis because new techniques to implement components are being steadily introduced. Attempting to consolidate such an implementation discipline at this moment would certainly be precipitate. Nonetheless, several implementation techniques have been identified that can be useful to implement application-oriented system designs.

One aspect that is likely to be shared by application-oriented system implementations is the programming language. Not because there is something inherent to the application-oriented system design method that depends on a programming language, but because of the very own nature of an operating system implementation, which demands full control over the underlying hardware. The C programming language [KR84], also dubbed “the portable assembly”, supplies this demand and has been extensively used to implement operating systems, however with known deficiencies in what concerns reusability and maintainability [Bur95].

Application-oriented system design is a multiparadigm design method that has object-orientation on its core. Hence, designs produced with this method would be more smoothly implemented in a multiparadigm language defined around object-orientation. As of today, the C++ programming language [Str86] is the non-experimental language that better satisfies these requisites: it exposes the bare hardware through its C subset, and it is a multiparadigm language centered on object-orientation. Therefore, C++ is the most probable choice to implement application-oriented operating systems and will be used in the examples throughout this section.

It is also important to keep in mind that abstractions in an application-oriented system design correspond to the components that are used to assemble a run-time support system for a dedicated application, and that the decision of including a component in a system configuration is taken before it is generated. Static configuration is a premise of application-oriented system design that originates from its focus on dedicated computing systems, so components are not loaded or replaced at run-time as in a reflective system (see section 3.1). This premise also calls for efficient implementations.

3.8.1 Abstractions

Abstractions in an application-oriented system design are modeled to be independent from execution scenario aspects and to be deployed independently of a specific framework. The construct that enables such a degree of independence is the *scenario adapter*, which engulfs an abstraction in order to mediate its interaction with a scenario dependent client. In this way, abstractions can be designed and implemented without making any consideration about the scenario in which they will be used or how they will be placed in a framework. A skillfully designed system can be implemented in such a way that abstractions are "incorporated by the framework" instead of themselves "incorporating framework elements" such as header files and base classes. This counts for maximum reusability, allowing abstractions to be reused even by non-application-oriented systems.

The assumption that the framework incorporates abstractions by means of scenario adapters, and not the other way round, also enables the reuse of binary components. A header file containing the interface of the abstraction realized by the component should suffice to incorporate a binary component in an application-oriented system. Nevertheless, some of the optimizations promoted by a statically metaprogrammed framework, function call elimination for example, would be inhibited in this case.

The condition that enables a source component to be incorporated in an application-oriented framework is the same: a header file with the interface of the class that realizes the abstraction. It does not mean however that an abstraction has to be realized by a single class. It only means that there must be a class that encapsulates all other classes used to implement the abstraction, constituting a unit of instantiation for the framework.

However, the representation of such interfaces in a system implemented in C++ is complicated by the lack of a real *interface* construct in the language. Usually the public part of a class declaration is regarded as its interface, but nothing prevents data and functions from being defined at the same time they are declared. Another commonly deployed strategy is to use a "pure" class declaration, i.e. a class declaration without definitions, to designate the abstraction's interface and another class to hold its implementation. Two variants of this mechanism are depicted in figures 3.21 and 3.22. The first makes use of an *abstract class* declaration (virtual functions initialized with zero), while the second uses a class declaration with a *protected default constructor*. Both share the following characteristics:

- Interfaces are declared in a separate name space to avoid clashes;
- The illogical operation of instantiating an interface is properly prevented;
- The realize relationship is implemented by means of private inheritance, so that no subtype relationship is established between interface and implementation[5];
- Disagreements between interface and implementation are detected at compile time.

They differ in three points: instantiation avoidance, compliance enforcement, and resulting structure. The interface of figure 3.21 prevents instantiation by declaring pure virtual operations, what also forces any class realizing the interface to implement them. The one of figure 3.22, on the other hand, prevents instantiation by declaring the default

[5]C++ programmers sometimes mix up the concepts of interface and base class: the unique meaningful relationship between a class and an interface is "realization".

```
namespace Interface
{
    class Abstraction
    {
    public:
        virtual int operation1(void) = 0;
        virtual void operation2(int) = 0;
    };
}

class Abstraction: private Interface::Abstraction
{
public:
    int operation1(void);
    void operation2(int);

private:
    // Implementation declarations
};
```

Figure 3.21: An abstraction interface as a C++ pure abstract class declaration.

constructor protected and enforces compliance by declaring operations private, thus producing a compile-time error when a realization fails to adhere to the interface. The latter variant has the advantage of avoiding the virtual function call mechanism. This mechanism is not likely to be eliminated for the interface that declares pure virtual operations, because syntactically it is identical to a base class, and there is no way to tell the compiler the contrary.

Notwithstanding, both workarounds to specify abstraction interfaces in C++ have a strong negative effect on the corresponding implementations: specifying the realize relationship between interface and implementation through inheritance modifies the implementation class. This happens as a side-effect of a language property that guarantees two objects never to be allocated the same address in memory, what is achieved by associating a size greater than zero to all classes, abstract ones that cannot be instantiated and do not have data members inclusive.

Therefore, asking for `sizeof(Interface::Abstraction)` yields a number greater than zero and `sizeof(Abstraction)` a number that corresponds to the effective size of an instance of `Abstraction` plus the minimum size allocated to the interface. This may be insignificant for many abstractions, but would have disastrous consequences for a system-level abstraction that is precisely mapped across different storage units (main memory and memory on a PCI device for instance). Hence, not representing realize relationships at all is sometimes a better alternative for C++ implementations. The extra interface classes are thus restricted to document abstractions for users, while the public portion of the declaration of implementation classes assume the role of interfaces.

```
namespace Interface
{
    class Abstraction
    {
    protected:
        Abstraction () {}

    private:
        int  operation1(void);
        void operation2(int);
    };
}

class Abstraction: private Interface::Abstraction
{
public:
    int  operation1(void);
    void operation2(int);

private:
    // Implementation declarations
};
```

Figure 3.22: An abstraction interface as a C++ class declaration with a protected default constructor.

In this way, problems with objects being shifted in memory are avoided.

Another reason not to represent realize relationships concerns constructors. As discussed in section 3.4, declaring constructors in interfaces may be convenient to represent different initialization semantics for an abstraction. However, since C++ lacks an interface construct, expressing a constructor in one of the interfaces described above would force the implementation class to forward arguments to the interface, completely degenerating the notion of interface.

The common package of a family of abstractions is also ultimately delivered by a single class that is incorporated, directly or indirectly, by all its members. The most typical way to incorporate the common package in an abstraction is through subclassing (private inheritance), so that members do not become a subtype of it. Care must also be taken to avoid the replication of the common package throughout the family. A member that derives from another that already has the common package does not need to incorporate it again, and a member that derives from multiple others expect these to have incorporated the common package using virtual inheritance.

Concerning the implementation of abstractions in families, one has firstly to consider the category in which the family falls: uniform, incremental, combined, or dissociated. An example of *uniform* family of abstractions is depicted in figure 3.23. Since uniform

```
class Common
{
protected:
     Common() {}

      //  Family commonalities
};

class Member_A: protected Common
{
public:
     virtual ~Member_A();
     virtual  int  operation1(void);
     virtual  void operation2(int  i );

private:
      //  Implementation declarations
};

class Member_B: public Member_A
{
public:
     void operation2(int  i );

private:
      //  Implementation declarations
};
```

Figure 3.23: An example of uniform family of abstractions implemented in C++.

families are usually polymorphic [section 3.6], this example uses virtual functions. The commonalities of this family are gathered in class `Common`, which is incorporated by the family's basic abstraction `Member_A` (base class) via protected inheritance (no subtyping). `Member_B` specializes the family's basic abstraction, inheriting `Common` and overriding its operations at convenience.

Figure 3.24 illustrates the case for incremental families. Every new member of this kind of family adds functionality to a basic abstraction (`Member_A` in the example), often declaring new operations (like `operation2` in `Member_B`). Since the basic abstraction usually does not bear an interface capable of representing all abstractions in the family, polymorphism is mostly avoided. As with a uniform family, commonalities are incorporated by the basic abstraction and are inherit by other members. Since `Member_B` extends `Member_A`, it can also be claimed for `Member_AB` responsibilities.

An example of dissociated family of abstractions is presented in figure 3.25. The unique syntactic aspect shared by the members of a dissociated family is its common package (class `Common` in the example), which is individually incorporated by each fam-

```
class Common
{
protected:
    Common() {}

     // Family commonalities
};

class Member_A: protected Common
{
public:
    int operation1(void);

private:
     // Implementation declarations
};

class Member_B: public Member_A
{
public:
    void operation2(int i );

private:
     // Implementation declarations
};

typedef Member_B Member_AB;
```

Figure 3.24: An example of incremental family of abstractions implemented in C++.

ily member. The dissociated members of this family implement and export operations in detriment of each other (`Member_B` knows nothing about `operation1`). Note that this does not rupture with the concept of family, since members can still be strongly semantically connected (see discussion about inflated interface binding in section 3.6).

Figure 3.26 presents a combined family of abstractions. At first glance it evokes a dissociated family, however, members of this kind of family can be arbitrarily combined in order to obtain an abstraction with the corresponding functionality. `Member_AB` in the figure illustrates the case. Nevertheless, these combinations are easier to assure structurally than semantically. In respect to structure, they are enabled by having a default constructor in each member and deploying virtual inheritance to reuse `Common`. Regarding semantics, however, there is no simple strategy that can be applied to assure the correctness of such arbitrary compositions.

Combined and dissociated families are seldom polymorphic, since the discrepancies between the types of their members are difficult to conciliate. This fact is usually evident already at early stages of a family design; however, some programmers familiarized

```
class Common
{
protected:
     Common() {}

      //  Family commonalities
};

class Member_A: protected Common
{
public:
     int  operation1(void);

private:
      //  Implementation declarations
};

class Member_B: protected Common
{
public:
     void operation2(int  i );

private:
      //  Implementation declarations
};
```

Figure 3.25: An example of dissociated family of abstractions implemented in C++.

with object-oriented programming may have difficulties to realize it. Therefore the implementation of the synchronization mechanism used as example in section 3.4 will be discussed next. That example was about two well-known synchronization mechanisms: *semaphores* and *condition variables*. These abstractions, though strongly connected by semantics, have no common operations and no conversion operator (as for the domain of operating systems, converting a semaphore into a condition variable, and vice-versa, is not defined). Forcing a common interface through polymorphism, in this case, may have negative consequences, beginning with a degenerated family organization.

```
class Common
{
protected:
      Common() {}

       //  Family commonalities
};

class Member_A: virtual protected Common
{
public:
      Member_A();
      int  operation1(void);

private:
       //  Implementation declarations
};

class Member_B: virtual protected Common
{
public:
      Member_B();
      void operation2(int  i );

private:
       //  Implementation declarations
};

class Member_AB: public Member_A, public Member_B {};
```

Figure 3.26: An example of combined family of abstractions implemented in C++.

Polymorphism degenerates the organization of the synchronizer family because it forces unreal relationships between its members. There are four possible organizations for a polymorphic synchronizer:

a) Member semaphore becomes the basic abstraction, with condition variable as its subtype;

b) The other way round, i.e. condition variable becomes the basic abstraction, and semaphore its subtype;

c) An extra abstract class gathering the operations of semaphore and condition variable builds a basic type for the family;

d) An extra empty abstract class serves as supertype for semaphore and condition variable.

Alternatives (a) and (b) clearly break the correspondence of abstractions with the problem domain. Alternative (c) is depicted in figure 3.27. In order to build a common interface, class `Synchronizer` has to declare all operations in the family. However, because there is not a single operation that is common to all members, it cannot declare pure virtual methods, having to resort to a protected constructor as well as to ordinary method declarations. This is probably the worst of the four solutions, since the operations declared in `Synchronizer` must also be defined and would be suitable to be invoked by applications. That is, an application would get no compiler error for invoking operation `broadcast` on a `Semaphore` if this was made trough a `Synchronizer *`. Alternative (d) does not add much more value than a `void *`. Therefore, none of these alternatives seems to be of any help for users.

```
class Synchronizer
{
protected:
      Synchronizer();

public:
      virtual ~Synchronizer();
      virtual  void p();
      virtual  void v();
      virtual  void wait();
      virtual  void signal();
      virtual  void broadcast();
};

class Semaphore: public Synchronizer
{
public:
      Semaphore(int value = 1);
      ~Semaphore();
      void p();
      void v();

private:
      //  Implementation declarations
};

class Condition: public Synchronizer
{
public:
      Condition();
      ~Condition();
      void wait();
      void signal();
      void broadcast();

private:
      //  Implementation declarations
};
```

Figure 3.27: A polymorphic family of synchronization abstractions implemented in C++.

3.8.2 Inflated Interfaces

The inflated interface of a family differs considerably from those of its member abstractions, for it is subject to the *partial* and *selective* realize relationships introduced in section 3.6. These relationships between inflated interfaces and family members are ultimately what enable an application-oriented operating system to be tailored to an application. Nevertheless, they add to the problematic of representing interfaces. The partial realize relationship to which these interfaces are subject allows a class to realize just a portion of an inflated interface, excluding the representation of interfaces by means of constructs that enforce integral compliance. At the same time, the selective realize relationship calls for a binding mechanism that can be controlled externally and that assures a single realization is bound to an interface at a time.

These conditions lead inflated interfaces to be represented in C++ by means of ordinary class declarations similar to the one illustrated in figure 3.28. The interface of this incremental family, represented by class `Family`, gathers all operations and constructors defined throughout the family. An application (last program fragment in the figure) could be written in terms of `Family`, thus postponing the choice of a family member, or delegating it to a tool. In this simple example, `Member_B` would be the choice, since it is the unique member that implements `operation2`.

When the inflated interface is unbound, class `Family` is exported from the `Interface` name space to the name space where realizations are declared, allowing application to be compiled. When one realization is selected, `Family` is again hidden in name space `Interface`, and the selected realization is renamed `Family`. Both operations can be accomplished in C++ through the type definition mechanism (`typedef`). If considered convenient by the designer, the binding of inflated interfaces can be centralized in a single configuration table, which could look like this:

```
// Forward declarations
class Member_A;
class Member_B;
namespace Interface { class Family; }
// Bindings
#if FAMILY == MEMBER_A
typedef Member_A Family;
# elif FAMILY == MEMBER_B
typedef Member_B Family;
#else
typedef Interface :: Family Family;
# endif
```

This is certainly a very simple example, but it serves to demonstrate the flexibility delivered by the inflated interface mechanism. Application programs can be written in terms of inflated interfaces, which are later bound to specific realizations either manually by the

```
namespace Interface
{
    class Family     // Inflated  interface
    {
    public:
        Family();
        Family(int  i );

        int  operation1(void);
        void operation2(int  i );
    };
}

class Member_A: protected Common // Partial realization
{
public:
    int  operation1(void);
     // ...
};

class Member_B: public Member_A // Full  realization
{
public:
    Member_B(int i = 0);
    void operation2(int  i );
     // ...
};

Family instance;
instance.operation1();
instance.operation2(1);
```

Figure 3.28: An example of inflated interface implemented in C++.

programmer or automatically by a tool the performs a syntactic analysis of the application so as to determine the best realization for each inflated interface. One such a tool has been implemented for EPOS, the prototype operating system developed to validate the ideas in this thesis. It will be described in section 4.6.

3.8.3 Scenario Aspects

Execution scenario aspects identified and isolated during the process of application-oriented system design can be implemented observing the same considerations stated about abstractions. Indeed, scenario aspects are usually also organized in families according to their commonalities. The inflated interface of these families is left unbound until the execution scenario for a certain application becomes evident. These aspects are then grouped by a *scenario* construct, and applied to abstractions through a *scenario adapter*.

The implementation of a family of scenario aspects is illustrated in figure 3.29. When the scenario aspects in this family are applied to an abstraction, they confer its operations atomicity. This is accomplished by having the scenario adapter to invoke `lock` and `unlock` operations so that the abstraction is trapped in a kind of monitor [Hoa74]. The `static` versions of `lock` and `unlock` are used to provide atomicity for class operations such as constructors[6], destructors, and some built-in operators. The `Coarse_Atomic` aspect locks all abstractions with a single mutex, while `Fine_Atomic` utilizes an individual mutex for each abstraction.

Before being applied to abstractions, scenario aspects are grouped in a *scenario*, which constitutes the first construct defined in the scope of an application-oriented component framework. While abstractions and execution scenario aspects are designed and implemented to be fully reusable independently of a specific framework, scenarios and scenario adapters are specific to frameworks. In general, the scenario construct incorporates selected scenario aspects via aggregation (which can also be implemented in C++ using single inheritance). If scenario aspects are smoothly applied to all abstractions, their inclusion in the scenario can be controlled using simple conditional compilation techniques.

Nevertheless, generic programming techniques can be deployed to enable aspects to be individually applied to abstractions. For instance the *traits* concept used in the C++ standard library [Str97]. Traits are parameterized classes whose static constant members describe the properties (the traits) of a certain type. An example is shown in figure 3.30. The parameterized class `Scenario_Traits` denotes the properties of abstractions with regard to execution scenarios. It can represent a pattern of properties that have to be supplied for each abstraction (private access control, `Member_A` in the figure), or it can itself represent default values for properties (public access control, `Member_B` in figure). The latter applies to any known type. If traits are used, the scenario construct should also be implemented as a parameterized class that relies on the `Scenario_Traits` to decide which aspects have to be applied to each abstraction.

Besides incorporating scenario aspects, a scenario construct usually also implements operations to establish the pre and post conditions required by the scenario for each interaction between a client and a scenario-independent abstraction. These operations have to

[6]Providing atomicity for object creation requires `static_lock` to be called from the scenario adapter `operator new` and `static_unlock` from the scenario adapter constructor.

```
namespace Interface
{
    class Atomic
    {
    public:
        void lock ();
        void unlock();
        static void static_lock ();
        static void static_unlock ();
    };
}

class Atomic_Common
{
protected:
    Atomic_Common() {}

public:
    static void static_lock () { static_mutex.lock (); }
    static void static_unlock () { static_mutex.unlock (); }

private:
    static Mutex static_mutex;
};

class Coarse_Atomic: protected Atomic_Common
{
public:
    void lock () { Atomic_Common::static_lock(); }
    void unlock () { Atomic_Common::static_unlock(); }
    static void static_lock () { Atomic_Common::static_lock(); }
    static void static_unlock () { Atomic_Common::static_unlock(); }
};

class Fine_Atomic: protected Atomic_Common
{
public:
    void lock () { mutex.lock (); }
    void unlock () { mutex.unlock (); }
    static void static_lock () { Atomic_Common::static_lock(); }
    static void static_unlock () { Atomic_Common::static_unlock(); }

private:
    Mutex mutex;
};
```

Figure 3.29: A C++ example of scenario aspect.

```
template<class T>
class Scenario_Traits
{
// public : or private :
        // ...
       static const bool is_atomic = false;
       static const bool is_shared = false;
       static const bool is_protected = false;
        // ...
};

template<>
class Scenario_Traits<Member_A>
{
public:
        // ...
       static const bool is_atomic = true;
       static const bool is_shared = false;
       static const bool is_protected = false;
        // ...
};

template<>
class Scenario_Traits<Member_B>:
       public Scenario_Traits<void>
{
public:
       static const bool is_atomic = true;
};
```

Figure 3.30: Traits of abstractions with regard to scenario aspects.

consider arbitrary combinations of scenario aspects, enforcing specializations when conflicts arise. A definitive characterization of a scenario, however, can only be formulated in the context of a specific framework.

Scenario adapters, which ultimately apply scenario aspects to abstractions, are also highly depended from the framework in which they are defined. Nevertheless, adapting an abstraction almost always implies in "wrapping" its operations so that they are executed enclosed within a pair of primitives with *enter* and *leave* scenario semantics. Several mechanisms to wrap objects and operations have been proposed for the C++ programming language. The direct ancestor of C++, C WITH CLASSES [Str94], allowed two special functions, namely `call` and `return`, to be defined for a class. These functions were implicitly called respectively before and after member functions of that class.

Tiemann [Tie88] proposed a language extension called "wrappers" that can be used to wrap the invocations of member functions of a class. Such a wrapper is identified by

```
class A_Class
{
public:
    int ()A_Class(int(A_Class::∗ function)(int ),  int  i ) {
        enter ();
        int  ret  = ( this−>∗function)(i );
        leave ();
        return ret ;
    }

    int  operation(int );
}

A_Class∗ p = new A_Class;
p−>operation(1);   //  p−>()A_Class(&A_Class::operation, 1);
```

Figure 3.31: Tiemann's proposal to wrap member functions in C++.

a pair of parentheses in front of the name of the class whose functions are to be wrapped (figure 3.31). A particularly interesting property of this mechanism is that it has access to arguments and to the return value of the function being wrapped, being useful to implement *Remote Procedure Call* mechanisms [BN84].

Unfortunately, none of these proposals has been incorporated by the standardized version of the language (ISO 14882). Nonetheless, some programming "tricks" can be used to overcome this language deficiency. Stroustrup [Str00] summarizes them in a parameterized class that wraps the methods of its argument class. This mechanism is depicted in figure 3.32. The `enter` primitive is called by the overloaded `operator->`, which returns a temporary object of type `Call_Proxy<T>` containing the pointer to the object on which the operation will be invoked. The `leave` primitive is called when this temporary object is destroyed. Actually no extra objects are created at all, since this metaprogrammed mechanism is completely resolved by the compiler into something like `(enter(); operation(); leave();)`. Another advantage of this mechanism is that it is not intrusive, i.e. it does not require modifications on the classes to which it applies. Its main disadvantage is that `enter` and `leave` do not have access to arguments supplied to the wrapped function.

```
template <class T>
class Call_Proxy
{
public:
    Call_Proxy(T* pp) : p(pp) {}
    ~Call_Proxy() { leave (); }
    T* operator->() { return p; }

private:
    T* p;
};

template <class T>
class Wrap
{
public:
    Wrap(T* pp) : p(pp) {}
    Call_Proxy<T> operator->()
        { enter (); return Call_Proxy<T>(p); }

private:
    T* p;
};

class A_Class
{
public:
    int operation1();
    int operation2(int);
};

Wrap<A_Class> obj(new A_Class);
obj->operation1();
obj->operation2(2);
```

Figure 3.32: Stroustrup’s proposal to wrap member functions in C++.

```
template<class OBJ, class RET> // no arguments
RET invoke(OBJ* obj, RET(OBJ:: * func)())
{ enter (); RET r = (obj->*func)(); leave (); return r ; }

template<class OBJ> // no arguments, no return
void invoke(OBJ* obj, void (OBJ:: * func )())
{ enter (); ( obj->*func)(); leave (); }

template<class OBJ, class RET, class ARG> // one argument
RET invoke(OBJ* obj, RET(OBJ:: * func)(ARG), ARG arg)
{ enter (); RET r = (obj->*func)(arg); leave (); return r ; }

template<class OBJ, class ARG> // one argument, no return
void invoke(OBJ* obj, void(OBJ:: * func)(ARG), ARG arg)
{ enter (); ( obj->*func)(arg); leave (); }

class A_Class
{
public:
    int operation1();
    int operation2(int i );
};

A_Class* obj = new A_Class;
invoke(obj, &A_Class::operation1);
invoke(obj, &A_Class::operation2, 1);
```

Figure 3.33: An alternative to wrap member functions in C++.

A solution to the member function wrapping problem that allows arguments to be accessed like in the Tiemann's proposal can be devised using function templates. The program fragments in figure 3.33 illustrate how. The first version of function template `invoke` takes two parameters: the type of the object whose functions are being wrapped, and the return type of the function being invoked. It then "sandwiches" the function between `enter` and `leave`, taking care of the return value. Function template `invoke` is specialized for functions that do not return any value (`void function()`) in the second fragment, and is successively overloaded to wrap functions that take a growing number of arguments (only the first overloading is shown in figure). In order to wrap an operation, one has to invoke it using the `invoke` function template, which has plain access to its arguments and return value. The whole mechanism can be made transparent by a wrapper class, though with considerable effort, since every operation defined for the wrapped class has to be represented in the wrapper.

3.9 Summary

Historically, applications have been adapted to the operating system, adhering to standardized application program interfaces that covey uncountable useless services (for each individual application), and yet fail to deliver much of what is necessary. An application-oriented operating system ruptures with this notion, implementing services that emanate from application requirements and delivering them as a set of configurable components that can be assembled to produce application-tailored system instances.

Such application-oriented operating systems can be constructed deploying the application-oriented system design method to engineer the envisioned domain. In this way, the domain is decomposed in abstractions that model application-specific perspectives of each entity. These abstractions are gathered in families according to what they have in common, with variability analysis being deployed to identify subsequent members of each family. During this process, variations that belong to the essence of abstractions are separated from those that emanate from execution scenarios, the former shaping family members, and the latter yielding scenario aspects. This separation improves on reusability, for scenario-independent abstractions can be reused in a larger variety of scenarios.

Maintainability in a component-based system is mainly a function of the number of components and of the complexity of each component. The separation of abstractions and scenario aspects improves both factors. Scenario-independent abstractions are less complex than their scenario-dependent counterparts, for they do not need to deal with environmental particularities. At the same time, the number of modeled software artifacts is reduced, since most scenario aspects apply to several abstractions (some even apply to all abstractions). Furthermore, not all of the variability observed in a family of abstractions yields family members. Some designate optional features that concern to several members at once. Instead of specializing each abstraction to produce versions that include such features, application-oriented domain decomposition suggests generic implementations of these configurable features to be modeled as constructs that can be reused by existing family members when the corresponding feature is required. An explosion of abstraction specializations is so avoided.

Still during domain decomposition, ad-hoc relationships between families of abstractions are exploited to model reusable software architectures. Such architectural specifications designate valid combinations of abstractions and scenario aspects that are subsequently materialized as component frameworks. In this way, application-oriented system design advances in one of the most contentious aspects of component-based software engineering: how to tell valid composites form invalid ones.

Abstractions modeled during domain decomposition originate the software components of an application-oriented operating system on 1-to-1 relation. They are delivered to users as abstract data types, with interfaces that clearly identify their responsibilities. Scenario aspects are maintained separately, being combined at user’s wish to shape the execution scenario for a certain composite. They are applied to abstractions by means of scenario adapters, which act as agents to mediate the interaction of scenario-dependent clients with scenario-independent abstractions.

In order to enable applications to deal with families of abstractions as they were single entities, an inflated interface is delivered for each family in an application-oriented system design. Such an interface exports a family as though a "super" component was available that implements all responsibilities assigned to the family. The choice of specific family members can thus be postponed or even delegate to a configuration tool. Such a tool would analyze the applications to determine which subsets of each inflated interface have been used, binding them to the most appropriate realization available.

The configurable system architectures modeled during domain analysis are delivered to users as component frameworks defined in terms of scenario adapters. Each scenario adapter constitutes a placeholder for an abstraction, pre-establishing relevant relationships. Abstractions are plugged to the framework via the inflated interface binging mechanism, which is also used to select scenario aspects.

An application-oriented operating system designed according to the directives of application-oriented system design can be implemented using a variety of techniques. As of today, it is probable that the C++ will the choice for most implementations, because it is one of the few programming languages that completely expose the underlying hardware, an essential condition for an operating system implementation. Furthermore, C++ supports multiple programming paradigms, including static metaprogramming, that can be combined to achieve efficient implementations.

Chapter 4

The EPOS System

This chapter describes EPOS, the experimental operating system developed in the scope of this dissertation to validate the concepts and techniques introduced in chapter 3. After an introduction of historical facts and fundamentals, the application-oriented system design of EPOS is presented, including the families of system abstractions, scenario aspects, and system architectures that result from the decomposition of the high-performance dedicated computing domain. Subsequently a strategy to automatically configure the operating system according to the needs of particular applications is presented.

4.1 A Bit of History

The EPOS system was born in 1997 at the Research Institute for Computer Architecture and Software Engineering (FIRST) of the German National Research Center for Information Technology (GMD) as a project to experiment with the concepts and mechanisms of application-oriented system design. Indeed, EPOS and application-oriented system design cannot be disassociated, since both have been evolving together from the very beginning as the research substratum for this dissertation.

The acronym EPOS[1] stands for *Embedded Parallel Operating System*. It was coined considering the main research goal established for the early system: to embed the operating system in a parallel application. The strategy followed to achieve this goal consisted in modeling the corresponding domain entities as a set of reusable and adaptable components, and developing a mechanism to allow parallel applications to easily specify

[1]The German word "epos" means "epic" in English, and, although suggesting that EPOS is an epic—about an academic operating system (a Portuguese sailor) that had to traverse a sea of difficulties (of standardized application program interfaces), passing by Taprobana (the common-sense defined by UNIX and Microsoft WINDOWS), to reach India (modern software engineering)—would be too pretentious, the metaphors about the epic "Os Lusiadas" [dC72] are left for the delight of readers (application programmers).

constraints to guide the arrangement of these components in a functioning system. As the system evolved, it became clear that this strategy could be applied to a broader universe of applications. Concerning design and organization, EPOS is inherently tied with dedicated computing and static configurability, but whether a platform is dedicated to an application temporarily (like in traditional parallel systems) or permanently (like in most embedded systems) does not play a significant role. Hence, *Embedded Parallel Operating System* can also be interpreted as a system that targets both embedded and parallel applications.

EPOS owes much of its philosophy to the PEACE system [SP94a], from which it inherited the notion that "generic" and "optimal" are adjectives that cannot be simultaneously applied to the same operating system, besides a rich perception of family-based design. EPOS implementation for the Intel iX86 architecture reuses some of the strategies adopted in ABOELHA [FAPS96], a former research operating system developed by the author. However, the design of ABOELHA did not promote reuse, so these strategies had to be completely remodeled for EPOS.

4.2 Fundamentals

EPOS was designed following the guidelines of *application-oriented system design* described in chapter 3. Indeed, EPOS has been created to experiment with application-oriented system design, which in turn has been evolving to contemplate design issues arisen by EPOS. Consequently, the design of EPOS is intrinsically application-oriented.

The domain envisioned by EPOS is that of high-performance dedicated computing, which comprises applications that, besides running with exclusivity on the respective platforms, require an efficient management of resources. It is important to notice that both adjectives, dedicated and high-performance, can be interpreted subjectively. As explained earlier in chapter 1, a dedicated system does not need to be permanently dedicated, it can also be scheduled to serve diverse applications, each of which gaining absolute control over the platform for a certain time. Similarly, a high-performance system does not need to be always associated to billions of floating point operations. In what concerns the operating system, a small, embedded application that demands the absolute majority of the resources available in the platform to accomplish its duties calls for a high-performance operating system just like a parallel application does.

Aiming at supporting applications in the high-performance dedicated computing domain, EPOS was designed pursuing the following goals:

- Functionality: EPOS shall deliver the functionality necessary to support the execution of high-performance dedicated applications.
- Customizability: EPOS shall be highly customizable, so that system instances can be tailored to specific applications; whenever possible, system tailoring shall succeed automatically.

- Efficiency: EPOS shall make resources available to applications with the lowest possible overhead, besides being itself thrifty.

The domain of high-performance dedicated computing aimed by EPOS is under constant evolution, steadily assimilating technological innovations. As a product of domain engineering, EPOS had to consider the dynamics of the envisioned domain, setting up an open and continuous domain analysis process that allows new entities to be included in the design as they are incorporated by the domain. Families of abstractions were thoroughly modeled, leaving room for upcoming members. Even the hypothesis of completely new families being added to the system has been taken in consideration.

The extreme scalability implied by these goals could only be achieved with a meticulous *separation of concerns*. As suggested by application-oriented system design, abstractions were modeled independently of each other, of execution scenario aspects, and of component frameworks. Consequently, EPOS abstractions can be extensively reused in a variety of scenarios. Furthermore, the framework can be adjusted to accommodate forthcoming abstractions, or to build particular software architectures, without affecting existing abstractions. In this way, EPOS could develop into an application-oriented operating system.

For the forthcoming discussion of EPOS design, however, it is important to bear in mind that the ultimate goal of EPOS is the validation of application-oriented system design concepts and techniques. Therefore, some complex domain entities were specified only to the extent of corroborating application-oriented system design. Such entities would have to be further refined in order to sustain an actual implementation.

4.3 System Abstractions

EPOS families of abstractions result from the application-oriented decomposition of the high-performance dedicated computing domain. Many of the entities in this domain that concern the operating system are conventions defined by computer scientists and system developers. Therefore, EPOS resorted to traditional operating system books (e.g. Tanenbaum [Tan92] and Silberschatz [SGP98]) as a reference to the operating system domain vocabulary. This decision helped to make EPOS user-friendlier, since its abstractions have been named after the classic concepts they represent, giving many of them a self-explanatory character. Systems that assign ethereal names to abstractions impose extra difficulties to users, which first have to discover the name of abstractions they want to use.

Yet, many operating system abstractions stem from physical devices. These abstractions are conventionally modeled considering the role physical devices play in the operating system. However, as an application-oriented operating system, EPOS gives priority to the role they play in application programs. For example, the *timer* physical resource is delivered as an abstraction capable of generating alarm events for applications, though it is internally reused as the operating system time-keeping device.

As proposed by application-oriented system design, EPOS abstractions have been modeled independently of execution scenario aspects and specific system architectures. Consequently, EPOS abstractions could reach a degree of reusability that allows, for instance, the same abstraction of the `Thread` family to be deployed in a single- or multi-tasking environment, as part of a μ-kernel or completely embedded in an application. The protection barrier the eventually separates applications from each other and from the operating system is not modeled in the context of abstractions, but as an architectural aspect of EPOS framework. Furthermore, the separation of scenario aspects from abstractions considerably reduced the number of components in EPOS repository, whereas numerous scenario-specific specializations could be suppressed.

Figure 4.1 shows a top-level representation of EPOS families of abstractions. Families were organized in six large groups: *memory management*, *process management*, *process coordination*, *inter-process communication*, *time management*, and *I/O management*. Each one of these groups will be subsequently described in this section, while applicable scenario aspects will be described in the next section. Exception is made to private or specialized scenario aspects, which are described in the scope of the family (or abstraction) to which they regard. Non-portable abstractions, such as *node* and *processor*, were modeled in EPOS as *hardware mediators*. These abstractions will be separately described in section 4.5.2.2.

The inflated interface of each family of abstractions will not be explicitly presented, since this would bring little contribution to the reasoning on EPOS design in comparison to explosion of details it would trigger. Nevertheless, the category of each family of abstractions—whether uniform, incremental, combined, or dissociated—will be identified, thus revealing how their inflated interfaces have been specified.

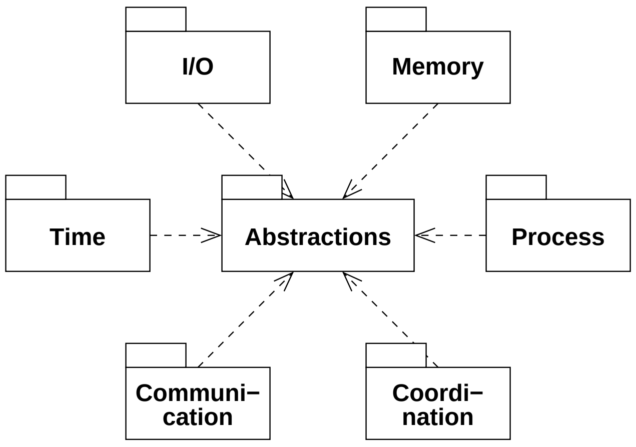

Figure 4.1: Groups of families of abstraction in EPOS.

4.3.1 Memory Management

Memory management in EPOS is accomplished by the families of abstractions shown in figure 4.2. The main memory available in the system is delivered to applications through the concept of memory segment realized by the `Segment` family. In order to be manipulated, a segment must first be attached to the address space of a process, which is realized by the `Address_Space` family. Besides supporting this logical concept, the family of address spaces is the one that ultimately implements a memory management policy for the system, since it controls the allocation and mapping of the physical memory that effectively makes up a memory segment.

In the context of application-oriented operating systems, *virtual memory* is not an autonomous domain entity, but a workaround to enable the execution of applications with memory requirements that exceed the available physical memory. That is, applications do not directly demand virtual memory, they demand memory. However, currently available virtual memory mechanisms are mostly inadequate to support applications in the domain of high-performance dedicated computing, firstly because many dedicated systems do not count on a secondary storage to extend main memory, but also because such mechanisms do not sustain high-performance. Therefore, no virtual memory strategy has been modeled for EPOS at this time.

4.3.1.1 Memory Segments

A memory segment is nothing but a chunk of main memory that can be used by applications to store arbitrary code and data. The physical memory corresponding to a memory segment is managed by the family of address spaces, thus keeping the `Segment` family independent of a memory management policy. Figure 4.3 shows EPOS family of memory segments, which was designed in an incremental fashion. The common aspects of the family are encapsulated in a package that is reused by member `Static`, which is in turn extended by member `Dynamic`. The former is responsible for memory segments that cannot be resized nor can have their properties modified after having been attached, while the latter allows for these operations.

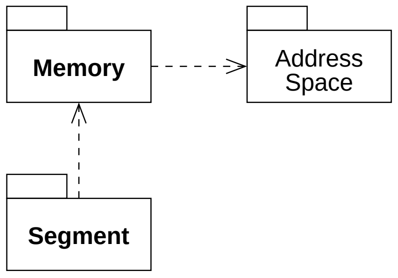

Figure 4.2: Families of abstractions concerning memory management in EPOS.

These abstractions could only be modeled as an incremental family because the effective allocation and mapping of physical memory to build a segment is carried out by the family of address spaces. Otherwise, differences in allocation policy would have made this design impractical. Furthermore, the family was reduced to only two members because sharing and protection were model as scenario aspects[2]. The `Shared` scenario aspect profits from the memory mapping capabilities of the `Address_Space` family to support the sharing of segments among processes, while the `Protected` aspect uses the mapping modes provided by that family to protect segments against detrimental access.

Both abstractions in this family are mutually exclusive, since their dependencies on the family of address spaces, itself mutually exclusive, cannot be accommodated simultaneously. In order to allow resizing, member `Dynamic` requires a non-contiguous mapping of physical memory, which is only provided by the `Paged` member of the `Address_Space` family. Member `Static` has no requirements in this regard and is able to operate properly with any kind of address space.

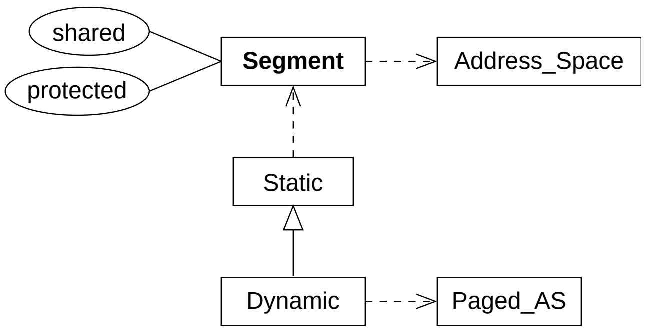

Figure 4.3: EPOS family of memory segments.

4.3.1.2 Address Spaces

The address space of a process corresponds to the range of memory addresses it can access. In order to support multitasking, most hardware architectures include a *Memory Management Unit* (MMU) that enables the separation of logical and physical address spaces. A process is thus said to have a logical address space, which is mapped into the physical memory address space. The MMU supports a non-contiguous mapping between these address spaces, enabling a rational use of memory by multiple processes. Nevertheless, many dedicated applications are carried out by a single process (on each node) and will not benefit from this mechanism. Moreover, some dedicated applications are executed on platforms that do not feature a MMU. Therefore, an address space abstraction for the dedicated computing domain must consider both perspectives: with and without logical address mapping.

[2]The `Shared` and `Protected` scenario aspects modeled here are specializations of global scenario aspects described in section 4.4.

The concept of address space is realized in EPOS by the dissociated family of mutually exclusive abstractions shown in figure 4.4. These abstractions are not intended to be directly used by application programmers. They are implicitly invoked as a result of the memory and process management policies in force. The `Flat` member is used when a single process (possibly multithreaded) executes alone in the system. It implements a policy that pre-allocates all the memory available in the system to this single process in a contiguous way, dispensing with a MMU and allowing for an efficient application-level memory allocator (`malloc`). This allocator would still have to keep track of the stretches of memory currently being used, but would no longer need to perform physical memory allocation and mapping. The contiguous property of the `Flat` address space also brings an important advantage to processes that perform user-level I/O: since logical and physical address spaces match, *Direct Memory Access* (DMA) transactions can be issued with logical addresses, thus eliminating translations and intermediate copies.

When the policy defined by the `Flat` abstraction is in force, the family of memory segments becomes an informative connotation, since there is no practical reason for a process to create a memory segment if it already possesses the whole memory and there are no other processes with which to share that segment.

The `Paged` member of the `Address_Space` family supports the mapping of logical address spaces to main memory through *paging* [KHPS61, HK61]. This strategy allows multiple processes to coexist in a rational way, since memory is allocated and mapped to their address spaces on demand and is reclaimed when they terminate.

At least two other alternatives are there to handle memory management in multitasking scenarios: *segmentation* [Den65] and *paged segmentation* [Org72]. However, the hardware mechanisms necessary to support them are mostly unavailable in contemporary computers[3], so they were not specified in EPOS.

A set of memory mapping modes (e.g. read-only, cached, etc) has been modeled as configurable features for the `Address_Space` family. These modes have defaults

[3]The Intel ix86 architecture [Int95a] uses segmentation or segmentation plus paging, but it can be configured to emulate pure paging.

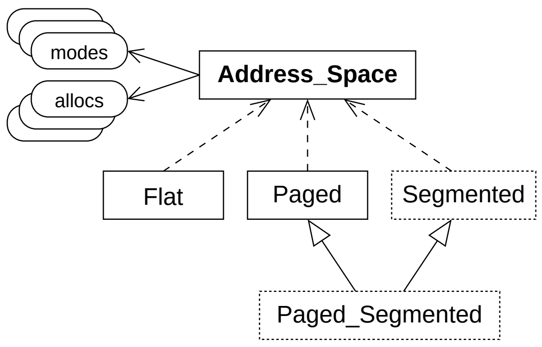

Figure 4.4: EPOS family of address spaces.

that apply to the whole address space, but can usually be dynamically overridden for each memory segment. A set of allocation algorithms has been modeled as a mutually exclusive configurable feature for the family.

4.3.2 Process Management

Process management in EPOS is delegated to the families of abstractions shown in figure 4.5. The concept of *process* is delivered to applications by two abstractions: `Task` and `Thread`. If a process is thought of as a program in execution, then a task corresponds to the activities specified in the program, while threads are the entities that perform such activities. This separation of concerns enables a task to be simultaneously executed by multiple threads. Threads are locally scheduled for execution by a `CPU_Scheduler`. Global schedulers to promote load balancing for parallel applications were not yet specified for EPOS. They shall inaugurate a new family of abstractions in the future.

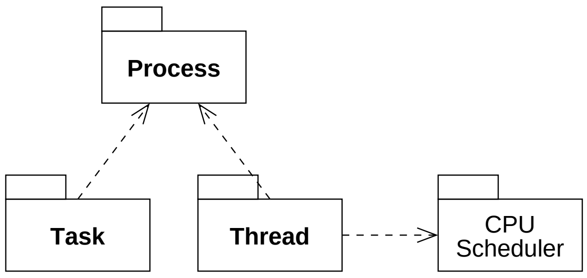

Figure 4.5: Families of abstractions concerning process management in EPOS.

4.3.2.1 Tasks

A *task* comprises the code, global data, and resources of a process, thus, being passively shared by its threads. A task constitutes the unit of distribution of a parallel application. Tasks are realized in EPOS by the `Task` family of abstractions depicted in figure 4.6, which was modeled as a dissociated family with two mutually exclusive members, namely `Exclusive` and `Mutual`. Abstractions in this family were modeled to be deployed independently of a particular thread abstraction, so both members can be used with a single or with multiple threads. Two instances of the `Segment` abstraction are allocated to hold respectively the code and the data associated with a `Task`.

The `Exclusive` member of the `Task` family was conceived to support a single process that has absolute control over the resources available in the platform. Hence, it pairs up with the `Flat` member of the `Address_Space` family, which implements a single contiguous address space. An exclusive task is set up in such a way that, when the corresponding process begins execution, all needed physical resources, including memory, are already allocated to it. This is one of the factors that enable EPOS to assume the embedded-into-the-application architecture, in which all system services are implemented at user-level (see section 4.5.3.2 for more details about this architecture).

The `Mutual` member of the `Task` family pairs up with the `Paged` member of the `Address_Space` family to accomplish a multitasking environment in which tasks share available resources that are allocated on demand. The scenario aspects that concern to this family (e.g. identification, location, protection, etc) have been modeled as global aspects, for they also apply to other abstractions. They will be described in section 4.4.

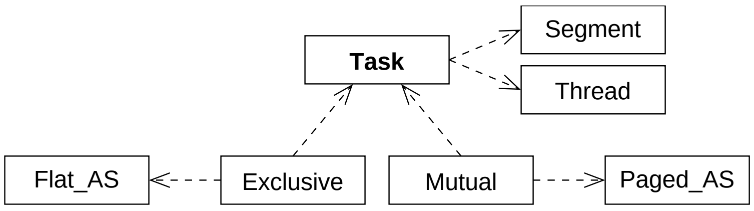

Figure 4.6: EPOS family of tasks.

4.3.2.2 Threads

Threads "execute a task", thus they correspond to the active part of a process. The resources of a task, including its data segment, are shared by all of its threads, but each thread has its own local data segment (*stack*) and execution context. A thread constitutes the unit of execution of an application. Threads are realized in EPOS by the `Thread` family of abstractions depicted in figure 4.7. This family was modeled in an incremental fashion, with commonalities being captured in a package that is reused by member `Exclusive` and inherited by other family members.

The `Exclusive` member of the `Thread` family was conceived to support a single-

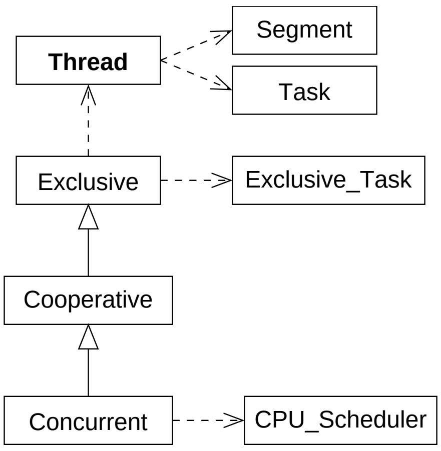

Figure 4.7: EPOS family of threads.

threaded process that executes alone in the system. It depends on the `Exclusive` member of the `Task` family, which ultimately shapes the single-tasking environment in which exclusive threads perform. Though primitive[4], this process model is sufficient to support a reasonable number of dedicated applications, and it has expressive advantages over other models in what regards performance. Since all resources available in the system are pre-allocated to this unique process during system initialization, and also because no scheduling is necessary, this process model can be achieved without any run-time overhead, i.e., all physical resources, including CPU time, are entirely available to the application process.

Support for multithreading is the subsequent increment to the `Thread` family. It is realized by the `Cooperative` family member, which covers the mechanisms necessary for multiple threads to coexist in the system. This family member was conceived to be independent of the task model in force, so no differentiation is made between threads that execute the same task and threads of single-threaded tasks. As stated before, the unit of execution in EPOS is the thread, not the task. The `Cooperative` thread abstraction addresses the issues relative to multiple threads that share the processor in a collaborative way, dispensing with processor scheduling. In this scenario, a thread voluntarily relinquishes the processor in favor of another one; hence, one could say that cooperative threads are self-scheduling.

The `Concurrent` abstraction extends the `Thread` family so that threads are executed concurrently. This family member relies on a family of processor schedulers, each implementing a different scheduling policy, in order to multiplex the processor among threads. According to the scheduling policy in force, the scheduler can be invoked regularly to proof whether the running thread still fulfills the necessary conditions to seize the processor, otherwise preempting it in favor of another thread. Being an extension of cooperative threads, concurrent threads also have the possibility to voluntarily relinquish the processor, either in benefit of another thread, or causing the scheduler to be invoked.

4.3.2.3 Processor Schedulers

Processor scheduling is deployed in order to multiplex the processor for concurrent thread execution. Therefore, EPOS only features a processor scheduler if the `Concurrent` member of the `Thread` family is in use. In this case, the scheduler is realized by the entities shown in figure 4.8. The `CPU_Scheduler` abstraction realizes the mechanisms that are necessary to suspend and resume the execution of threads (i.e. context switching), invoking `Policy` to select a thread to occupy the processor whenever it becomes idle. That is, EPOS does not actually feature a family of schedulers, but a family of scheduling policies that are enforced by a single `CPU_Scheduler` abstraction.

The isolation of scheduling policies was achieved by modeling `Policy` as a polymor-

[4]Only three operations are valid on objects of type `Exclusive_Thread`: self-referencing, status reporting, and termination.

phic uniform family of abstractions. In this way, `CPU_Scheduler` can invoke `Policy` without having to consider which policy is currently in force, and policies can be changed at run-time. In order to configure the scheduler, one or more policies are selected at generation-time to be deployed at run-time. Since the processor scheduler is an internal entity, invisible to applications, the selection of policies at run-time is performed through operations provided by the `Concurrent` thread abstraction.

If one of the scheduling policies selected for a given system configuration is preemptive, the `preemption` configurable feature is enabled, causing the scheduler to be periodically activated by the `Alarm` abstraction [section 4.3.5] to check for policy compliance. A thread that voluntarily relinquishes the processor also causes the scheduler to be activated. If the leaving thread indicates a candidate to replace it on the processor, that thread temporarily assumes the scheduling characteristics of the leaving one. For example, if static priority is the policy in force, then the priority of the designated thread is raised to equal that of the designator until the next scheduling event. Otherwise, a thread can relinquish the processor leaving the decision of which thread shall substitute it to the policy enforcer.

A third situation in which the scheduler can be activated occurs when a concurrent thread goes waiting for an event, for example I/O completion. The traditional approach of generic multitasking systems to deal with this situation is to block the waiting thread and invoke the scheduler to select another thread to occupy the processor. In this scheme, blocked threads are said to be "idle waiting". Nevertheless, this solution is not so straightforward for dedicated systems, which may be better served if "busy waiting" is deployed. Several high-speed peripherals, including high-speed networks, may imply in busy waiting cycles that are shorter than the time needed to block the running thread and reschedule the processor. Therefore, busy and idle waiting have been modeled as configurable features that can be selected according to application needs.

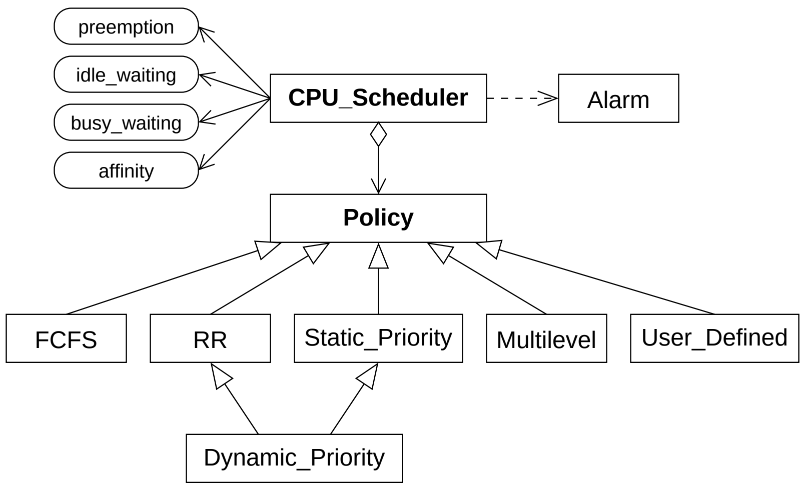

Figure 4.8: EPOS family of processor scheduling policies.

The CPU_Scheduler abstraction is also able to perform processor scheduling in a *Symmetric MultiProcessor* (SMP) environment [JS80, TG89]. Regardless of the number of processors, a single queue of threads ready to execute is maintained. When a processor becomes idle, CPU_Scheduler is invoked by that processor to select a new thread to execute. Race conditions originated from parallel invocations of the scheduler are prevented by activating one of the Locked members of the Atomic scenario aspect (see section 4.4.6)[5].

In multiprocessor environments, the configurable feature affinity can be enabled to request CPU_Scheduler to consider processor affinity [TTG93] while choosing a thread to occupy an idle processor, so that threads that were formerly executing on that processor are given preference. If the scheduler cannot find a thread under the affinity criterion, a thread formerly executed in another processor is selected. Processor affinity is useful with short-term scheduling to promote cache locality, since a thread returning to a processor it has recently seized may still find part of its working-set of memory on that processor's cache.

The FCFS member of the Policy family realizes the *first-come-first-served* scheduling policy. When concurrent threads operate under this policy, they acquire several of the characteristics of a cooperative thread, for a thread only releases the processor on its own free will. Nevertheless, if the idle waiting configurable feature is enabled, a thread going into a waiting state releases the processor for the next coming thread, retrieving it when it returns to the ready state. The RR member of the Policy family realizes the *round-robin* processor scheduling policy. With this policy, the processor is multiplexed on time among threads, so that each thread is given an identical fraction of processing time (*time-slice*), after which it is preempted and sent to the end of the ready queue.

The Static_Priority member realizes a processor scheduling policy based on statically assigned priorities [ZRS87]. When a thread is created, it is assigned a priority that is used for scheduling decisions: higher priority threads are executed first. Although applications are provided with means to redefine the priority of a thread afterwards, the scheduler itself never takes this initiative. This policy influences the handling of asynchronous events in EPOS [section 4.3.6.3], since an event only causes the processor to be preempted if the thread assigned to handle it has a higher priority than the one currently being executed.

The Dynamic_Priority member extends Static_Priority and RR to accomplish a policy that automatically adjusts the priority of threads to promote interactiveness. The strategy used to build this policy consists in allowing a thread to seize the processor for a certain time (usually greater than the time-slice defined for round-robin), after which it is preempted and its priority is recalculated. Priorities are recalculated as a function of the processing time effectively used, so threads that voluntarily relinquish the

[5]The dependency of CPU_Scheduler on Atomic is externally expressed as a composition rule [section 4.5.1.2]. It can be easily suppressed for eventual scheduling algorithms that are able to cope with non-blocking synchronization.

processor before their time-slice is over have their priority increased, while those that tend to monopolize the processor have their priority decreased. This is basically the scheduling policy adopted in the UNIX SYSTEM V operating system [Bac87]. It promotes interactiveness because interaction with users is achieved via I/O operations, causing interactive threads to release the processor prematurely and consequently raising their priority[6].

In principle, a single processor scheduling policy is in force at a time. However, the `Multilevel` member of the `Policy` family supports scheduling policies to be combined by gathering threads in groups and applying distinct inter- and intra-group policies. For example, threads could be gathered in three groups scheduled with static priorities (i.e. the inter-group policy). The group with the highest priority could again deploy static priorities to support a set of real-time threads; the intermediate priority group, consisting of interactive threads, could deploy round-robin; and the group with the lowest priority, comprised of batch threads, could be subject to FCFS. Although such a complex scheduling scenario is improbable for a dedicated system, some simpler combinations of scheduling policies may be useful.

`User_Defined`, the last member of the `Policy` family of abstractions, allows applications to specify their own processor scheduling policy. It is accomplished by a multilevel scheme that assigns the highest priority to an application thread that acts as the scheduler, while all other threads are assigned the same lower (than the scheduler thread) priority. The scheduler thread implements the desired policy and relinquishes the processor in benefit of the selected thread.

A preemptive user-level scheduler [ABLL92] can be accomplished in a variety of ways, the simplest one implies in defining a round-robin policy for the high priority group of threads, which comprises only the scheduler thread. This would grant that the scheduler thread regains control over the processor on a regular basis. It can then decide to reschedule the thread that originally occupied the processor, or select another one. Nevertheless, the `User_Defined` policy can considerably affect performance if used to perform short-term scheduling, since it doubles the scheduling overhead.

4.3.3 Process Coordination

The mechanisms available in EPOS to coordinate the parallel execution of processes are realized by the `Synchronizer` and `Communicator` families of abstractions shown in figure 4.9. However, the `Communicator` family delivers coordination as a side-effect of inter-process communication, and hence will be described later in section 4.3.4. Nevertheless, it is important to observe that every time two processes exchange a (possibly empty) message, they implicitly exchange status information that can be used for coordination purposes. For example, when a process sends a message to another, it signalizes the recipient process that it is ready with the computation needed to produce that message. The

[6]In order to achieve this effect in EPOS, the `idle_waiting` configurable feature must be enabled.

set of possibilities to indirectly coordinate processes through message exchange grows considerably if inter-process communication implies in a rendezvous between the communicating processes. In this case, sender and receiver are implicitly synchronized on each message exchange.

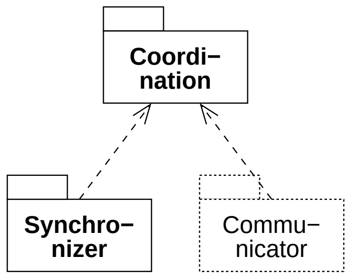

Figure 4.9: Families of abstractions concerning process coordination in EPOS.

4.3.3.1 Synchronizers

Synchronizers are used to avoid race conditions during the execution of parallel programs. A race condition occurs when a thread accesses a piece of data that is being modified by another thread, obtaining an intermediate value and potentially corrupting that piece of data. A synchronizer prevents such race conditions by trapping sensible data in a critical section, which is exclusively executed by a single thread at a time. EPOS dissociated family of synchronizers is depicted in figure 4.10.

The `Mutex` member of the `Synchronizer` family implements a simple mutual exclusion device that supplies two atomic operations: `lock` and `unlock`. Invoking `lock` on a mutex locks it, so subsequent invocations cause the calling threads to wait. When a thread invokes the operation `unlock` on a mutex and there are threads waiting on it, the first thread put to wait is allowed to continue execution, immediately locking the mutex. If no threads are waiting, the `unlock` operation has no effect; it is not accumulated to match forthcoming `lock` operations. The mutex mechanism is sometimes called a *binary*

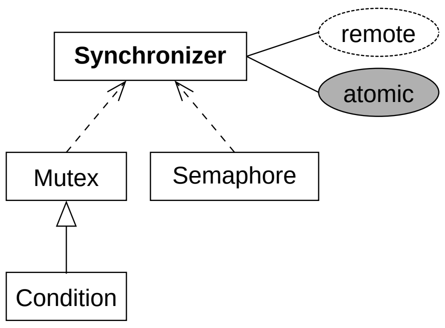

Figure 4.10: EPOS family of synchronizers.

semaphore.

The `Semaphore` member of the `Synchronizer` family realizes a *semaphore variable* [Dij65]. A semaphore variable is an integer variable whose value can only be manipulated indirectly through the atomic operations `p` and `v`. Operation `p` atomically decrements the value of a semaphore, and operating `v` atomically increments it. Invoking `p` on a semaphore whose value is less than or equal to zero causes the thread to wait until the value becomes positive again. A semaphore initialized with "1" acquires the initial semantics of a mutex, but it can be initialized with any other value to accomplish other synchronization semantics.

The `Condition` member of the `Synchronizer` family realizes a system abstraction inspired on the *condition variable* language concept [Hoa74], which allows a thread to wait for a predicate on shared data to become true. `Condition` protects the associated shared data in a critical section using the capabilities inherited from `Mutex`. In order to wait for the assertion of a predicate, a thread invokes operation `wait`, which implicitly unlocks the shared data and puts the thread to wait. Several threads can be waiting on the same condition. The assertion of a predicate can be announced in two ways: operation `signal` announces it to the first waiting thread, and operation `broadcast` announces it to all waiting threads. When a thread returns from the `wait` operation, it implicitly regains control over the critical section.

The global scenario aspect `remote`, which will be described later in section 4.4.7, provides a remote object invocation mechanism that shapes a distributed scenario for abstractions. When used in this scenario, abstractions of the `Synchronizer` family accomplish a centralized solution to coordinate processes of a parallel application. The solution consists in having one of the processes to create the synchronizer locally, while the remaining processes share it via the remote object invocation mechanism. However, this mechanism was not designed focusing distributed coordination and may cause a bottleneck in the parallel application[7]. Therefore, a specialization of the `remote` scenario aspect has been considered for the `Synchronizer` family to optimize this scenario. A fully distributed solution, however, can only be achieved with the introduction of a new family member.

A negative specialization (cancellation) was specified for the `atomic` global scenario aspect [section 4.4.6], since operations on synchronizers are inherently atomic.

4.3.4 Inter-Process Communication

Inter-process communication in EPOS is delegated to the families of abstractions shown in figure 4.11. Application processes communicate with each other using a `Communicator`, which acts as an interface to a communication `Channel` implemented over a

[7]The node in which the synchronizer resides becomes a kind of "coordinator", with which all other processes have to communicate.

`Network`. The messages sent through a `Communicator` can be specified as sequences of bytes of a known length, or they can be covered by an `Envelope`.

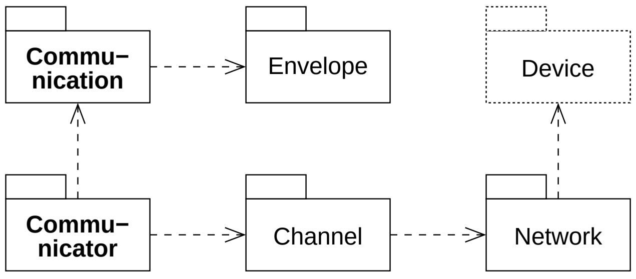

Figure 4.11: Families of abstractions concerning inter-process communication in EPOS.

4.3.4.1 Communicators

A *communicator* is an end-point for a communication channel that enables application processes to exchange data with each other. Therefore, when an application selects a communicator, it implicitly designates the kind of communication channel that will be used. Communicators, like most other system abstractions, are assigned to tasks, thus being shared by their threads. Communicators are realized in EPOS by the `Communicator` family of abstractions shown in figure 4.12, which was modeled as a dissociated family whose members can be simultaneously deployed.

The `Link` member of the `Communicator` family realizes an end-point for logical connections between processes that carry byte streams. The `Port` and `Mailbox` members realize end-points for a communication channel in which datagrams flow, but a port always belongs to a single task, while mailboxes can be shared among tasks.

The `ARM_Segment` (*Asynchronous Remote Memory Segment*) member of the `Communicator` family realizes an end-point for a communication mechanism that supports asynchronous access to a memory segment in a remote node. This mechanism is asyn-

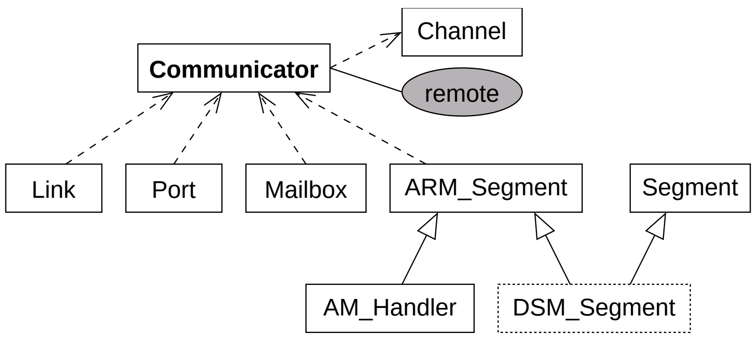

Figure 4.12: EPOS family of communicators.

chronous because processes manipulating an `ARM_Segment` are not implicitly synchronized and can corrupt the data in that segment. Data read from a remote segment becomes local and private to the reading process. If necessary, synchronization has to be achieved by other means (e.g. distributed semaphores). In order to use this communicator, a process specifies a memory segment on a remote node that has been previously exported by its owner. It can then invoke operations to read from and to write to this segment (asynchronous remote memory segments are not mapped into the address space of processes).

The `AM_Handler` (*Active Message Handler*) member of the `Communicator` family realizes an end-point for active messages [vECGS92]. The basic idea behind this concept is that a message, besides transporting data, also carries a reference to a handler that is invoked, in the context of the receiving process, to handle the message upon arrival. This kind of communication is sometimes called single-sided because the receive operation is not explicitly expressible. For this mechanism to work properly, means must be provided to the sending process so it can specify a handler that is valid in the context of the destination process. The most typical answer to this issue is to deploy active messages in an SPMD (*Single Program, Multiple Data*) environment, in which all processes have an equivalent address space. However, indirection mechanisms and the exchange of handler references using other communication mechanisms are also possible.

Active messages have been modeled in EPOS in such a way that communication is hidden behind a remote handler invocation, with messages being indirectly exchanged as arguments to the handler. When a process instantiates an `AM_Handler`, it supplies a reference to a handler on a remote process. Afterwards it can invoke the handler supplying arguments that are transparently marshaled in a message and delivered to the remote handler.

The `DSM_Segment` member of the `Communicator` family, which would realize a *Distributed Shared Memory* (DSM) mechanism for EPOS, could have been modeled as an extension of the `ARM_Segment` communicator and of a member of the `Segment` family of memory segments. Unlike an `ARM_Segment`, however, a `DSM_Segment` would be attached to the address space of processes, dispensing with explicit read and write operations. It would also implement a mechanism to grant data coherence. This communicator would enable application programmers to write parallel applications for distributed memory machines as if they were shared memory ones.

A negative specialization (cancellation) was specified for the `remote` global scenario aspect [section 4.4.7], since a process is not allowed to exchange messages using communicators created by other processes on remote nodes.

4.3.4.2 Channels

A communication *channel* is the entity effectively responsible for inter-process communication in EPOS. It uses network resources to build a logical communication channel through which messages are exchanged. A channel implements a communication protocol

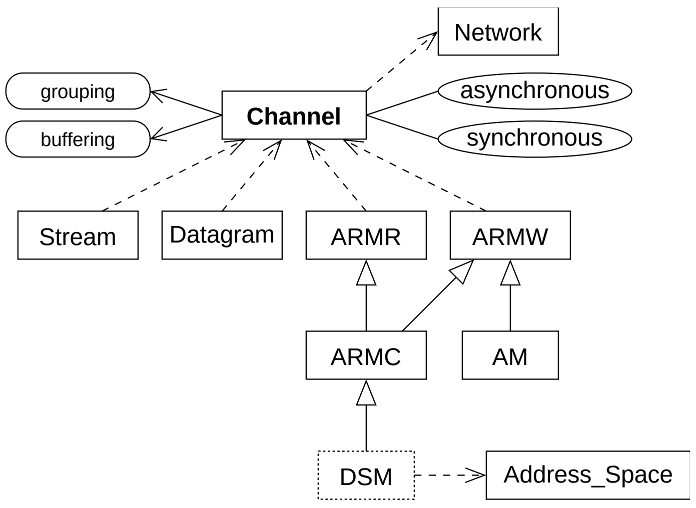

Figure 4.13: EPOS family of communication channels.

that, according to the *Basic Reference Model for Open Systems Interconnection* (ISO/OSI-RM) [ISO81], would be classified at level four (transport). EPOS family of communication channels is depicted in figure 4.13. It was modeled as a dissociated family, whose members are indirectly accessed through the corresponding members of the `Communicator` family.

A communication channel has an implicit capacity [Sha48]. Trying to insert a message into a saturated channel causes the transmitter to wait until the channel can accommodate the message. Likewise, the attempt the extract a message from an empty channel causes the receiver to wait until a message is available. Whether a thread waiting on a channel performs busy or idle waiting hinges on the related configurable feature from the `CPU_Scheduler` family [section 4.3.2.3]. Notwithstanding this, a channel can have its capacity extended by enabling the `buffering` configurable feature. In this case, messages sent through a saturate channel are accumulated for posterior handling.

Sometimes it is desirable to fork a channel, so that a transmitted message is simultaneously multicasted to several receivers, or broadcasted to all receivers. The collective operations used in many parallel applications could be considerably optimized in this way. EPOS allows a channel to be forked when the `grouping` configurable feature is enabled. In this case, special identifiers are supplied to designate a group of communicators as the recipient of a message. The effect of `buffering` and `grouping` configurable features on a channel is illustrated in figure 4.14.

The `synchronous` scenario aspect yields an execution scenario in which the operations used to inject a message into a channel only conclude when the message is extracted from the channel at the receiver's side. In a synchronous communication scenario, processes involved in a message exchange are said to make a “rendezvous”. Conversely, the `asynchronous` scenario aspect modifies these operations so they conclude as soon as

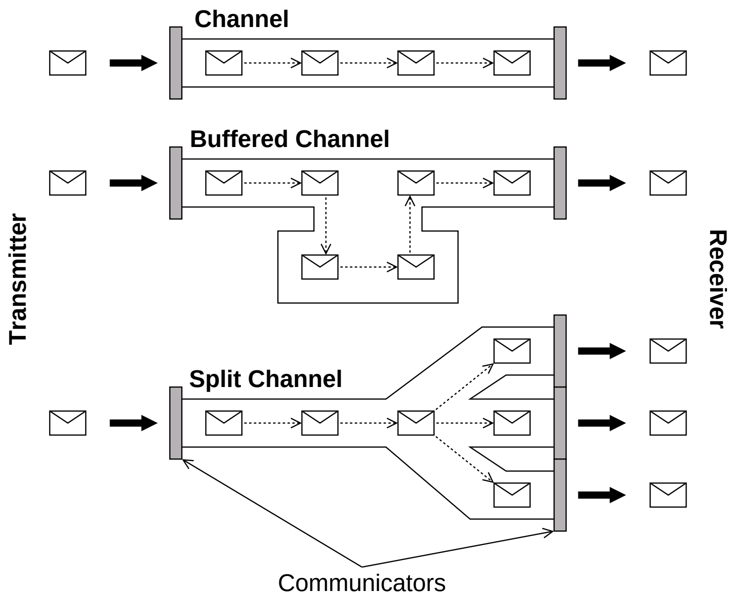

Figure 4.14: The effect of configurable features `buffering` and `grouping` on communication channels.

the delivery of a message is negotiated. If the `buffering` configurable feature is enabled, this is achieved by copying the message into a buffer and scheduling it for delivery. Otherwise, the sender is supposed not to modify the message until indicated to do so. An operation is provided that enables a process to check for this condition.

The `Stream` member of the `Channel` family realizes a connection-oriented channel that can be used to transfer streams of bytes. It pairs up with the `Link` communicator. The `Datagram` member realizes a channel that supports the transmission of *datagrams*. It has two possible end-points: `Port` and `Mailbox`. Three members concern asynchronous access to a remote memory segment: `ARMR`, `ARMW`, and `ARMC`. They realize communication channels respectively for reading, writing, and copying (reading and writing) from/to a remote memory segment and are delivered to applications through the `ARM_Segment` communicator.

The `AM` (*Active Message*) channel specializes `ARMW` to introduce the concept of a message handler that is automatically invoked when the message reaches its destination. It pairs up with the `AM_Handler` communicator. The communication channel used to support distributed shared memory would specialize the `ARMC` channel in order to map it to the address space of processes.

4.3.4.3 Networks

A communication channel is, at last, an abstraction of a network, in that networks provide the physical means to build logical channels. The idiosyncrasies of each network tech-

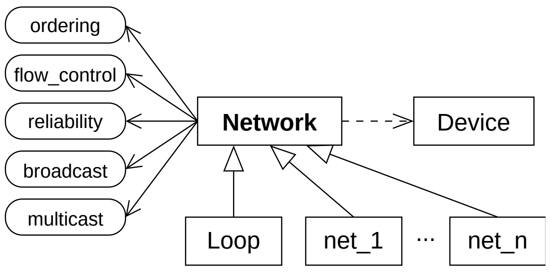

Figure 4.15: EPOS family of networks.

nology, however, could require the members of the `Channel` family to be specialized too often. This picture was prevented in EPOS by modeling networks as members of a uniform family of abstractions, so that all networks are equivalent from the standpoint of channels.

The uniform design of the `Network` family, which is outlined in figure 4.15, should not subdue special features delivered by a particular network, since abstractions in this family implement high-level transport services that are seldom implemented by the hardware. The virtual networks in this family can use special services provided by the network to optimize the implementation of such transport services. Some of these special features are used to implement the configurable features modeled for the family.

The `Network` family features a member for each network technology supported in the system (e.g. FAST ETHERNET and MYRINET). Each member encapsulates a physical network device that has been previously abstracted by the `Device` family [section 4.3.6.2]. The family also features a `Loop` device that is used to establish a communication channel between processes executing on the same node. In principle, abstractions in this family are used indirectly through a communicator, but they are also made available for the convenience of applications that need, for instance, to implement special communication protocols.

A set of configurable features, corresponding to operational modes, was modeled for the `Network` family. These features are interpreted as follows: `ordering` requires messages sent through a network to be delivered at the destination in the same order they were sent; `flow_control` requires a network abstraction to implement flow control; `reliability` requires a network to assure error-free delivery of messages; `broadcast` enables the interpretation of broadcast addresses, so messages can be broadcasted to all hosts in the local network; `multicast` enables the interpretation of multicast addresses, causing a message to be delivered at multiple hosts. These configurable features are usually specialized for each family member to profit from eventual hardware support.

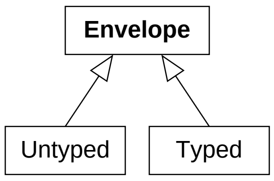

Figure 4.16: EPOS family of message envelopes.

4.3.4.4 Message Envelopes

The members of the `Communicator` family can be used to exchange unstructured messages in the form of sequences of bytes of a certain length. However, it might be adequate for some applications to count on an *envelope* abstraction to cover a message before it is sent. Such an envelope would be allocated from the operating system, loaded with one or more messages, and then inserted into a communication channel through a communicator. An envelope allocated by the operating system would enable several optimizations, ranging from cache alignment to zero-copy processing. Besides, additional information can be put in the envelope to describe and protect messages. After all, an envelope would enable a comfortable syntax to express communication in object-oriented applications, for example:

```
Envelope envelope(recipient, length);
envelope « "Hello␣world!";
communicator « envelope ;
```

EPOS supports the concept of *message envelope* through the `Envelope` uniform family of abstractions represented in figure 4.16. The maximum length of message that an envelope can hold is specified when it is instantiated, while the effective length of the message(s) it contains is dynamically determined. An envelope must be addressed before it is posted.

The `Envelope` family comprises two members: `Untyped` and `Typed`. The former realizes a simple message envelope that can be used to gather messages before sending, while the latter collects type information for each message inserted to enable format conversions on heterogeneous systems. A *secure envelope* was not modeled due to the characteristics of a dedicated computing system, which usually do not require encryption nor authentication of messages[8].

4.3.5 Time Management

Time is managed in EPOS by the `Timer` family of dissociated abstractions shown in figure 4.17. The `Clock` abstraction is responsible for keeping track of the current time. It

[8]A discussion about protection in dedicated systems is presented in section 4.4.4.

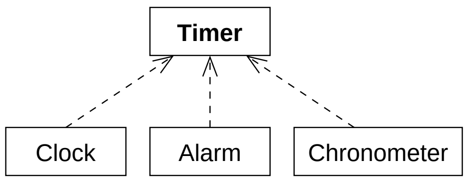

Figure 4.17: EPOS family of timers.

is only available on systems that feature a real-time clock device. The `Alarm` abstraction can be used to put a thread to "sleep" for a certain time. It can also be used to generate timed events. For this purpose, an application instantiates an abstraction from the `Interrup_Handler` family (see section 4.3.6.3) and registers it with `Alarm` specifying a time interval and the number of times the handler object is to be invoked[9].

The `Timer` family is completed by the `Chronometer` abstraction, which is used to perform time measurements. The unit of reference for these abstractions is the *second*, with times and delays represented as real numbers. The precision of these abstractions, however, depends on the hardware platform on which they are implemented.

4.3.6 I/O Management

The interaction between applications and peripheral devices is managed in EPOS by the families of abstractions represented in figure 4.18. As a rule, peripheral devices are abstracted in a way that is convenient to applications by the members of the `Device` family. However, dedicated systems often deploy dedicated devices that will not be found on this family. Therefore, EPOS also delivers applications means to directly interact with a peripheral device. In this context, the `Bus` family is responsible for detecting and activating devices connected to a bus, which are abstracted as dynamically created members of the `Device` family. The `Interrupt_Handler` family of abstractions allows applications to handle interrupts generated by a device.

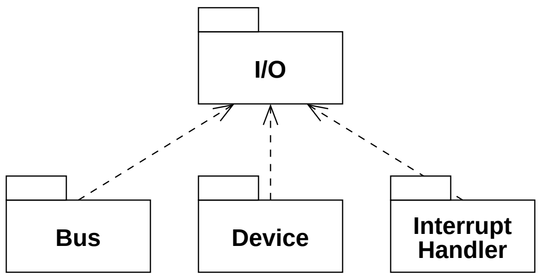

Figure 4.18: Families of abstractions concerning I/O management in EPOS.

[9]This mechanism is also used to activate the process scheduler.

4.3.6.1 Buses

EPOS family of I/O bus abstractions is responsible for detecting and activating physical devices connected to a bus. The `Bus` family sketched in figure 4.19 was modeled as a dissociated family, with a member for each supported bus technology (e.g. ISA, PCI, SCSI). When a member of the `Bus` family is invoked to activate a device, it arranges for the application process to have total control over the device, mapping eventual memory and I/O ports to its address space. According to the bus technology, operations are provided to further configure devices and sometimes the bus itself.

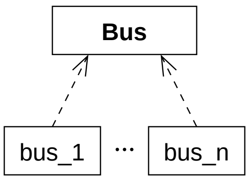

Figure 4.19: EPOS family of buses.

4.3.6.2 Devices

From a historic perspective, *device drivers* are some of the most important pieces of an operating system. From the beginning, operating systems have the duty of hiding the peculiarities of each device from application programs, easing programming and allowing them to survive steady upgrades. As an application-oriented system, EPOS extends the notion of device driver as a set of I/O routines, modeling devices as abstractions organized in a family. The makeup of this family is depicted in figure 4.20. Differently from the families of abstractions presented until now, the commonalities of the `Device` family are not collected in a single package, but split according to bus technologies (`bus_1_dev` through `bus_n_dev` on the diagram). Each bus technology defines a dissociated subfamily of devices.

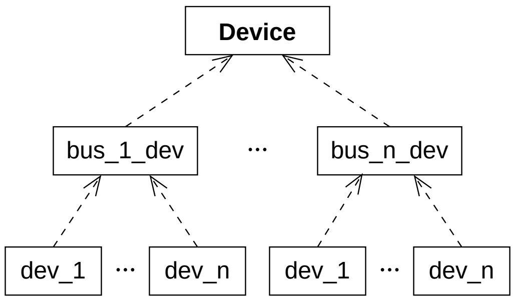

Figure 4.20: EPOS family of device abstractions.

The decision for a dissociated family of devices defies the uniform organization consolidated by UNIX, in which all devices adhered to a common pseudo-file interface [Tho78]. This interface has an "escape" operation, namely `ioctl`, that is used to pack operations that cannot be represented under the file interface[10]. EPOS gives priority to the preservation of the individual characteristics of each device, allowing them to define their own operations and eliminating the "hideous" `ioctl` operation. Nevertheless, the lack of a common interface makes it more difficult to add a device at run-time. Indeed, when an application invokes a member of the `Bus` family to detect and activate a device, the returned device abstraction is supposed to be handled at user-level in the scope of the calling process. All but bus-specific operations for this device have to be implemented by the application itself. When EPOS is configured as a kernel, neither the kernel nor other processes know about such a device. If sharing is required, a device "server" process has to be devised.

4.3.6.3 Interrupt Handlers

EPOS allows application processes to handle hardware generated interrupts at user-level via the `Interrupt_Handler` family of abstractions depicted in figure 4.21. The three members of this dissociated family can be used simultaneously. The `Function_IH` member assigns an ordinary function supplied by the application to handle an interrupt. The system transforms such a function in an interrupt handler that is invoked every time the associated interrupt is generated. In contrast, the `Thread_IH` member assigns a thread to handle an interrupt. Such a thread must have been previously created by the application in the *suspended* state. It is then resumed at every occurrence of the corresponding interrupt. After handling the interrupt, the thread must return to the suspended state by invoking the respective operation.

Nevertheless, these two members of the `Interrupt_Handler` family may present problems if an interrupt is successively generated while the associated handler is active. The `Function_IH` handler passes the issue on to the application, which must either grant the handler is fast enough, or implement a reentrant function. The `Thread_IH`

[10]LINUX `ioctl_list man page` brings an incomplete list of `ioctl` operations valid on kernel 1.3.27. It comprises 412 operations.

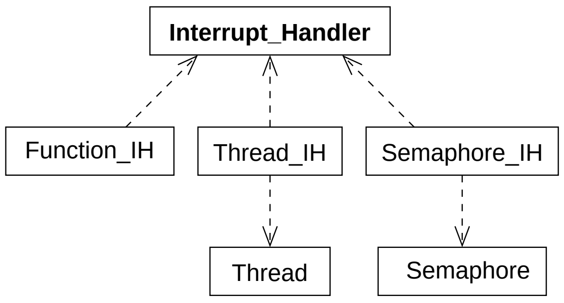

Figure 4.21: EPOS family of interrupt handlers.

handler is even more restrict in this regard, since resuming an already active thread has no effect.

The issue of interrupt loss is addressed by the `Semaphore_IH`, which assigns a semaphore, previously created by the application and initialized with zero, to an interrupt. The operating system invokes operation `v` on this semaphore at every interrupt, while the handling thread invokes operation `p` to wait for an interrupt. The strategy gives interrupt handling a producer/consumer flavor and prevents interrupts from being lost.

Nevertheless, preventing interrupts from being lost may not be enough to grant that I/O data will not be lost. A device that generates a subsequent interrupt while a former is still being handled must have memory to accumulate eventual data. If this is not the case, a handler must be conceived that is able to match up the device's operational speed. Moreover, depending on the scheduling policy in force, interrupts may not be handled immediately. For example, if the thread responsible for handling an interrupt has a lower priority than the thread currently being executed, the handler will be enqueued for later execution.

4.3.7 External Abstractions

Some of the system services offered by EPOS are not implemented on the dedicated computing system itself, but on an external server. For example, file services and graphic output. In order to enable these services to be remotely accessed, EPOS communication system is emulated over an ordinary operating system. A computer running this emulator becomes a gateway to an EPOS-based environment that allows external abstraction to be accessed via remote object invocation (see section 4.4.7). This scheme is illustrated in figure 4.22 (EPOS is represented in the figure on its most typical architecture, i.e. fully embedded in the application).

The set of "stub" functions used to perform remote invocation can also be used to give external services an EPOS-flavor. For instance, POSIX file operations can be delivered through a file abstraction.

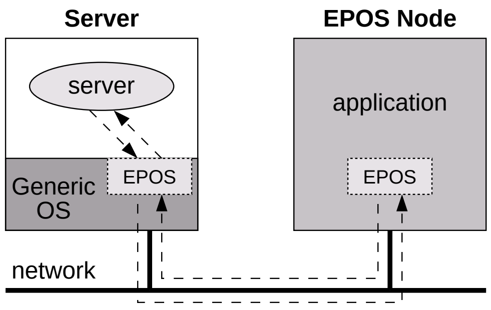

Figure 4.22: Access to external abstractions in EPOS.

4.3.8 Summary of EPOS Abstractions

A summary of EPOS abstractions is presented in tables 4.1 and 4.2. Families of abstractions are grouped by category. A brief description of responsibilities, type, and dependencies are given for each family and for each member of a family.

Family	Responsibilities	Type	Dependencies
	Memory Management		
Segment	logical memory segments	$\oplus$	Address_Space
Static	not resizable, flags not changeable	$\Diamond\star$	-
Dynamic	resizable, remappable	$\Diamond\star$	Paged
Address_Space	logical address space of a process	$\ominus$	-
Flat	single, contiguously mapped address space	$\Box\star$	-
Paged	multiple, paged address spaces	$\Box\star$	-
	Process Management		
Task	code, data, and resources of a process	$\ominus$	Segment
Exclusive	single-tasking	$\Diamond\star$	Flat
Mutual	multitasking	$\Diamond\star$	Paged
Thread	stack and context of a process	$\oplus$	Segment
Exclusive	single-threading	$\Diamond\star$	Exclusive_Task
Cooperative	cooperative multithreading (no scheduler)	$\Diamond\star$	-
Concurrent	concurrent, scheduled multithreading	$\Diamond\star$	CPU_Scheduler
CPU_Scheduler	processor scheduling policies	$\odot$	Alarm
FCFS	first-come-first-served	$\Box\star$	-
RR	round-robin	$\Box\star$	-
Static_Prio	static priorities	$\Box\star$	-
Dynamic_Prio	dynamic priorities	$\Box\star$	RR,Static_Prio
MultiLevel	multiple policies	$\Box$	-
UserDef	user defined policy	$\Diamond$	-
	Process Coordination		
Synchronizer	process coordination	$\ominus$	-
Mutex	mutual exclusion device	$\Diamond$	-
Semaphore	semaphore variable	$\Diamond$	-
Condition	condition variable	$\Diamond$	-

Family types: $\oplus$ incremental, $\ominus$ dissociated, $\odot$ uniform.
Member types: $\Diamond$ visible to applications, $\Box$ invisible, $\star$ mutually exclusive.

Table 4.1: EPOS abstractions (part I).

Family	Responsibilities	Type	Dependencies
	Inter-Process Communication		
Communicator	communication end-points	⊖	Channel
Link	connection for streams	◊	Stream
Port	non-sharable mailbox for datagrams	◊	Datagram
Mailbox	sharable mailbox for datagrams	◊	Datagram
ARM_Segment	asynchronous remote memory segment	◊	ARMx
AM_Handler	active message handler	◊	AM
Channel	communication channels	⊖	Network
Stream	streams	□	-
Datagram	datagrams	□	-
ARMR	asynchronous remote memory read	□	-
ARMW	asynchronous remote memory write	□	-
ARMC	asynchronous remote memory copy	□	-
AM	active messages	□	-
Network	logical networks (OSI level 4)	⊙	Device
Several	not covered in this table	◊	
Envelope	message envelopes	⊙	-
Untyped	envelope for untyped messages	◊	-
Typed	envelope for typed messages (heterogeneity)	◊	-
	Time Management		
Timer	time keeping	⊖	-
Clock	current time keeping	◊	-
Alarm	timed event generation	◊	-
Chronometer	time measurement	◊	-
	I/O Management		
Bus	device detection and activation	⊖	-
Several	not covered in this table	◊	
Device	physical device abstraction	⊖	-
Several	not covered in this table	◊	
Interrupt_Handler	interrupt handling	⊖	-
FunctionIH	calls a handling function	◊	-
ThreadIH	resumes a handling thread	◊	Thread
SemaphoreIH	invokes v on a handling semaphore	◊	Semaphore

Family types: ⊕ incremental, ⊖ dissociated, ⊙ uniform.
Member types: ◊ visible to applications, □ invisible, ⋆ mutually exclusive.

Table 4.2: EPOS abstractions (part II).

4.4 Scenario Aspects

Application-oriented system design is particularly concerned with scenario independence. When a domain entity is identified, considerations about its origin are made in order to decide whether it will shape an abstraction or a scenario aspect. EPOS scenario aspects were modeled in accordance with this principle, yielding reusable pieces of software that can be controllably applied to the system abstractions described in the previous section.

Likewise system abstractions, EPOS scenario aspects were organized in families according to what they have in common. The commonalities of each family are captured in a *common package* that is reused by member aspects. Accordingly, families of aspects were classified as *uniform*, *incremental*, *dissociated*, or *combined*.

In order to build a scenario for an application, selected scenario aspects are merged in a *scenario* construct and applied to abstractions by means of a *scenario adapter*. These two constructs, however, will be described in the scope of EPOS component framework later in section 4.5.1.

4.4.1 Identification

"An object has state, behavior, and identity."

(Grady Booch [Boo94])

This axiom of object-orientation, which evidently also applies to the instances of system abstractions[11], impart that an object has an identity that distinguishes it from all the others. At programming language level, this identity is usually represented by the object's address in memory (pointer). This form of identity is rather adequate if objects are only manipulated in the scope of the process that created them, but it is inadequate to identify objects that are shared by multiple processes. In this case, the operating system has the duty of generating identifiers for system objects that are able to distinguish them across all processes. If objects can be shared in a distributed environment, identifiers must be extended to accomplish unique identification across all nodes in the platform. This variability concerning the identity property is typical of scenario aspects, for it affects the manifestation of an abstraction in a scenario, but not its internal structure.

Therefore, the identity of system objects was modeled in EPOS as an incremental family of scenario aspects. This family is represented in figure 4.23. Member `Pointer`, the family's basic aspect, simply reuses the fundamental identity of objects, i.e. their address in memory. Since this form of identifier is always relative to the address space of the process that created the object, it cannot be used for inter-process interactions on objects. The `Local` member extends the `Id` family with an identifier that consists of a pair `(type,`

[11]Instances of system abstractions will be designated "system objects" hereafter, or simply "objects" when no confusion can arise. In this sense, an abstraction corresponds to a "class of system objects".

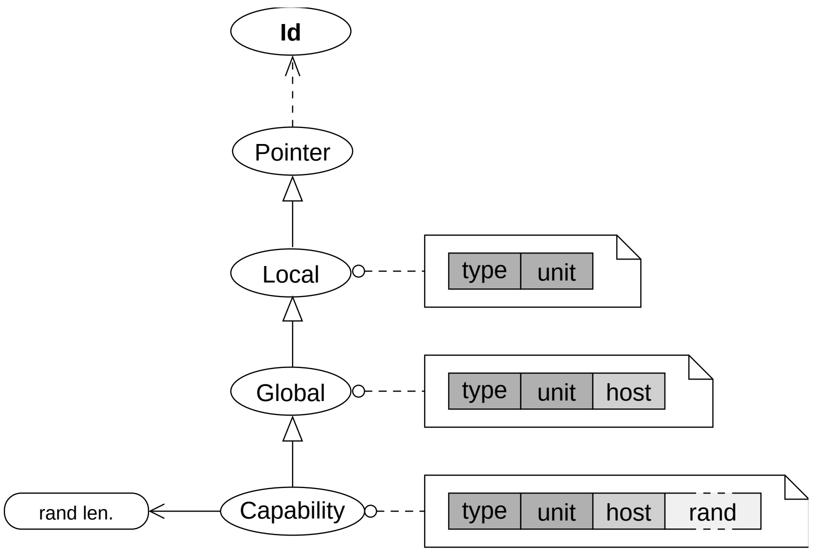

Figure 4.23: EPOS family of identification aspects.

`unit)` and uniquely identifies a system object in a stand-alone system configuration. The `type` field is obtained from the *traits* of an abstraction [section 4.5.1], while `unit` represents the rank of an object in the allocation table of its class. Hence, the first instance of a `Myrinet` network abstraction would be identified as `(Traits<Myrinet>::type, 0)`.

Member `Global` extends the `Id` family so that a network-wide identifier is assigned to system objects, allowing them to be remotely accessed. The structural extension succeeded by this member consists of adding a logical host number to identifiers, yielding the tuple `(host, type, unit)`. Although embedding location in identifiers complicates the migration of objects in a distributed environment, EPOS opted for this solution because it is efficient and fulfills the demands of most parallel systems. Differently from distributed systems, dedicated parallel systems seldom deploy object migration due to the high overhead associated. Other load balancing techniques that do not conflict with the `Global` identifier, such as data set partitioning and global scheduling, are often preferred [SP94b]. Nevertheless, the logical ids realized by this family could be locally remapped to reflect the location of migrated objects.

`Local` and `Global` identifiers have a lifetime that corresponds to the lifetime of the system. That is, if a process creates a system object and later deletes it, a newly created object may be assigned the same identifier formerly assigned to that object. This, however, should not disturb applications, since EPOS keeps a reference counter that prevents objects in a shared scenario (i.e. a scenario for which the `Shared` aspect [section 4.4.2] has been enabled) from being deleted while there are still processes using it. Nonetheless, flawed programming may cause a process to keep on using an identifier after the respective system object has been destroyed, inadvertently accessing another object. If these incidents are to be handled by the operating system, the `Capability` identifier must be used, since its lifetime is restricted to the lifetime of the object it identifies.

The `Capability` identifier extends the `Id` family by adding a randomly generated number to the `Global` identifier. System objects are thus identified by the tuple `(host, type, unit, rand)`. The `rand` field makes identifiers sparse, reducing the probability of an identifier being reused to a negligible level [MT86]. The exact length of this field, however, depends on the quality of the random number generator used and on the dynamics and lifetime of applications. Hence, it was modeled as a configurable feature.

4.4.2 Sharing

There are many reasons that lead processes to share resources. Synchronizers are shared to accomplish coordination, memory segments are shared as an intra-node communication mechanism, mailboxes are shared to support sever replication, and so forth. When system resources are shared, the operating system has to provide means to preserve their integrity. For example, if a shared resource has its state modified by one of the sharing processes, the remaining must be assured a coherent view of it. In particular, when a shared resource is destroyed by one of the sharing processes, subsequent operations invoked by the remaining processes have to be handled accordingly.

EPOS supports abstractions to be shared through the `Shared` incremental family of scenario aspects depicted in figure 4.24. When this aspect is enabled, abstractions are tagged with a reference counter by the `Referenced` family member, so that a shared abstraction is actually only destroyed when the last sharing process releases it. This scenario aspect is specialized for the family of memory segments [section 4.3.1.1] to provide adequate support for memory sharing.

The `Enrolled` member of the `Shared` family of aspects extends `Referenced` to sustain multitasking configurations of EPOS in which resources need to be reclaimed. In this scenario, EPOS takes note of which tasks are sharing each resource, so that resources not returned by an exiting process can be reclaimed.

When the `Shared` aspect is enabled, system abstractions gain two extra construc-

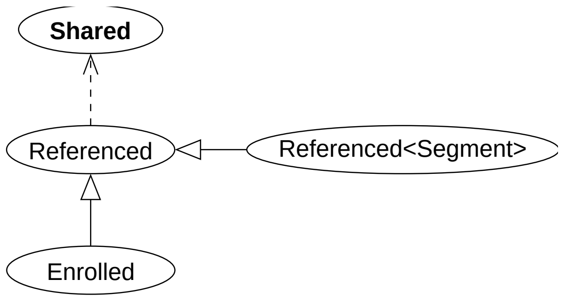

Figure 4.24: EPOS family of sharing aspects.

tors[12] that are used to designate sharing. The first is the ordinary copy constructor, which takes a reference to an existing system object as argument and returns a share to that object. This constructor is restricted to share abstractions inside the same address space. The second constructor takes an identifier as argument and therefore can be used independently of locality, inclusive to share remote objects. A C++ program could deploy these constructors as follows:

```
Abstraction instance;
Abstraction share1(instance);
Abstraction share2(share1.id ());
```

4.4.3 Allocation

In all-purpose operating systems, the memory internally used to store the state of system-level abstractions is expected to be dynamically allocated, since it is impossible to predict which abstractions and how many instances of each abstraction will be effectively required by forthcoming applications. Some dedicated computing systems, however, have a predictable demand for resources. In these cases, pre-allocating resources may significantly enhance performance. For example, if a process is known to spawn n threads, pre-allocating these threads—partially initializing the associated control structures and allocating the corresponding stacks—may eliminate a considerable fraction of the thread creation overhead.

A more imperative reason to avoid dynamic memory allocation can be observed for abstractions that have (part of) their state stored outside main memory. Several abstractions of physical devices fall in this category. If the device includes memory that is shared with the main processor, or if it memory-maps control registers, then the corresponding system abstraction will have part of its state mapped over the I/O bus. A dynamic memory allocator unaware of such particularities would allocate incoherent instances of the abstraction; making the memory allocator aware of them would compromise portability, for device mapping is an operation that depends on the I/O bus. Situations like this could be handled by an utility that would execute previously to the operating system to pre-allocate indicated abstractions, delivering them later to the operating system. One such a setup utility was designed for EPOS and will be described later in section 4.5.2.1.

EPOS supports two allocation scenarios for abstractions: `Late` and `Ahead`. Both are modeled as members of the `Allocated` dissociated family of scenario aspects presented in figure 4.25. The former member defines a scenario in which the memory needed to hold the state of system abstractions is dynamically allocated as the abstraction is instantiated, while the latter assumes it has been previously allocated. The `Ahead` aspect relegates

[12]The term *constructor* is being used here independently from the C++ programming language to designate a function used to instantiate an object. Applications written in programming languages that do not explicitly support the concept would be supplied operations with equivalent semantics.

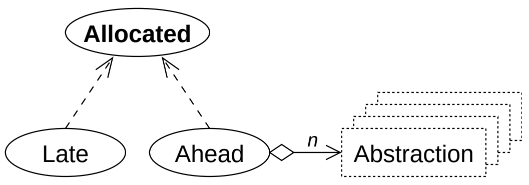

Figure 4.25: EPOS family of allocation aspects.

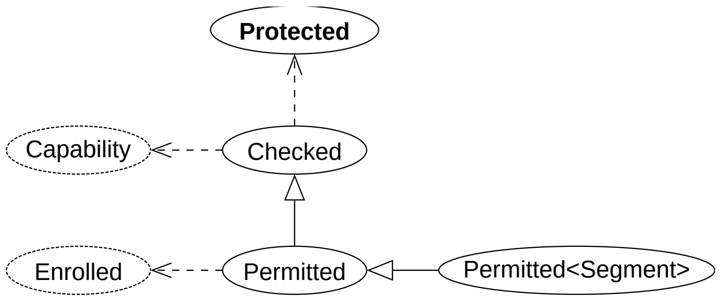

Figure 4.26: EPOS family of protection aspects.

the allocation of many abstractions to marking instances "busy". EPOS knows how many instances of an abstraction have been pre-allocated consulting its *traits*.

A scenario in which the `Late_Allocated` aspect is used with exclusivity is not probable, since some abstractions (e.g. devices) will always be allocated in advance. Therefore, the application of this scenario aspect must be evaluated individually for each abstraction (consulting its *traits*).

4.4.4 Protection

In multitasking environments, various processes compete for memory, processor, network, and other system resources. In such environments, a *protection* mechanism to ensure that *processes* do not inappropriately interfere with one another's activities is desirable. In order to gain access to resources, processes would thus have to obtain explicit authorization from the operating system.

Protection in generic systems tends to extend towards *security*, which concerns preventing unauthorized *users* from interfering with ongoing computations. However, the mechanisms of authentication and encryption used in the scope of distributed and web-based computing to attain security are rarely required for dedicated computing, because, as a matter of fact, dedicated systems are (temporary) single-user systems. Therefore, EPOS protection mechanisms are restricted to control access to resources. These mechanisms are modeled by the `Protected` incremental family of scenario aspects depicted in figure 4.26.

The `Checked` aspect of the `Protected` family profits from the `Capability` identifier described in section 4.4.1 as a protection mechanism. That identifier uses a sparse random number to achieve identification uniqueness, which can also serve protection purposes: in order to access a system object, a process has to know its capability. The probability of gaining access to an object by betting can be made insignificant if capabilities are sufficiently sparse, what is controlled by the `random_length` configurable feature of the `Capability` aspect. Lengths ranging from 32 to 64 bits shall be in order for most systems.

The `Permitted` protection aspect extends `Checked` to build a protection scenario in which operations on abstractions are individually authorized or denied for each process [MT86]. In order to achieve this scenario, `Permitted` wraps the random number generator used by the `Capability`, adding permission flags to the `rand` field (the `rand_length` configurable feature is adjusted to accommodate these extra bits). A new operation on objects of type `Capability` is supplied that produces restricted versions of a capability as a non-reversible function of the unrestricted version (the *owner* capability). A list with owner capabilities corresponding to the objects created by each process is maintained by the `Enrolled` sharing aspect. Access control is thus enforced by the scenario adapter, which checks for permission before allowing an operation to proceed.

This family of scenario aspects is specialized for the `Segment` abstraction [section 4.3.1.2] to accomplish memory protection if the hardware so allows. In this way, segments are attached to the address space of a process respecting the permissions in the supplied capability. Memory protection is also useful in single-tasking environments: by properly protecting segments, programming mistakes such as "lost pointers" can be detected, thus helping to debug applications. When protection is violated, EPOS produces a useful report of the circumstances. Besides, EPOS always protects its own memory segments when the `Protected` scenario is enabled.

4.4.5 Timing

A thread often goes waiting for events such as I/O completion, thread termination, and message arrival. Some of these events, however, may suffer delays beyond acceptable limits; some may not occur at all. In some cases, it may be more adequate for an application to lose a message sent over a jammed network than delaying a computation to wait for it. Similarly, it may be better for a control application to succeed a predefined action than missing a deadline waiting for user intervention. Such situations can be managed by assigning a time limit to operations that may cause a thread to wait. If the operation does not conclude within the specified interval, it is terminated by the operating system with a failure status.

A mechanism to specify time-outs for operations was modeled in EPOS as a global scenario aspect. However, associating time-outs to operations that are not eligible to suffer delays would add unnecessary overhead to the system. Therefore, the `Limited` scenario aspect responsible for time-outs is only applied to abstractions after consulting

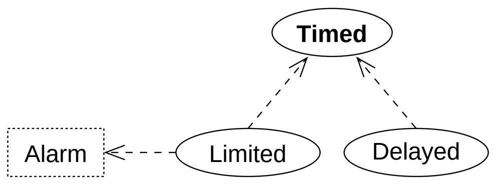

Figure 4.27: EPOS family of timing aspects.

their *traits* [section 4.5.1]. This scenario aspect belongs to the `Timed` dissociated family represented in figure 4.27. `Limited` relies on the `Alarm` timer to implement time-outs.

A timing aspect that is less often deployed is realized by the `Delayed` member of the `Timed` family. It supports operations on system abstractions to be delayed by a certain time. It is useful, for instance, to migrate embedded applications with strict timing dependencies to a faster hardware platform, or to achieve a homogeneous execution time for processes of a parallel application running on heterogeneous speed nodes.

4.4.6 Atomicity

Events in a parallel program are no longer strictly ordered as in sequential programs, they can occur simultaneously. When multiple threads are allowed to execute in parallel, it its possible that they simultaneously “enter” the operating system, i.e. pass to execute operating system code to accomplish a system service. An operating system that allows for this scenario is said to be reentrant and is itself subject to race conditions. Reentrance is often approached in monolithic systems by properly coordinating the execution of critical sections identified in the course of system development. The dynamic architecture of an application-oriented operating system, however, makes this solution impractical. The combination of abstraction, aspects, and configurable features may have profound consequences on system architecture, invalidating some of the hand-made critical sections.

Therefore, EPOS handles the synchronization pitfalls brought about by reentrance ensuring that system operations are *atomic*. In this way, transformations of the state of system objects either occur completely or do not occur at all. The atomicity property was modeled to be orthogonal to abstractions, yielding the uniform family of scenario aspects depicted in figure 4.28.

The `Uninterrupted` aspect achieves atomicity by disabling the generation of hardware interrupts that could cause the processor to be preempted from a thread in the middle of a system operation. This family member has a low overhead, but it may also have undesirable side-effects on applications. Disabling interrupts may spoil I/O subsystems due to delays in interrupt handling and cause “scheduling skew”. Furthermore, the `Uninterrupted` scenario aspect is not suitable to be used in multiprocessor environments, since threads executing in parallel can simultaneously invoke system abstractions independently of interrupts.

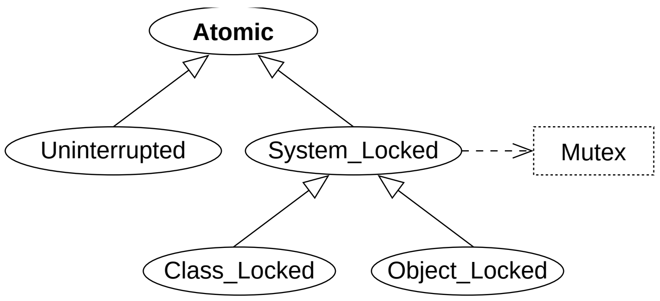

Figure 4.28: EPOS family of atomicity aspects.

Therefore, three other members were modeled for the `Atomic` family that do not assume interrupts to be disabled. All three deploy the `Mutex` member of the `Synchronizer` family of abstractions [section 4.3.3.1] to transform every system operation in a critical section[13]. They also share similar run-time overheads, corresponding to the invocation of `Mutex` methods to accomplish the mutually exclusive execution of system operations. Besides, they lock unallocated system resources under a single mutex, providing atomicity to constructors, destructors, and other class operations.

These three members differ in the degree of parallelism they sustain. The `System_Locked` member uses a single mutex for all abstractions, yielding a scenario in which a single thread is allowed to execute system operations at a time (null reentrance). Reentrance is actually supported by the `Class_Locked` and `Object_Locked` scenario aspects. The former assigns a different mutex to each class of system objects, so that operations on objects of different types can be invoked simultaneously. The latter assigns each system object its own mutex, achieving the highest level of reentrance[14]. Although the run-time overhead incurred by these scenario aspects is equivalent, their consumption of synchronization resources is extremely diverse: `System_Locked` uses a single mutex, `Class_Locked` uses a "global" mutex plus one mutex for each abstraction configured for the system, and `Object_Locked` uses one mutex for each system object created plus the "global" one.

4.4.7 Remote Invocation

In a distributed environment, processes may need to access system resources residing on remote nodes. In order to do so, an application would have to create a process on each node containing useful resources and deploy a communicator to interact with them.

[13]When the `idle waiting` configurable feature of `CPU_Scheduler` is enabled in an `Atomic` scenario, the operation used to block a thread is modified so that blocked threads temporarily leave the critical section, thus enabling other threads to invoke system operations. This modification affects only the critical sections defined by the `Atomic` aspect; critical sections defined by the application are not affected.

[14]System abstractions that are intrinsically atomic, such as `Mutex`, are properly escaped by the scenario adapter [section 4.5.1].

However, the burden of explicit message passing for accessing remote resources can be eliminated by a *Remote Procedure Call* (RPC) [BN84] or, in an object-oriented context, by a *Remote Object Invocation* (ROI) [LT91] mechanism. These mechanisms hide inter-process communication behind ordinary method invocations, so that processes can transparently access remote resources.

EPOS supports remote object invocation as a scenario aspect that can be transparently applied to virtually any abstraction. This aspect is realized by the `Remote` scenario aspect represented in figure 4.29. It relies on the `Port` communicator for message exchange, on the `Global_Id` (or a derivative) scenario aspect for object location, and on the `Shared` scenario aspect to control global sharing.

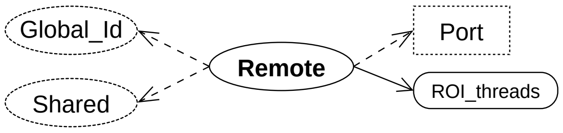

Figure 4.29: EPOS remote invocation scenario aspect.

An object invocation in a `Remote` scenario is depicted in figure 4.30. The `object` being remotely manipulated is represented in the `client`'s process domain by a `proxy`, and mediated on its own domain by an `agent`. When an operation is invoked on the `object`'s `proxy`, the arguments supplied are marshaled in a `request` message and sent to the `object`'s `agent`. This message is addressed to a well-known ROI `port` on the node on which `object` resides (which is designated by the `host` field of its identifier). This port is listened by the `agent` using a private thread, thus preventing the blockage of ongoing computations. When the `agent` receives a `request`, it unpacks the arguments and performs a local method invocation. The whole process is then repeated in the opposite direction, producing a `reply` message that carries eventual return arguments back to the client process.

The number of threads effectively created by the system to listen the ROI port is controlled by the `ROI_threads` configurable feature, with zero meaning that threads are dynamically created for each invocation. The placement of the ROI port and ROI agents, however, varies according to EPOS resultant architecture. In the embedded-in-the-application configuration, these entities belong to the single application process, and, in the μ-kernel configuration, they belong to a separate ROI server process.

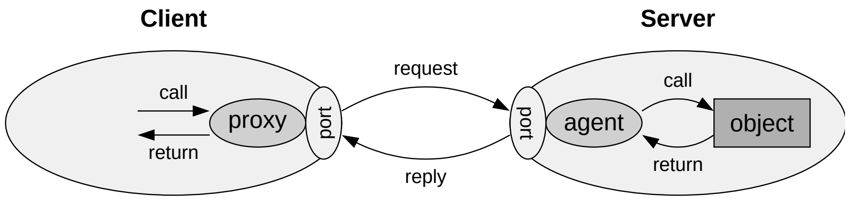

Figure 4.30: A remote object invocation in the `Remote` scenario.

Applying the `Remote` scenario aspect to abstractions would require a special scenario adapter, able to cope with proxies and agents. This could be a problem for ordinary scenario adapters that are hand-written, since a new adapter would have to be defined for every abstraction. This problem was eliminated by inserting placeholders in EPOS component framework to install ROI proxies and agents (figure 4.31), which can be automatically generated from the interface of abstractions by a tool (see section 4.6 for a description of how EPOS syntactical analyzer can be used to obtain the operation signatures of all abstractions). These proxies and agents would be arranged in the proper places when `Remote` is enabled, being replaced by metaprogrammed dummies, which are completely eliminated during compilation, otherwise.

A remote object invocation is firstly expressed in a program using the extra constructor defined by the `Shared` aspect for all system abstractions. A process knowing the id of a remote object can use such a constructor to obtain a remote share of the object. Subsequently, that process can invoke operations on the object disregarding locality. For example, a remote thread could be suspended as follows:

```
Thread thread(remote_thread_id);
thread.suspend ();
```

and a thread could be created on a remote node in this way:

```
Thread remote_thread(remote_task_id, entry_point, arguments ...);
```

The actual owner of this thread would be the ROI agent on the node where it was created.

4.4.8 Debugging and Profiling

Being able to trace the invocation of system operations, or to watch the state of system abstractions, can be useful to debug application programs. Likewise, being able to summarize how much time an application spends with each system abstraction may be a source of optimization.

EPOS supports these features through the `Debugged` family of dissociated scenario aspects depicted in figure 4.32. The device used to report debugging and profiling information is selected through the `outupt_dev` configurable feature. The scenario aspects

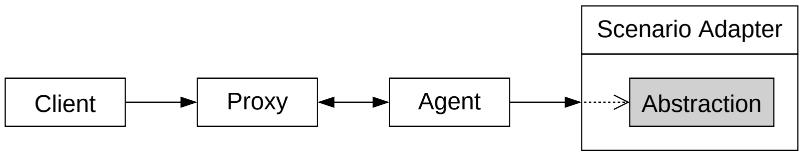

Figure 4.31: The `Remote` scenario aspect adapter.

in this family are interpreted as follows: `Watched` causes the state of a system object to be dumped every time it is modified; `Traced` causes every system invocation to be signalized; and `Profiled` audits the time spent by a process with each system operation, producing a report when the process terminates. The amount of information produced for each abstraction is controlled by its *traits*.

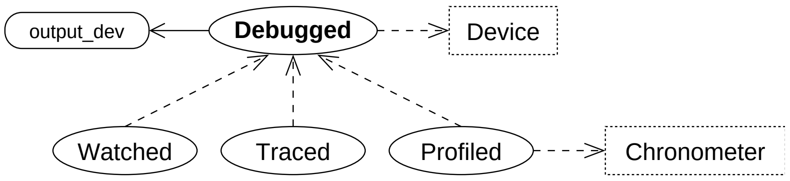

Figure 4.32: EPOS family of debugging and profiling aspects.

4.4.9 Summary of EPOS Scenario Aspects

A summary of EPOS scenario aspects is presented in table 4.3. A brief description of responsibilities and dependencies are given for each family of scenario aspects and for each member of a family.

Scenario Aspect		Responsibilities	Dependencies
Identification	$\star\oplus$	system object identification	-
Pointer		intra-process id	-
Local		local node id	-
Global		SAN-wide id	-
Capability		sparse SAN-wide id	-
Sharing	$\star\oplus$	sharing of system objects among processes	-
Referenced		reference counter	-
Enrolled		clients list	-
Allocation	$\ominus$	memory allocation for system objects	-
Late		dynamic allocation	-
Ahead		pre-allocation	Setup
Protection	$\star\oplus$	system object access control	-
Checked		id knowledge	Capability
Permitted		permitted operations	Enrolled
Timing	$\ominus$	system operations timing	Alarm
Limited		time-out	-
Delayed		delay	-
Atomicity	$\star\odot$	system operations reentrance	-
Uninterrupted		interrupt disabling	-
System_Locked		global monitor	Mutex
Class_Locked		abstraction monitor	Mutex
Object_Locked		system object monitor	Mutex
Remote		remote object invocation	Port, Global_Id, Shared
Debugging	$\ominus$	system debugging and profiling	Device
Watched		system object watching	-
Traced		system operation tracing	-
Profiled		system operation profiling	-

Aspect types: $\oplus$ incremental, $\ominus$ dissociated, $\odot$ uniform. $\star$ mutually exclusive.

Table 4.3: EPOS scenario aspects.

4.5 System Architectures

As an application-oriented operating system, EPOS has a highly scalable architecture that is modeled to accomplish the needs of applications it supports. Ultimately, EPOS architecture results from the organization of the components selected for a given system configuration. Distinct combinations of system abstractions and scenario aspects lead to different software architectures, some of which are delivered as a kernel, others are completely embedded in the application. Therefore, the component framework represents the core of EPOS software architecture, for it dictates how abstractions can be combined considering the peculiarities of the target execution scenario.

Notwithstanding the significance of a component framework to an application-oriented system design, a thorough handling of portability and initialization issues is fundamental to accomplish a scalable system architecture. If hardware architectural aspects are absorbed by the operating system's software architecture, porting it to another platform could obliterate much of the flexibility offered by the component framework. Besides, the bare hardware on which an operating system performs is seldom prepared to deal with the high-level language constructs of a component framework. The hardware platform has to be appropriately initialized to house a composite of application-oriented system abstractions.

4.5.1 Component Framework

An application-oriented component framework captures elements of reusable system architectures while defining how abstractions can be arranged together in a functioning system. In this context, EPOS component framework was modeled as a collection of interrelated scenario adapters that build a "socket board" for system abstractions and scenario aspects. These are "plugged" to the framework via inflated interface binding.

EPOS component framework is realized by a *static metaprogram* and a set of *composition rules*. The metaprogram is responsible for adapting system abstractions to the selected execution scenario and arranging them together during the compilation of an application-oriented version of EPOS. Rules coordinate the operation of the metaprogram, specifying constraints and dependencies for the composition of system abstractions. Composition rules are not encoded in the metaprogram, but specified externally. They are interpreted by composition tools in order to adjust the parameters of the metaprogram.

The separation of composition rules from the framework metaprogram allows a single framework to yield a variety of software architectures. Indeed, one could say that EPOS has many frameworks, each corresponding to the execution of the metaprogram with a different set of arguments. Moreover, the use of static metaprogramming to compose system abstractions does not incur in run-time overhead, thus yielding composites whose performance is directly derived from their parts.

4.5.1.1 Framework Metaprogram

EPOS *component framework metaprogram* is executed in the course of a system instance compilation, adapting selected abstractions to coexist with each other and with applications in the designated execution scenario. During this process, scenario-independent abstractions have their original properties preserved, so that internal compositions can be carried out before scenario adaptation. This is accomplished having the framework metaprogram to import scenario-independent abstractions in one namespace and export the corresponding scenario-adapted versions in another.

For example, the cascaded aggregation of `Communicator`, `Channel`, and `Network` [section 4.3.4] takes place at the scenario-independent level. The resultant composite is later adapted to the selected scenario as a whole. Similarly, the `Atomic` scenario aspect instantiates a scenario-independent `Mutex` [section 4.4.6], i.e. without the adaptations performed by the framework metaprogram, which might have transformed it, for instance, in a remotely accessible, shared, and protected abstraction to match up the scenario required by the application.

Figure 4.33 shows a top-view diagram of the component framework static metaprogram. Each of the elements represented in the figure will be subsequently described. A class diagram representing the `Handle` framework element is depicted in figure 4.34. Like most other elements, parameterized class `Handle` takes a system abstraction (class of system objects) as parameter. When instantiated, it acts as a "handle" for the supplied abstraction, realizing its interface in order that invocations of its methods are forwarded to `Stub`. Hence, system objects are manipulated by applications via their "handles".

`Handle` provides additional operations to check if a system object was successfully created[15] and to obtain its id. Besides, when the `Shared` scenario aspect [section 4.4.2]

[15]The use of C++ exceptions as a mechanism to signalize system object creation failure was avoided

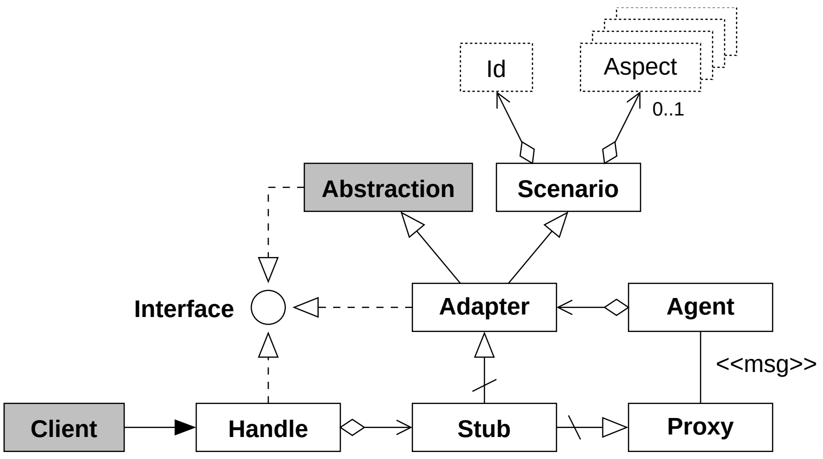

Figure 4.33: A top-view of EPOS component framework metaprogram.

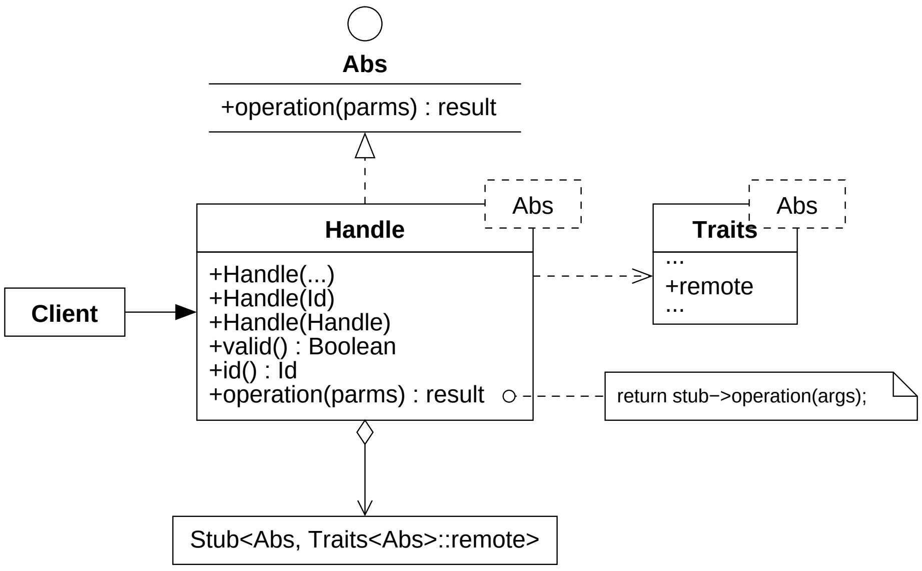
Abs
+operation(parms) : result
Handle
Abs
+Handle(...)
+Handle(Id)
+Handle(Handle)
+valid() : Boolean
+id() : Id
+operation(parms) : result
Traits
Abs
...
+remote
...
Client
return stub−>operation(args);
Stub<Abs, Traits<Abs>::remote>

Figure 4.34: EPOS framework: the `Handle` element.

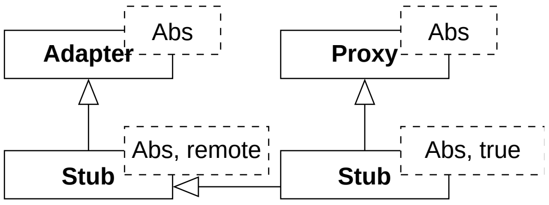

Figure 4.35: EPOS framework: the `Stub` element.

is enabled, `Handle` provides the extra constructors used to designate sharing in that scenario. The aggregation relationship between `Handle` and `Stub` enables the system to enforce allocation via a system allocator (instead of a programming language one), thus allowing for the `Allocated` scenario aspect[16].

The `Stub` framework element is depicted in figure 4.35. This parameterized class is responsible for bridging `Handle` either with the abstraction's scenario adapter or with its proxy. It declares two formal parameters: an abstraction and a boolean flag that designates whether the abstraction is local or remote to the address space of the calling process. By default, `Stub` inherits the abstraction's scenario adapter, but it has a specialization, namely `Stub<Abs, true>`, that inherits the abstraction's proxy. Therefore, making `Traits<Abstraction>::remote = false` causes `Handle` to take the scenario adapter as the `Stub`, while making it `true` causes `Handle` to take the proxy.

because it would make difficult the integration of EPOS with applications written in other programming languages.

[16]The allocator used with each abstraction is selected by `Adapter` after consulting `Traits<Abs>::allocated`.

The Proxy framework element is deployed when the remote scenario described in section 4.4.7 is established. Proxy realizes the interface of the abstraction it represents, forwarding method invocations to its Agent (see figure 4.36). Each instance of Proxy has a private ROI message, which is initialized in such a way that forthcoming invocations only need to push parameters into the message. Moreover, because Proxy is metaprogrammed, parameters are pushed directly into the message, without being pushed into the stack first. Proxy operations invoke method invoke[17] to perform a message exchange with Agent, which, likewise Handle for a local scenario, forwards invocations to the abstraction's Adapter.

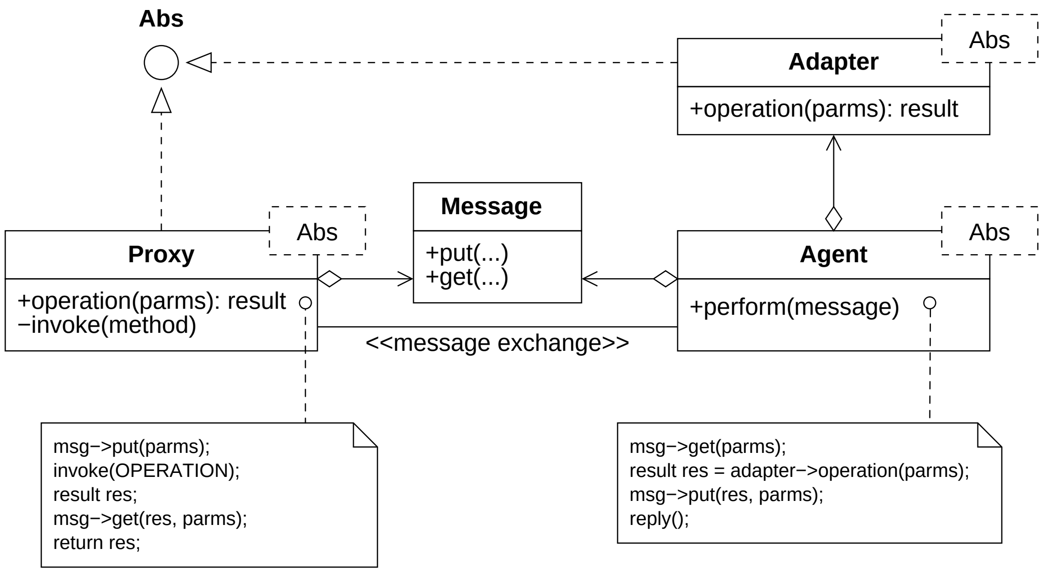

Figure 4.36: EPOS framework: Proxy and Agent elements.

The Adapter framework element is depicted in the figure 4.37. This parameterized class realizes a scenario adapter for the abstraction it takes as parameter, adapting its instances to perform in the selected scenario. Adaptations are carried out by wrapping the operations defined by the abstraction within the enter and leave scenario primitives, and also by enforcing a scenario-specific semantics for creating, sharing, and destroying its instances. The role of Adapter in EPOS framework is to apply the primitives supplied by Scenario to abstractions, without making assumptions about the scenario aspects represented in these primitives. In this way, Adapter is able to enforce any combination of scenario aspects.

The execution scenario for EPOS abstractions is ultimately shaped by the Scenario framework element depicted in figure 4.38. Each instance of parameterized class Scenario delivers scenario primitives that are specific to the abstraction supplied as param-

[17]The semantics of the invoke method varies according to the selected configuration. In some cases, it causes the application process to "trap" into the kernel, in others, it directly accesses a communicator to perform a message exchange.

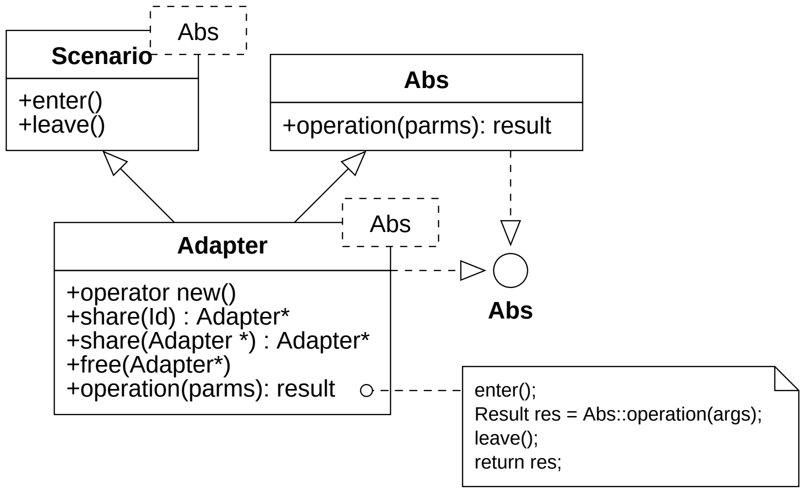

Figure 4.37: EPOS framework: the `Adapter` element.

eter. Firstly, it incorporates the selected `Id` aspect, which is common to all abstractions in a scenario; then it consults the abstraction's `Traits` to determine which aspects apply to it, aggregating the corresponding scenario aspects. The strategy to cancel an aggregation is similar to the one used with `Stub`, i.e. a parameterized class that inherits the selected aspect by default, but is specialized to inherit nothing in case the aspect is not selected for the abstraction.

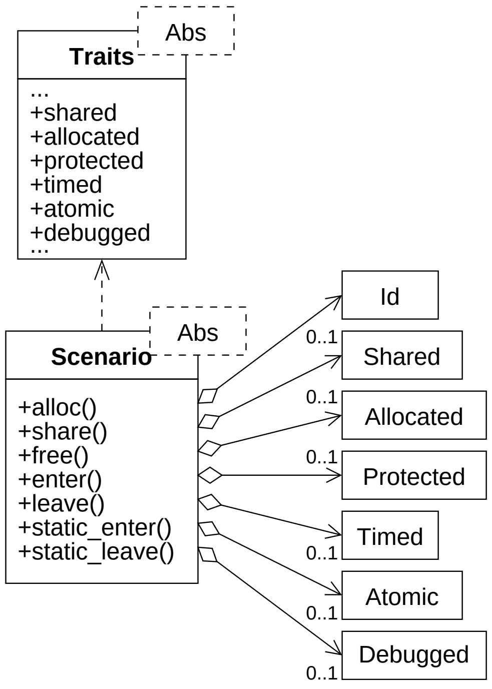

Figure 4.38: EPOS framework: the `Scenario` element.

Besides designating which scenario aspects apply to each abstraction, the parameterized class `Traits` maintains a comprehensive compile-time description of abstractions that is used by the metaprogram whenever an abstraction-specific element has to be configured.

4.5.1.2 Composition Rules

EPOS component framework metaprogram is able to adapt and assemble selected components to produce an application-oriented operating system. However, though the metaprogram knows about particular characteristics of each system abstraction from its *traits*, it does not know of relationships between abstractions and hence cannot guarantee the consistency of the composites it produces. In order to generate a meaningful instance of EPOS, the metaprogram must be invoked with a coherent parameter configuration.

Therefore, the operation of the framework metaprogram is coordinated by a set of *composition rules* that express elements of reusable system architecture captured during design. A consistent instance of EPOS comprises system abstractions, scenario aspects, hardware mediators, configurable features, and non-functional requirements. Composition rules specify dependencies and constraints on such elements, so that invalid configurations can be detected and rejected. Nevertheless, guarantying that a composite of EPOS elements is “correct” would depend on the formal specification and validation of each element, what is outside the scope of this research.

Sometimes, composition rules are implicitly expressed during the implementation of components. For example, by referring to the `Datagram` channel, the `Port` communicator implicitly specifies a dependency rule that requires `Datagram` to be included in the configuration whenever `Port` is deployed. However, most composition rules, especially those designating constraints on combining abstractions, can only be expressed externally. For instance, the rule that expresses the inability of the `Flat` address space to support the `Mutual` task abstraction must be explicitly written.

In order to support the external specification of composition rules, EPOS elements are tagged with a *configuration key*. When a key is asserted, the corresponding element is included in the configuration. Elements that are organized in families are selected by assigning a member’s key to the family’s key, causing the family’s inflated interface to be bound to the designated realization. This mechanism implements selective realize relationships [section 3.6] modeled during design. For example, writing `Synchronizer := Semaphore` causes the inflated interface of the `Synchronizer` family of abstractions to be bound to member `Semaphore` and writing `Id := Capability` binds the `Id` scenario aspect to `Capability`. Elements that do not belong to families have their keys asserted accordingly. For example, writing `Busy_Waiting := True` enables the `Busy_Waiting` configurable feature if the `CPU_Scheduler` abstraction.

Composition rules are thus defined associating pre- and postconditions to configuration keys. For instance, the following rule for the `Task` family of abstractions requires the `Paged` address space to be selected before the `Mutal` task can be selected:

```
Mutual ⇒ pre:  Address_Space = Paged
```

Alternatively, this constraint could be expressed as a composition rule for the `Address_Space` family that selects the `Exclusive` task whenever the `Flat` address space is selected:

```
Flat ⇒ pos:  Task := Exclusive
```

Composition rules are intended to be automatically processed by configuration tools. Hence, it is fundamental to keep them free of cycles. The following rule, though understandable for a human, could bring a tool to deadlock:

```
A1 ⇒ pre:  B = B1
B1 ⇒ pre:  A = A1
```

In order to ensure that EPOS composition rules build a direct acyclic graph, the following directives were observed:

- Configuration keys are *totally ordered* according to an arbitrary criterion;
- Preconditions are restricted to *expressions* involving only *preceding* keys;
- Postconditions are restricted to *assignments* involving only *succeeding* keys.

Along with the *traits* of each abstraction, composition rules control the process of tailoring EPOS to a particular application. The set of configuration keys selected by the user is validated and refined by means of composition rules, yielding a set of elements that are subsequently assembled by the framework metaprogram consulting the traits of abstractions.

4.5.2 Portability

The steady evolution of computing systems makes it easy to predict that many dedicated applications will experience more than a single hardware platform along their life cycles. From an application-oriented perspective, applications should transparently endure such hardware migrations, delegating portability issues to the run-time support system and the compilation environment. Therefore, EPOS visible elements were designed in such a way as to conserve syntax and semantics when ported to new hardware platforms. Elements may be unavailable in a platform because it does not feature the necessary hardware support, but those available behave accordingly in all platforms.

The extremely flexible architecture of EPOS requires portability issues to be dealt with consequently. If abstractions, scenario aspects, or component framework elements incorporate hardware idiosyncrasies, porting them to a new hardware platform could impair EPOS and consequently applications running on it. Nevertheless, as an operating system that aims at delivering a high-performance foundation to dedicated applications, EPOS has to ponder the implications that portability may have on performance [Lie96].

EPOS pursues the balance between portability and performance by means of two artifacts: a setup utility and a set of hardware mediators. The *setup utility* runs previous to the operating system to prepare the hardware platform to host a mostly portable system. As the utility builds an elementary execution context for EPOS, it initializes numerous hardware components, setting the system free from a main source of non-portability. The setup utility is itself highly dependent from the hardware platform for which it is implemented and does not aim at being portable.

Non-portable hardware interactions after the initialization phase are avoided in EPOS whenever possible. Sometimes, non-portable hardware mechanisms are replaced by portable ones in software, as long as resources are not compromised. For instance, the context switch operation available in some processors can usually be replaced by a software routine without impairments. Nevertheless, some configurations of EPOS cannot escape non-portable interactions with the hardware. The `Paged` address space abstraction, for instance, requires EPOS to interact with the MMU to create, destroy, and modify the address space of processes.

Architectural dependencies that cannot be handled by the setup utility are encapsulated in *hardware mediators*. When a system abstraction or a scenario aspect needs to interact with the hardware, it does it via a mediator, thus promoting portability. Hardware mediators, likewise the setup utility, are not portable; they are specifically designed and implemented for each platform.

4.5.2.1 The Setup Utility

EPOS *setup utility* is a non-portable tool that executes previous to the operating system to build an elementary execution context for it. The resources used by this utility are completely released before the first application process is created, so it can lessen on resource rationalization in benefit of an extensive setup procedure that carefully validates each configuration step.

The setup utility receives a `SysInfo` structure from the *bootstrap* that describes the relevant characteristics of the forthcoming EPOS configuration, so it knows which devices have to be activated and which elementary memory model has to be implemented. As the utility proceeds with hardware setup, it updates and completes `SysInfo`, including information about the physical resources configured, a memory map describing how the operating system has been loaded, the node's logical id, etc. This structure is later delivered to the operating system to assist the initialization of portable system components.

When the `Ahead_Allocated` scenario aspect is enabled, the setup utility pre-allocates indicated abstractions in the data segment of the operating system, relocating the respective pointers. For instance, if an FAST ETHERNET network adapter is marked to be allocated in advance, the setup initializes the device, maps it to a `Network` system object, and attaches it to the operating system's list of resources.

Differently from what one could imagine, the setup utility is seldom a large and complex software artifact. On the low-end of embedded systems, the setup may be relegated to load the operating system, since many of the microcontrollers used in such systems dispense further setup procedures. In contrast, the setup utility for the high-end workstations of a cluster can usually rely on a built-in monitor to configure the platform.

Besides promoting portability, the setup utility considerably reduces the complexity of other EPOS elements, for they no longer need to cope with hardware initialization. A void configuration of EPOS, i.e. a resulting system size of zero bytes, becomes possible in this scenario: some applications only need to be loaded on a pre-configured platform, dispensing with further operating system assistance. Such extreme cases are on the summit of application-orientation, demanding the highest degree of scalability from the operating system. They are only manageable in EPOS due to the division of operating system responsibilities with the setup utility.

4.5.2.2 Hardware Mediators

A *hardware mediator* abstracts elements of the hardware platform that are used by system abstractions and scenario aspects. However, it is not the intention of these mediators to building a "universal virtual machine" for EPOS, but hiding the peculiarities of some hardware components that are frequently used by the operating system. For example, the memory management unit and the interrupt controller. Mediators realize an operating system interface for these components, thus preventing architectural dependencies from

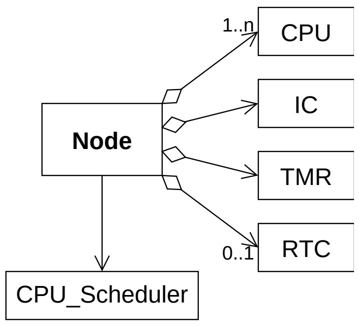

Figure 4.39: EPOS hardware mediator `Node`.

spreading over the system. Mediators are themselves hardware dependent, being sometimes coded in assembly, using macros, or static metaprogramming techniques.

The `Node` hardware mediator depicted in figure 4.39 yields a topmost abstraction for the computer in which EPOS executes, being its unique global object. The `Node` comprises a set of other hardware mediators, including one or more processors (`CPU`), an interrupt controller (`IC`), a timer (`TMR`), and a real-time clock (`RTC`). These hardware mediators are only included in a system configuration if the equivalent hardware components are available in the platform and if they were required by some system abstraction or scenario aspect selected by the user. Additionally, `Node` is associated with the `CPU_Scheduler`.

A special realization of the `Node` hardware mediator is deployed for symmetric multiprocessor nodes[18] that is able to cope with the scheduling and synchronization singularities of a parallel environment. This version of `Node` relies on the `Atomic` scenario aspect [section 4.4.6] to coordinate parallel system invocations. Whether processor affinity is considered for processor scheduling hinges on the corresponding configurable feature of `CPU_Scheduler` [section 4.3.2.3].

The `CPU` mediator depicted in figure 4.40 aggregates three other mediators: the floating point unit (`FPU`), the memory management unit (`MMU`), and the time-stamp counter (`TSC`). As explained earlier, these mediators are not aimed at simulating the hardware components they represent, but to implement an operating system interface for them. For instance, the `MMU` mediator is invoked to map physical memory to a paged address space, allocating and plugging the necessary frames to the designated page table.

Besides acting as a hardware mediator, `CPU` implements operations to save and restore the context of processes, and to perform the bus-locked read-and-write transactions (*Test and Set Lock*) required by the `Synchronizer` family of abstractions. It also holds a reference to the thread currently running on the processor.

Together with the setup utility, hardware mediators enable most of the abstractions

[18]Asymmetric multiprocessing is handled in EPOS with the ordinary `Node` mediator representing the "main" processor and secondary processors being represented as devices.

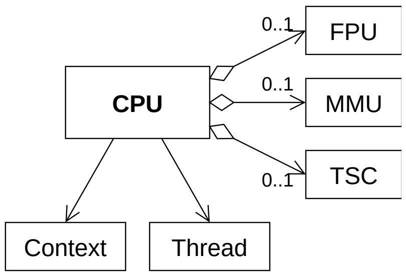

Figure 4.40: EPOS hardware mediator `CPU`.

and scenario aspects described earlier in this chapter to be transparently ported to new hardware platforms. Only some abstractions that directly concern hardware elements, such as `Bus` and some members of the `Device` family, are subject to portability pitfalls.

4.5.3 Initialization

EPOS splits the initialization of the computing system it manages in two phases. The first, which was described in the previous section, concerns the initialization of the hardware platform, while the second concerns the initialization of system data structures and the creation of the first (and possibly unique) application process. Both procedures have been designed aiming at supporting the flexible system architecture delivered by EPOS component framework.

The procedure of system initialization uses many algorithms and temporary data structures that are never used again during ordinary operation. The architecture of EPOS enables the resources allocated during this procedure, in particular memory, to be returned to the pool of free resources, so they can be reused later by applications. The first stage of EPOS initialization, the setup utility, completely erases itself when ready with its tasks, releasing all resources it had allocated. The second stage, the init utility, is activated as the last activity of setup, receiving the update `Sys_Info` structure created by the *bootstrap*.

4.5.3.1 The Init Utility

EPOS *init utility* is not a process, nor is it part of the operating system. It is just a routine that has plain access to the address space of the operating system, thus being able to invoke system operations. The initialization procedure carried out by the init utility consists in checking the traits of each abstraction to determine whether it has been included in the current system configuration[19], invoking the `init` class method for present abstractions. Abstractions and hardware mediators are requested to define this class method,

[19]Traits are compile-time structures, so consulting the traits of an abstraction that has not been selected for a system configuration does not alter the configuration.

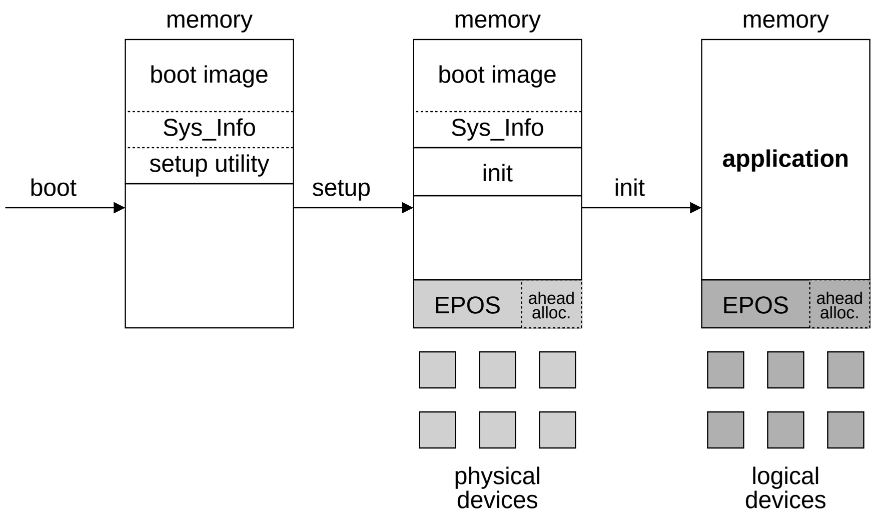

Figure 4.41: An overview of EPOS initialization.

which is responsible for initializing associated operating system structures. Besides having access to the traits of the abstraction it initializes, class method `init` also receives the `Sys_Info` structure as parameter, so it can consult the system description left by the setup utility. Furthermore, these class methods undergo a special link-editing process that causes them to be linked to the init utility rather than the operating system.

After having invoked the `init` class method for all present abstractions, the init utility invokes EPOS operations, which by now are fully operational, to create the first process. If the dedicated application running on EPOS is executed by a single process (per node), then the process created by the init utility is the application's unique process. Otherwise, this process is a *loader* that subsequently creates application processes in a multitasking environment. Before finishing, init releases all resources it had allocated, leaving EPOS alone with application processes.

An overview of EPOS initialization is depicted in figure 4.41. After loading the boot image, which includes a preliminary system description (`Sys_Info`), the bootstrap invokes the setup utility to configure the hardware platform. Considering the specifications in `Sys_Info`, the setup utility builds an elementary memory model, configures required devices, loads EPOS, pre-allocates `Ahead_Allocated` abstractions, loads the init utility, and activates it. The init utility, in turn, invokes the `init` class method of every abstraction included in the system to initialize its logical structure. It finishes loading the executable provided in the boot image to create the first process.

4.5.3.2 System Organization

The initialization procedure described above causes EPOS to assume one of the organizations depicted in figure 4.42. If the *embedded-in-the-application* organization is to be accomplished, the setup utility arranges for a single address space that is shared by the operating system and the application process. Hence, this organization implies the single-tasking environment realized by the `Exclusive` member of the `Task` family.

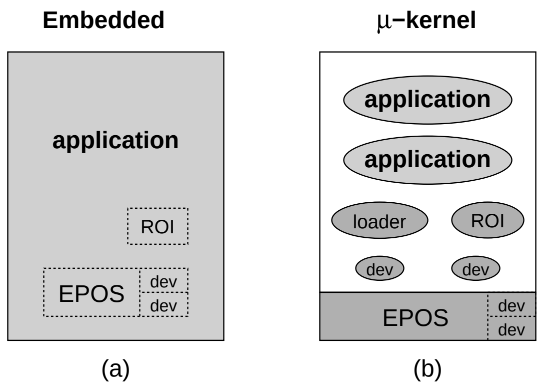

Figure 4.42: EPOS organizations: (a) embedded-in-the-application and (b) μ-kernel.

Though sharing the same address space, application and operating system are not linked together when the embedded-in-the-application organization is selected. Instead, the application is supplied a copy of the operating system's symbol table during linkage, thus enabling it to invoke system operations via ordinary procedure calls[20]. This design decision enables the setup utility to adopt an operating system loading procedure that is common to both organizations, embedded-in-the-application and μ-kernel.

Furthermore, avoiding linking operating system and applications together enables the portion of the address space in which the operating system resides to be protected against misleading applications. This is useful, even in single-tasking environments, to assist debugging applications. If the hardware platform permits, EPOS code segment is marked *execute-only*, while its data segment is marked accessible only from the code segment. Access violations regarding this scheme are caught and can be monitored activating the `Debugged` scenario aspect.

If EPOS is configured as a μ-kernel, the setup utility prepares two address spaces: one for the kernel and one for the first process. In this case, the init utility is temporarily loaded in the address space of the kernel. With the μ-kernel organization, EPOS features a system-call interface that is used by applications to interact with system abstractions. This interface implies in the `Remote` scenario aspect being enabled. The first process in this organization is the application loader, which additionally starts the ROI server if the `Global_Id` scenario aspect is enabled. Shared devices are delivered through device servers, which are also created by the loader. Since the loader is an ordinary user-level process, no modifications in the initialization procedure are necessary to accomplish this model.

[20] EPOS component framework metaprogram is processed in the context of applications, therefore a fraction of the operating system will always be embedded in the application. Moreover, some elementary system operations may completely embed in the application (through inlining).

4.6 Automatic Configuration

EPOS application-oriented system design enables it to be tailored to match the requirements of particular applications. By specifying configuration keys, users select the system abstractions, scenario adapters, and hardware mediators that are necessary to support a given application. Composition rules facilitate the task of tailoring EPOS, allowing users to specify a reduced number of configuration keys that are expanded in a coherent system configuration. Moreover, a user-friendly visual tool can be deployed to carry out this manual configuration process.

However, EPOS comprises hundreds of configurable elements. Delivering application programmers such a massive repository of components and a mechanism to select and plug them in a component framework may be inadequate. Users can miss the most appropriate system configuration for a given application simply because there are excessive alternatives. Even if composition rules allow them to specify a reduced number of configurable elements, it is possible that significant elements will be relegated to inadequate default configurations. Fortunately, application-oriented system design makes it possible to automate the system configuration process to reduce the occurrence of such incidents.

For a component-based operating system like EPOS, accomplishing an automated "system factory" is mainly a matter of understanding application requirements. The way EPOS associates configuration keys to abstraction interfaces allows application programmers to specify application requirements about the operation system simply by writing down system invocations. If programmers are not sure about which member of a family of abstractions should be used in a situation, or if they realize the selection may need to be changed in the future, they can use the family's inflated interface instead. Afterwards, the application can be submitted to a pipeline of tools that will tailor EPOS to fit it.

A schematic representation of the process of automatic configuring EPOS is presented in figure 4.43. The first step consists in performing a syntactical analysis of the application's source code to identify *which* system abstractions have been invoked by the application and *how* they have been invoked. This step is carried out by the *analyzer*, which generates a preliminary configuration blueprint consisting of family's (inflated) and member's interfaces.

The preliminary configuration blueprint produced by the analyzer is subsequently passed to the *configurator*. This tool consults the catalog of composition rules and the list of abstractions to refine the configuration blueprint. During this refinement, some inflated interfaces are bound to satisfy dependencies and constraints. Those that remain unbound are subsequently bound to their "lightest" realizations. The output of this tool is a table of configuration keys that shapes a framework for the desired EPOS configuration.

In order to support the proposed automatic configuration scheme, EPOS abstractions and scenario aspects have been sorted in a cost model corresponding to their intrinsic *overhead*. This ordering is revealed by the sequence in which abstractions and scenario aspects have been introduced in their families along this chapter, i.e. the `Exclusive`

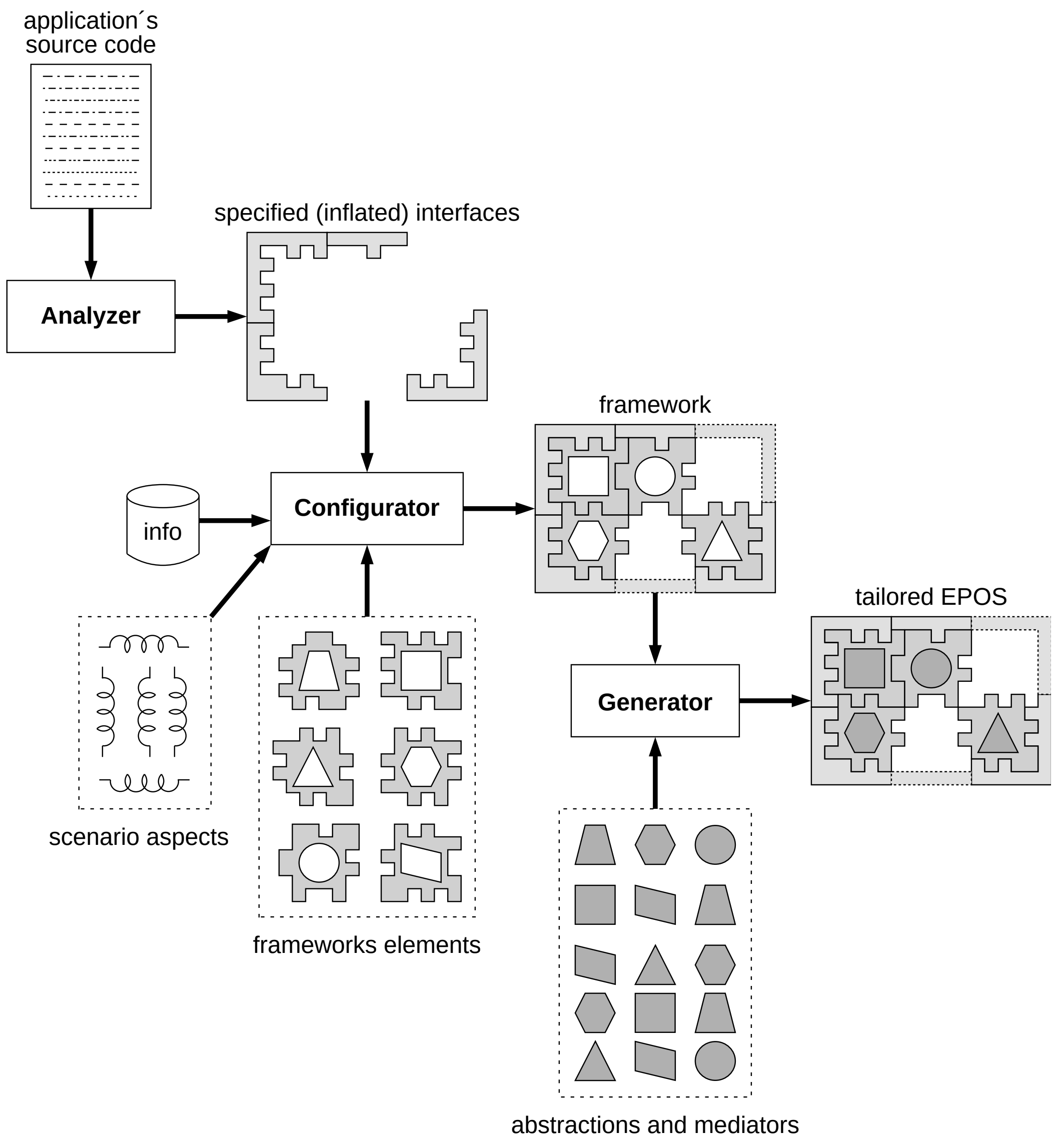

Figure 4.43: EPOS automatic configuration tools.

member of the `Thread` family has a lower estimated cost than member `Cooperative`, and the `Local` member of the `Id` family has a lower estimated cost than member `Global`.

The last step in the automatic configuration process is performed by the *generator*. This tool uses the set of configuration keys produced by the configurator to compile an application-oriented version of EPOS. Such EPOS instances include only those elements that are necessary to support the execution of associated applications. Moreover, abstractions designated via inflated interfaces are selected as to minimize the operating system overhead on resource management. An implementation of these configuration tools as a compiler front-end will be discussed in section 5.3.3. Such implementation allows EPOS to be transparently configured as the application is compiled.

Nevertheless, EPOS "operating system factory" cannot guarantee the tailored system to be optimal, even if the application completely delegates configuration to the presented tools by interacting with the system exclusively via inflated interfaces. The inaccuracy of the automatic configuration process arises from the cost criterion used to order abstractions in each family: overhead. The overhead criterion properly represents the cost of most abstractions, but in some cases, leads to non-optimal configurations. The processor scheduling policy, for instance, can be wrongly estimated under this criterion. Sometimes the higher inherent overhead of a scheduling policy is compensated by producing a sequence of scheduling events that matches the application's execution flow.

Therefore, the set of automatic configuration tools was designed considering interrup-

```
#include <iostream>
#include <synchronizer.h>
#include <thread.h>

using namespace System;
using namespace std;

Synchronizer fork[5];

int philosopher(int n)
{
    int first  = (n < 4)? n : 0;
    int second = (n < 4)? n + 1 : 4;
    for (;;) {
        cout « "Philosopher␣" « n « "␣thinking ␣...\n";
        fork[ first ].lock();     // get first fork
        fork[second].lock();      // get second fork
        cout « "Philosopher␣" « n « "␣eating ␣...\n";
        fork[ first ].unlock();   // release first fork
        fork[second].unlock();    // release second fork
    }
}

int main()
{
    Thread* phil[5];
    for(int i = 0; i < 5; i++)
        phil[i] = new Thread(&philosopher, i);

    for (;;);
}
```

Figure 4.44: The dinning philosophers problem in EPOS.

tions after each phase, so users can invoke the manual configuration tool to check and modify the automatically generated configuration before it is submitted to the generator.

For an example of automatic system configuration, the program in figure 4.44 will be considered. This is a valid EPOS implementation of the classic *dining philosophers* problem [Dij71]. It requires three system abstractions: `Thread`, `Synchronizer`, and `Console`. The *analyzer* would identify these families, as well as the scope in which they were used, and output a table with the signatures of used operations. Signatures are known by the analyzer from the interfaces supplied in the system header files included. The syntactical analysis of this program, ignoring the complex `iostream` header, would look like the following:

```
global {
    System::Interface :: Synchronizer::Synchronizer(void);
    System::Interface :: Synchronizer::lock(void);
    System::Interface :: Synchronizer::unlock(void);
}

main {
    System::Interface :: Thread::Thread(int (*)(int ),  int)
}
```

The analysis report reveals that objects of type `Synchronizer` have been created in the global scope using the default constructor, and that operations `lock` and `unlock` have been invoked on them. It also reveals that objects of type `Thread` have been created in the scope of function `main` using a constructor that takes a pointer to a function and an integer as parameter.

The *configurator* would process this information considering the catalog of composition rules and the list of system abstractions. This would lead the `Synchronizer` inflated interface to be bound to `Mutex`, for it supplies the required operations. Even if `Semaphore` had declared a default constructor and operations `lock` and `unlock` as synonyms for `p` and `v`, thus emulating a `Mutex`, the cost ordering would have made `Mutex` the best choice.

The selection of a realization for the `Thread` inflated interface would be based on a special composition rule that requires method `pass`, which hands processor control over another thread, to be explicitly invoked in order that realization `Cooperative` is selected. Since the `Exclusive` realization of `Thread` does not feature thread creation, member `Concurrent` would be selected.

However, the `Concurrent` thread implies in the `CPU_Scheduler` abstraction, which in this case would adopt `FCFS` as the scheduling policy, for it incurs in the lowest overhead. Nonetheless, this decision would only be adequate if the application is executed in a node with five or more processors. Otherwise, some philosophers would never execute. This reveals the limitations of EPOS automatic configuration. Though the number of processors on each node is available to configuration tools, the decision of bypassing the cost model to select a different scheduling policy cannot be taken automatically. If less than five processors are available in the indicated node, the user would either have to manually correct the configuration, or modify the ordering of scheduling policies in the cost model to define another default.

4.7 Summary

EPOS (*Embedded Parallel Operating System*) is an experimental application-oriented operating system developed in the scope of this dissertation to validate concepts and techniques of application-oriented system design. The domain envisioned by EPOS is that of high-performance dedicated computing, which comprises applications that, besides running with exclusivity on the respective platforms, require an efficient management of resources. This domain comprises mainly embedded and parallel applications.

In order to cope with the steady evolution of the envisioned domain, EPOS established an open and continuous domain analysis process that allows new entities to be included in the design as they are identified in the domain. Domain entities were modeled aiming at high scalability and reusability, so that EPOS can be tailored to particular applications. Abstractions are mostly independent of each other, of execution scenario aspects, and of component frameworks. Consequently, they can be extensively reused in a variety of scenarios. Furthermore, EPOS component framework can be adjusted to accommodate forthcoming abstractions, or to build particular software architectures, without affecting existing abstractions.

EPOS captures the entities in the domain of high-performance dedicated computing with application-oriented abstractions that realize mechanisms concerning the management of memory, processes, coordination, communication, time, and I/O. Besides, a remote object invocation mechanism allows for external abstractions, such as files and graphical display, to be transparently accessed. Entities with strong architectural dependency, such as node and CPU, were separately modeled as non-portable hardware mediators that realize a portable operating system interface.

EPOS models the following properties of domain entities as scenario aspects: identification, sharing, allocation, protection, timing, atomicity, remote invocation, debugging, and profiling. An execution scenario for abstractions is shaped by combining suitable scenario aspects, which are transparently applied to abstractions via scenario adapters.

EPOS component framework captures elements of reusable system architectures, defining how abstractions can be composed. It was modeled as a collection of interrelated scenario adapters that build a “socket board” for system abstractions and scenario aspects, which are “plugged” via inflated interface binding. The framework is realized by a static metaprogram and a set of externally defined composition rules. The metaprogram is responsible for adapting abstractions to the selected execution scenario and arranging them together during the compilation of an application-oriented instance of EPOS, while the rules coordinate the operation of the metaprogram, specifying constraints and dependencies for the composition of abstractions.

The scalable software architecture accomplished by the component framework is sustained by the *setup* and *init* utilities. The former isolates architectural dependencies concerning hardware initialization, while the latter isolates system initialization procedures. In cooperation with the framework metaprogram, these utilities are able to deliver an EPOS instance either as a μ-kernel or fully embed it in a single-tasked application.

EPOS design allows for automatic configuration and generation. Abstraction and family (inflated) interfaces referred to by an application can be collected by EPOS *analyzer* to yield a blueprint for the system that has to be generated. This blueprint is subsequently processed by EPOS *configurator*, which consults a catalog of composition rules and a list of existing system abstractions to configure the component framework according to application needs. At last, EPOS *generator* compiles an instance of EPOS including the necessary abstractions, mediators, and scenario aspects.

Chapter 5

EPOS Implementation for the SNOW Cluster

The EPOS system described in chapter 4 illustrates the deployment of application-oriented system design to engineer the domain of high-performance dedicated computing. That case study helped to corroborate the concepts and techniques of application-oriented system design proposed in chapter 3. However, if EPOS is to be accepted as a validation of application-oriented system design, its design must also be validated. Although many of EPOS design decisions have been well substantiated, a design can hardly be positively evaluated before it is implemented to a significant extent. Therefore, a prototype implementation of EPOS for the SNOW cluster was carried out in the scope of this dissertation.

5.1 Why a Cluster of Workstations?

The idea of clustering commodity workstations as a cost-effective alternative to expensive *Massively Parallel Processors* (MPP) has now been explored for quite a long time. Perhaps one of the first approaches has been the one in which the idle time of ordinary workstations interconnected in a *Local Area Network* (LAN) was collected to form a computational resource pool that could be used to support the execution of parallel applications [HCG$^+$82, Che84]. This idea has been around for almost as long as the LAN itself and undoubtedly helped to open the way for cluster computing. Nowadays, however, cluster computing is better associated with a pile of workstations interconnected in a *System Area Network* (SAN) and completely dedicated to support the execution of parallel applications. During this natural evolution towards performance, cluster computing has reshaped the concept of commodity computing by triggering the migration of software and hardware concepts from the supercomputer environment to the desktop.

Undoubtedly, cost-effectiveness has been a key factor to boost the utilization of clusters of workstations to support high-performance computing. However, price/performance is not a metric when the problem to be solved, i.e. the parallel application to be executed, demands performance figures that cannot be supplied by a cost-effective cluster. That these FLOP famished applications are there, no one questions, but the suggestion that an MPP is the only way to appease them, is increasingly controversial. Recent developments suggest that clusters are going, in the short term, to overcome MPPs also in absolute performance [TFC00, CPL$^+$97]. This optimism is not accidental—it arises from simple market laws that favor large-scale production. Improving performance of computational systems demands large efforts on engineering and manufacturing, which are usually achieved at a very high cost. While MPPs have to share these costs among a few produced units, clusters can share them with the huge workstation market. This phenomenon has been slowing the development of custom hardware, in favor of commodity components, in such a way as to indicate that both technologies, MPPs and clusters, are about to merge. The Intel ASCI Red MPP [San01] is a good example for this merge: its processing elements are ordinary Intel Pentium Pro microprocessors normally found in PCs, and the whole machine has only two non-commodity components in the interconnection system.

Although the commodity workstation hardware scene looks bright, the same cannot be stated about commodity software. The run-time support systems running on ordinary workstations have not been designed with parallel computing in mind and usually fail to deliver the functionality needed to execute a parallel application. The commodity solution to this problem is to add a middleware layer, so that features like location transparency, remote invocation, and synchronization are appended to the system. However, this configuration often fails to achieve the performance level demanded by the parallel application [NAS97, MT00].

Indeed, the need for specialized run-time support systems in order to support parallel computing on clusters of workstations has already been recognized by the fraction of the cluster computing community compromised with high-performance [ABLL92, Fel92, BRB98]. In this scene, modifications in the operating system kernel, customizations in run-time support libraries, and specialized middleware implementations are common-practice, with user-level communication being a frequently explored optimization. This can be observed in projects like U-NET at Cornell University [WBvE96], FAST MESSAGES (FM) at the University of Illinois [PKC97], PM at the Real World Computing Partnership [THIS97], BASIC INTERFACE FOR PARALLELISM (BIP) at the University of Lyon [PT98], and others.

However, run-time support system specializations for cluster computing are usually not customizable with regard to application requirements, focusing on improving performance by exploring particular architectural features. Especially regarding communication, most solutions comprise a single protocol, disregarding the particular characteristics of each application. In contrast, EPOS was designed to be customized according to application needs, delivering abstractions that can be adjusted and combined to produce a large variety of run-time systems. For example, EPOS communication subsystem [section 4.3.4] comprises six families of abstractions and a number of configurable features that can be arranged to deliver applications a tailored communication system.

5.2 The SNOW Cluster

The SNOW cluster was assembled in 1997 at the Research Institute for Computer Architecture and Software Engineering (FIRST) of the German National Research Center for Information Technology (GMD). The cluster consists of 24 processing nodes interconnected with a high-speed network and connected to a server through a service network. These are all off-the-shelf components that have been separately acquired and locally assembled by the institute staff.

A sketch of the SNOW cluster from the perspective of EPOS is presented in figure 5.1. The server acts as a front-end to the cluster, from which parallel applications are started and monitored. While running EPOS, processing nodes are entirely managed from the server, which triggers a remote boot procedure to start up a tailored EPOS for each application session. The server also provides parallel applications running on processing nodes with a persistent storage facility.

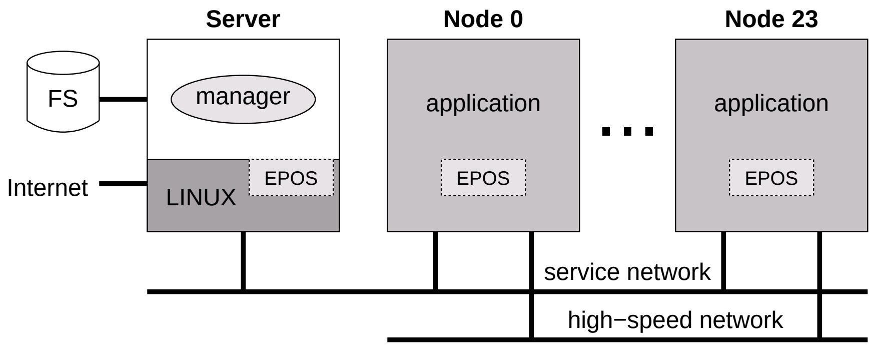

Figure 5.1: The SNOW cluster from the perspective of EPOS.

5.2.1 Processing Nodes

SNOW processing nodes are diskless workstations, each comprising one or two Intel P6 processors, main memory, and network interface cards (NIC) for the computing (MYRINET) and service (FAST ETHERNET) networks. A schematic of a SNOW node focusing on hardware components that are significant to EPOS is shown in figure 5.2.

Every hardware platform has its highs and lows. When used for a purpose other than the originally conceived, it is natural that lows become more noticeable. Commodity workstations are not constructed to support parallel computing and, when used for this purpose, often fail to exhibit the properties anticipated by applications. Before beginning

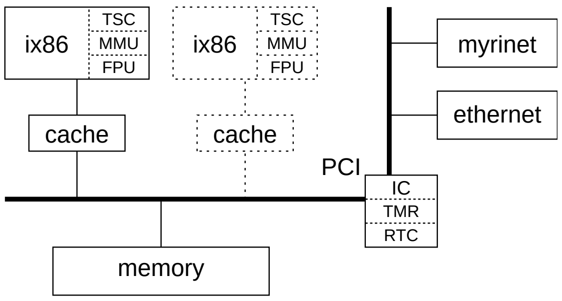

Figure 5.2: A SNOW node from the perspective of EPOS.

the implementation of EPOS for the SNOW cluster, a series of experiments has been performed in order to identify hardware limitations concerning high-performance computing that could be reduced by proper operating system design. These experiments identified two major limitations:

1. Low memory bandwidth: SNOW nodes, like most ix86-based workstations, are equipped with low bandwidth memory subsystems. This deficiency is usually not perceptible to the interactive applications that are traditionally executed on such platforms thanks to a high-bandwidth cache mechanism that hides main memory latency. Nevertheless, a parallel application performing short computation cycles on large data sets can easily overflow such mechanism, leading the processor to wait for the memory subsystem.

2. Interconnect on I/O bus: in contrast to MPP nodes, which usually integrate interconnect mechanisms to the processor/memory path, clusters of commodity workstations rely on ordinary network interface cards plugged in the I/O bus (figure 5.2). As consequence, the path to the network is lengthened and the overhead on inter-node communication increased. In the particular case of SNOW, the data transfer rate between main memory and the MYRINET adapter is lower than between two MYRINET NICs on different nodes.

There is not much an operating system can do to overcome the first limitation. Caring for cache-aligned memory *allocation* could help, but the operating system cannot enforce cache-aligned memory *access* (a compiler could). Regarding the second limitation, however, the experiments conducted with SNOW helped to design a pipelined communication subsystem that is able to hide a significant part of the latency induced by the hardware architecture. This mechanism will be discussed later in section 5.3.1.

5.2.2 Interconnects

SNOW processing nodes are interconnected by two networks: a service network used for interactions with the server, and a computing network used for message exchange between processes of parallel applications. The organization of these networks is depicted in figure 5.3. The *service network* consists of a full-switched, full-duplex FAST ETHERNET network with a GIGA ETHERNET link to the server. It was configured to give each node an equally wide communication channel with the server.

The *computing network* consists of a Myricon MYRINET high-speed network [BCF+95]. MYRINET has been chosen as the computing network for SNOW because its interfaces and protocols are open and published, and its network interface cards are programmable. Indeed, the MYRINET NICs that equip SNOW nodes (figure 5.4) are embedded computing system, with own processor and memory. The LANAI processor on MYRINET NICs can be programmed to release the main processor from most communication-related tasks.

Two architectural characteristics of MYRINET NICs are of special interest for an operating system project:

- The memory on the NIC can be shared with the main processor, yielding an asymmetric multiprocessor system;
- The DMA engines used to send and receive data to/from the network and to transfer data between host and NIC can operate in parallel.

The asymmetric multiprocessor configuration enables the network to be modeled as a *bounded buffer,* with application processes on the host *producing* outgoing messages that are *consumed* by a process on the MYRINET NIC. Incoming messages are handled the other way round, with a MYRINET process playing the role of producer and application processes on the host playing consumers. This asynchronous communication model, in combination with the parallel operation of DMA engines, allows a message (or a piece of

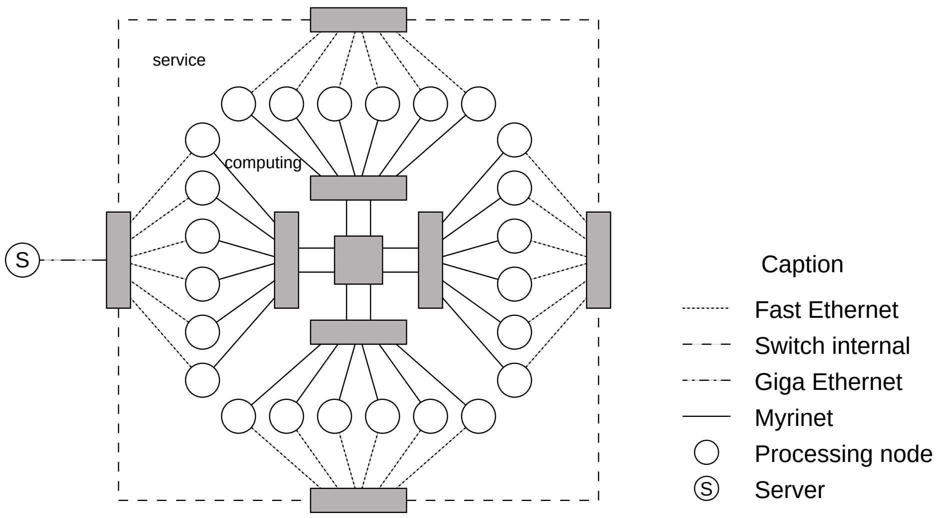

Figure 5.3: The organization of SNOW networks.

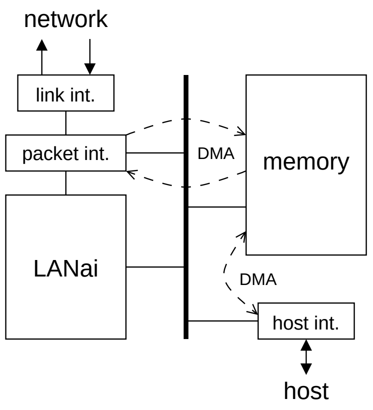

Figure 5.4: The MYRINET network interface card from the perspective of EPOS.

a message) to be transferred between host and MYRINET at the same time another one is being transferred over the network, thus sustaining a communication pipeline

5.3 EPOS Implementation

Aiming at validating the domain engineering described in chapter 4, a prototype of EPOS was implemented for the SNOW cluster. Emphasized were the software engineering techniques introduced by application-oriented system design, such as adaptable abstractions, scenario aspects, inflated interfaces, and component frameworks. Classic operating system concepts, which have already been implemented in numerous other systems, were given a lower priority. In particular, no support for multiprocessor nodes has been included in this prototype.

The implementation of EPOS prototype for SNOW was conducted as a last refinement of design, rather than an isolated phase with sporadic meeting points. Constant feedback helped to enhance design specifications while yielding better support for implementation decisions. This scheme, which reassembles *eXtreme Programming* (XP) [Bec99], was only feasible because of the solid domain decomposition guided by application-oriented system design. Cycling design to extend or correct the functional specification of individual abstractions or include new abstractions does not have major consequences on already implemented entities. In fact, conducting a design to its finest level independently of implementation experiments is seldom viable. However, a sloppy domain decomposition would have been exposed to deeper modifications, perhaps requiring abstractions and scenario aspects to be repartitioned, thus compromising the software development process.

EPOS prototype for SNOW was mostly implemented in standard C++, with some few hardware mediator methods written in assembly. Two versions of the prototype have been produced: one runs “natively” on SNOW, and the other as a “guest operating system” on

LINUX. From the point of view of applications, both versions are functionally equivalent, though the LINUX-guest implementation suffers performance impairments due to LINUX memory and process management strategies. Specifically, there is no way to disable multitasking and scheduling on UNIX-like systems, consequently affecting the execution of dedicated applications that do not need such features.

Several of the EPOS abstractions described in section 4.3 have been implemented for SNOW, including representatives of all modeled families. Particularly interesting in the context of application-orientation, were those minimalist abstractions unavailable in general-purpose operating systems, such as the `Flat_Address_Space`, the `Exclusive_Task`, and the `Exclusive_Thread`.

The `Flat_Address_Space` has a practically empty implementation, since the configuration of the ix86's MMU to simulate a flat address space is performed by the setup utility[1]. The abstraction only features operations to assert the status of the memory subsystem and to modifying memory-mapping modes (through the methods of the `MMU` hardware mediator). The `Exclusive_Task` is also mostly realized by the setup utility, which loads the task's code segment in a ready-only portion of the address space, assigning the remaining memory to the task's data segment.

The `Exclusive_Thread` implementation consists basically of status and termination operations. If the `Alarm` abstraction is not explicitly instantiated by the application, timer interrupts remain disabled, since processor scheduling is not necessary for this abstraction[2]. Together with `Flat_Address_Space` and `Exclusive_Task`, `Exclusive_Thread` is able to produce an EPOS instance of size zero: after hardware and system initialization, absolutely all resources are delivered to the application process.

The decomposition guided by application-oriented system design resulted in relatively simple implementations even for EPOS most complex abstractions. For instance, a large number of entities pertaining the implementation of the `Paged_Address_Space`, `Mutual_Task`, and `Concurrent_Thread` abstractions have been separately implemented as hardware mediators and scenario aspects. Implementing the `Paged_Address_Space` abstraction without having to consider the idiosyncrasies of ix86's MMU (and vice-versa) was undoubtedly more effective than implementing a monolithic abstraction.

EPOS scenario aspects [section 4.4] have been modeled relying on the ability of *scenario adapters* to autonomously enforce a *scenario* (i.e. a composition of structural and behavioral scenario aspects) to abstractions, thus eliminating eventual dependencies from the component framework. Consequently, scenario aspects could be implemented as autonomous objects that are aggregated to form a `Scenario` object. This is subsequently applied to abstractions by the `Adapter` framework element (see figure 4.33 in section 4.5.1).

[1]Completely disabling the MMU of a ix86 processor is not possible, so the setup utility has to arrange for a mapping in which logical and physical addresses match.

[2]Time operations performed with `Clock` and `Chronometer` do not depend from timer interrupts.

Accomplishing an implementation of EPOS component framework metaprogram [section 4.5.1] that respect the autonomy of system abstractions and scenario aspects, i.e. that does not require them to incorporate specific framework elements, was the most time-demanding activity during the implementation of EPOS prototype. A precise realization of the design described in section 4.5.1 as a set of C++ templates could only be accomplished after a long streamline process. This process exposed many pitfalls of static metaprogramming in C++, especially concerning maintainability[3]. Achieving the expected functionality was not problematic, but the first versions of the metaprogram were almost unintelligible.

Nevertheless, the resultant framework metaprogram, besides being able to handle independently defined abstractions and scenario aspects, has a null intrinsic overhead and is relatively easy to maintain.

5.3.1 The Myrinet Network Abstraction

A distinguished abstraction implemented for the SNOW cluster is the `Myrinet` member of the `Network` family. This abstraction was firstly implemented for the guest version of EPOS focusing on pipelining the communication system [Tie99]. Subsequently, a native version emphasizing aspects of application-oriented system design was conducted [FTSP00].

A design diagram of the `Myrinet` abstraction is depicted in figure 5.5. `Myrinet` uses the `Myrinet_NIC` member of the `Device` family to realize the uniform inflated interface of the `Network` family. The `Myrinet_NIC` device abstraction deploys mechanisms provided by its family common package to map the control registers and the memory of a MYRINET NIC to the address space of the main processor. In this way, the state of a `Myrinet_NIC` instance matches the state of the corresponding physical NIC[4]. `Myrinet` subsequently uses `Myrinet_NIC` to realize a member of the `Network` family of abstractions, defining methods to send and receive packets of data to peer nodes on the MYRINET system area network.

A configurable feature is more than a flag to control the inclusion of an optional family feature; it also designates a generic implementation of the feature that can be reused by family members. With regard to MYRINET, the `ordering` configurable feature is permanently enabled, since the source routing strategy used by MYRINET implicitly ensures in-order delivery. However, the generic implementations of configurable features `multicast` and `broadcast` would be reused, since MYRINET does not support them by design.

Nevertheless, a MYRINET NIC has its own processor and could itself implement

[3]The framework metaprogram is not exposed to end users.

[4]The setup utility pre-allocates one instace of `Myrinet_NIC` for each MYRINET NIC installed in the node.

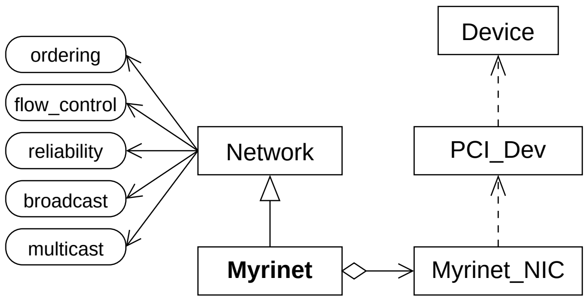

Figure 5.5: EPOS `Myrinet` abstraction.

the hardware support that is missing to deliver the configurable features defined for the `Network` family. Therefore, none of the generic implementations has been reused by `Myrinet`, which relies on processes running on the NIC to realize them. Indeed, most of the functionality of the `Myrinet` abstraction is delivered by such processes, which perform in an asymmetric multiprocessor scenario with processes on the host.

The communication strategy implemented by the `Myrinet` abstraction consists in writing *message descriptors* on the memory shared by NIC and host, designating the address and length of outgoing messages and incoming message buffers, and signalizing a process on the NIC to perform a message exchange. Two processes run on the NIC in behalf of `Myrinet`: *sender* and *receiver*.

The `Myrinet` *sender* process (see the activity diagram in figure 5.6) waits for a message descriptor to be signalized ready and then autonomously processes the outgoing message. It first starts a DMA transaction to fetch the message from main memory. Meanwhile, it generates a header for the outgoing message based on the information present in the descriptor. When the completion of the DMA transfer is signalized, a new DMA transaction is started to push the message, now with a header, into the network. Finally, the descriptor is updated to signalize that the message has been sent.

The `Myrinet` *receiver* process operates complementarily to *sender*. It is activated by interrupt whenever a message arrives[5], extracting it from the network with a DMA transaction and storing it in a local buffer. Subsequently, *receiver* checks for a message descriptor indicating that a process on the host is waiting for the message. If one is available, it initiates a second DMA transaction to move the message to the main memory location indicated by the descriptor, otherwise the *sender* process will perform this step later when a receive descriptor becomes available. On the eminence of buffer overflow, a flow control mechanism is activated if the `flow_control` configurable feature is enabled, otherwise messages are dropped.

This communication strategy was conceived considering the `Flat` member of the `Address_Space` family [section 4.3.1], for which logical and physical addresses do

[5]The LANAI processor on MYRINET NICs is a dual-context processor, with context exchange being automatically triggered by interrupts.

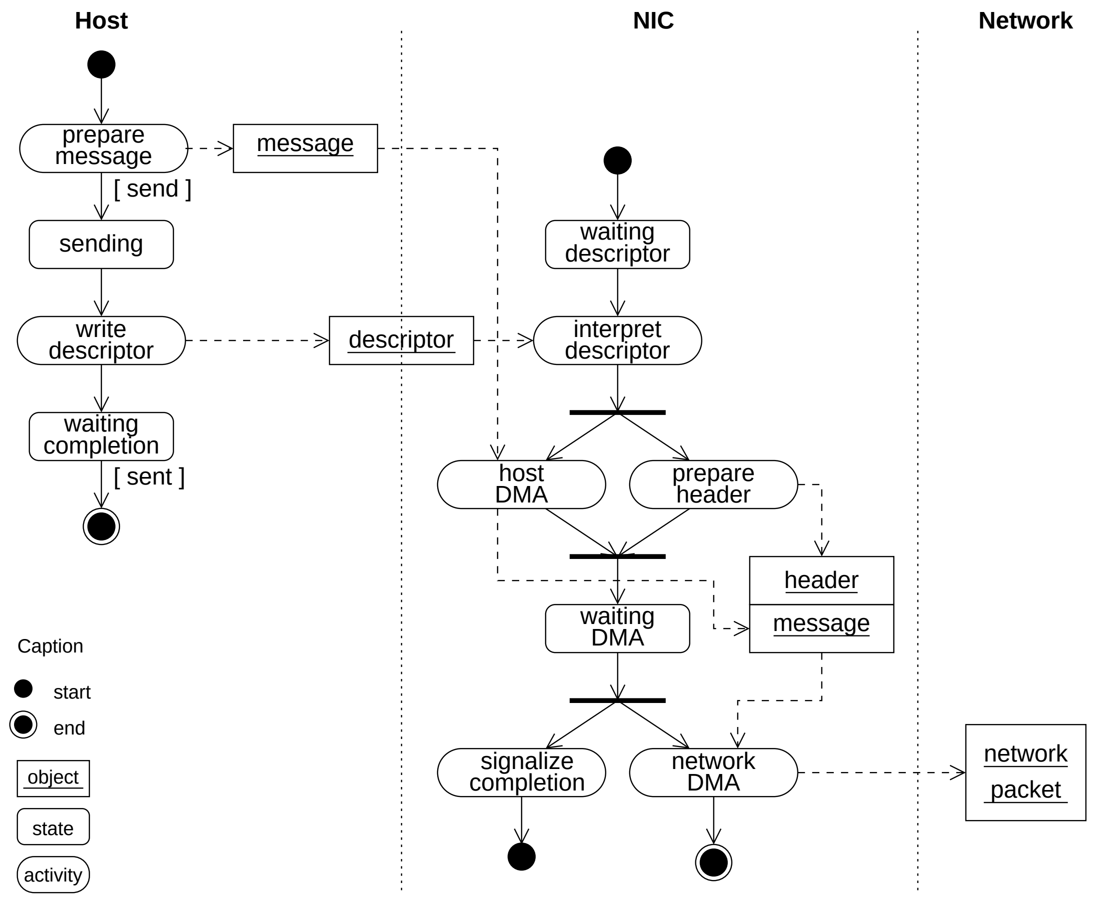

Figure 5.6: Activity diagram of EPOS `Myrinet` sender process.

match, since the LANAI processor on the MYRINET NIC does not know of the address translation scheme on the host. If the `Paged_Address_Space` is used, or if EPOS is running as guest on LINUX, messages can be scattered across the memory in frames that cannot be directly identified from their logical address. Therefore, an intermediate copy to a contiguously allocated system buffer, of which the physical address is known, has to be performed on the host. A message exchange over MYRINET, including the additional copies, is depicted in figure 5.7.

The ability of a MYRINET NIC to simultaneously perform two DMA transfers (each processor cycle comprises two memory cycles) was explored to transform the steps represented in figure 5.7 in stages of a *communication pipeline*. Messages are thus broken in small packets that move through the pipeline in parallel. In this way, it is possible to start sending a message over the network while it is still being fetched from main memory. When the pipeline is full, stages 2 and 3.1, as well as stages 4 and 3.2, are overlapped. The delay between stages 3.1 and 3.2, i.e. the physical network latency, is much smaller than the time spent by packets on other pipeline stages, hence both stages can be merged to yield a single stage 3. If the copy stages 1 and 5 are necessary, message descriptors

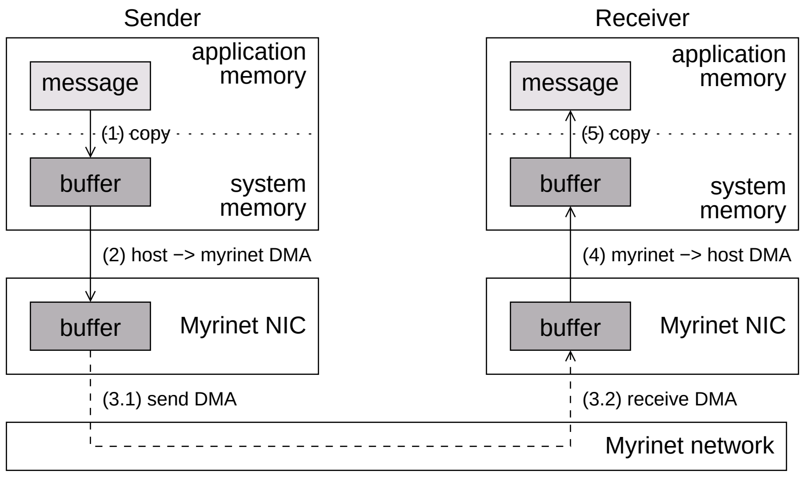

Figure 5.7: A message exchange with MYRINET.

are assigned on per-packet basis, so the processes on the MYRINET NIC are not forced to wait for the copies to be completed. However, stages 1 and 2, as well as stages 5 and 4, concur for the same physical resource, namely memory, and will seldom overlap.

Furthermore, the proposed communication pipeline has a critical parameter: the packet size. Adopting an excessively large packet size increases the pipeline's setup time, while excessively small packets exacerbate the pipeline's internal overhead. Furthermore, the influence of the packet size on the pipeline depends on the length of the messages being transmitted. In order to estimate the optimal packet size as a function of the message length, the bandwidth of each pipeline stage and the latency experienced by packets on each stage were obtained. Applying a linear cost model, the optimal packet size for different ranges of message lengths was calculated [Tie99]. The result was used to implement an adaptive communication pipeline that automatically switches to the best packet size for each message.

Although the communication pipeline has a low intrinsic overhead, programming DMA controllers and synchronizing pipeline stages may be more onerous than using programmed I/O for messages shorter than a certain length. Therefore, the pipeline is bypassed for messages shorter than a definable threshold (256 bytes by default in the current implementation).

The performance of the `Myrinet_Network` abstraction implemented for the SNOW cluster was assessed sending messages of increasing length from one node to another. Both the iX86-native (no copy) and the LINUX-guest (copy) versions were considered. Figures 5.8 and 5.9 show respectively the bandwidth and the latency observed during the experiment. A bandwidth of 116 Mbytes/s for 64 Kbytes messages represents roughly 90% of the bandwidth available to transfer data between main memory and the MYRINET NIC over the 32 bits/33 MHz PCI bus that equips SNOW nodes.

Figure 5.8 makes evident the contention for memory access between pipeline stages

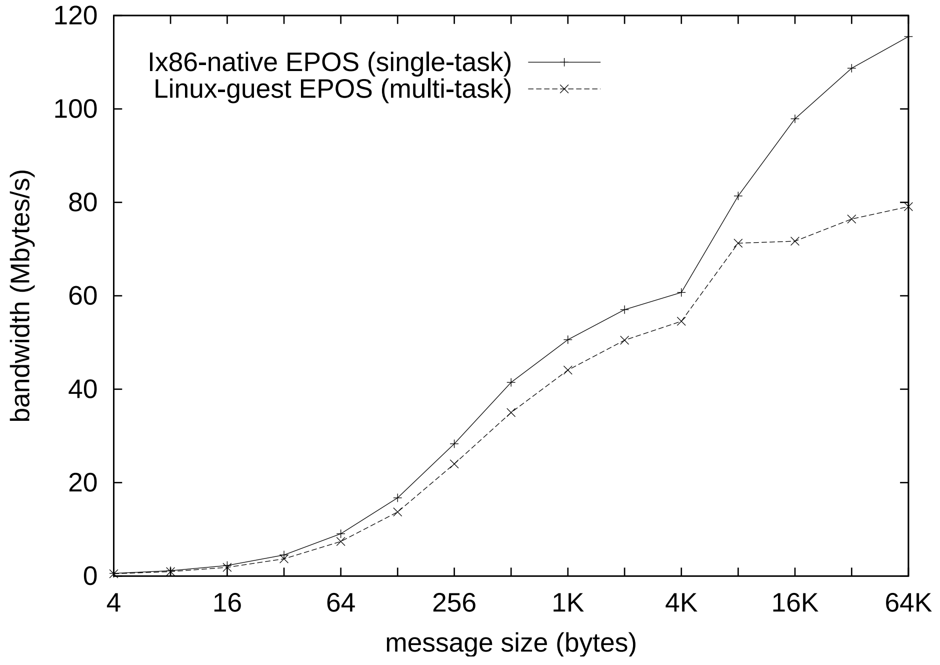

Figure 5.8: One-way bandwidth achieved by the `Myrinet_Network` abstraction on SNOW.

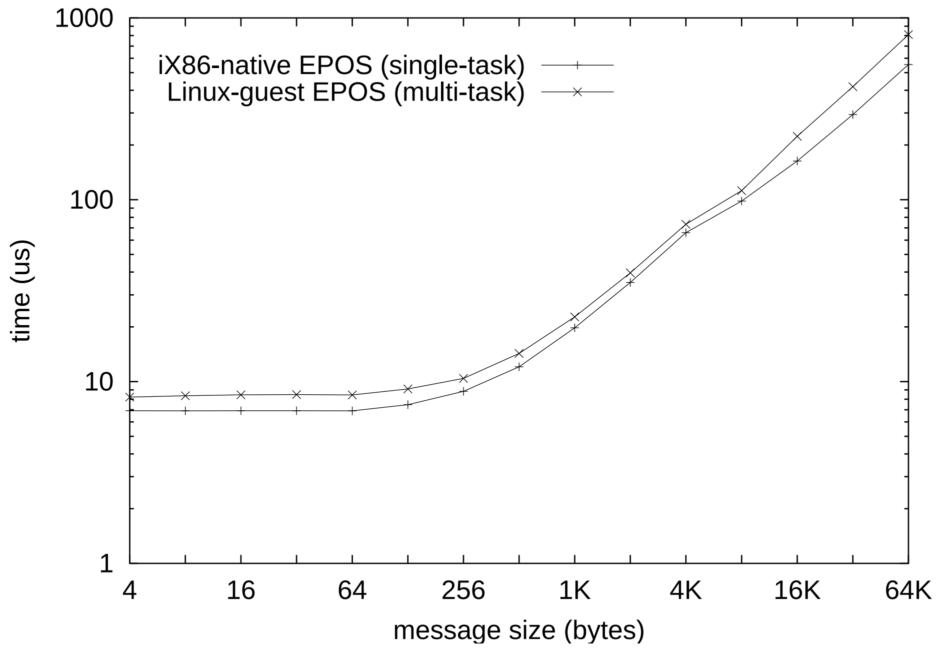

Figure 5.9: One-way latency measured for the `Myrinet_Network` abstraction on SNOW.

1 (copy) and 2 (host to NIC DMA) in the LINUX-guest implementation. The ix86-native version, which does not use stage 1, shows a growing advantage over the guest version as the message length increases. Even if large messages are split in small packages, the bursts of memory transactions performed by both stages exceed the memory subsystem capacity. This phenomenon is direct evidence that apparently harmless features of conventional operating systems that are enabled by design (multitasking support in this case) do indeed impact applications that do not need them.

5.3.2 Utilities

EPOS *setup utility* [section 4.5.2.1] was implemented for SNOW taking advantage of the configuration capabilities of the service network. For ease of maintenance, SNOW computing nodes do not have disks. Hence, booting is accomplished by each node downloading a boot image from the server over the service network. This boot image includes a node-specific description of EPOS—which among other configuration information contains the node's logical id—and the executable image of the first process. In this manner, each node can be initialized with a distinct application (and consequently a distinct configuration of EPOS) or with distinct processes of a task-parallel application.

The initialization of data-parallel applications in a *Single Program, Multiple Data* (SPMD) configuration is accomplished with a single boot image that is downloaded from the server by all nodes. Subsequently, the setup utility replaces the in-image description of EPOS with a node-specific version obtained from the server using the *Dynamic Host Configuration Protocol* (DHCP) [Dro93] capabilities of the service network. Therefore, users can select the initialization strategy according to the needs of each application.

Both setup and init utilities expect executable images, including the operating system, to be in the *Executable and Linking Format* (ELF) [Int95b]. This enables ordinary compilation and linking tools on an ix86-based LINUX workstation to be used as a cross-compiling environment for EPOS. The separation of the *init utility* [section 4.5.3.1] from the operating system image is accomplished using these tools. The `init` method of every abstraction is defined in a separate compilation unit that is linked with the init utility. Undefined references to EPOS addresses are then resolved by the linker consulting (but not linking) EPOS ELF image.

5.3.3 Tools

A simplified version of EPOS *syntax analyzer* was implemented for SNOW. This analyzer is able to identify abstractions that are necessary to support the execution of applications written in C++. It invokes the compiler indicated by the user to compile the application with all inflated interfaces left unbound. If libraries are supplied, incremental linking is performed in order to collect further references to EPOS abstractions contained in those libraries. The output of this compilation/linking process is an object file whose symbol table contains undefined references to EPOS abstractions designated by the application (directly or indirectly).

The symbol table is subsequently manipulated by the analyzer to produce a list of operations invoked for each EPOS abstraction. Abstractions that are directly referenced, i.e. that are not referenced through the inflated interface of their families, are also included in the analysis to substantiate further configuration decisions. For example, an application can directly designate a system abstraction that constrains the binding of other inflated interfaces.

Submitting the *dinning philosophers* program presented in section 4.6 to the syntax analyzer produces the following results:

```
Synchronizer {
    constructor(void);
    lock(void);
    unlock(void);
}

Thread {
    constructor(int (*)( int ), int);
}
```

In comparison to a full-fledged EPOS analyzer, this implementation misses detailed scope information. Instead of functions, compilation units are used as scope delimiters. In the example above, the information that `Synchronizer` was instantiated in the global context, and that `Thread` was instantiated in the context of function `main` was lost. Nevertheless, using the compiler to parse applications on behalf of EPOS analyzer brings two important advantages: first, conflicts between analyzer and compiler due to subjective parsing strategies are eliminated; second, the syntactical information output by compilers on object files represents the input program after the execution of eventual static metaprograms (i.e. templates are instantiated, static constant data members and inline member functions are resolved). It would be extremely difficult for an autonomous syntax analyzer to deliver these characteristics, especially with regard to static metaprograms that optimize the input program.

An ideal implementation of EPOS analyzer could be obtained with a compiler that output the parse tree and the flow graph of programs as they are compiled. Such a compiler would retain the qualities of the current implementation while attaining more detailed scope information. Unfortunately, none of the compilers available for SNOW offers this feature.

The results of the syntactical analysis are subsequently processed by EPOS *configurator*. This tool relies on a catalog of composition rules to select the components that better match the requirements collected by the analyzer. This catalog embeds a list of existing

```
<?xml version="1.0"?>
<!DOCTYPE EPOSConfig SYSTEM "eposconfig.dtd">
<EPOSConfig>
 ...
    <family id="Thread" type="Incremental" default="Exclusive">
        <member id="Exclusive" type="Sole" pre="Task=Exclusive"/>
        <member id="Concurrent" type="Sole"/>
        <member id="Cooperative" type="Sole"/>
    </family>
 ...
    <family id="CPU_Scheduler" type="Uniform" default="FCFS"
            pre="Thread=Concurrent">
 ...
    <family id="Synchronizer" type="Dissociated" default="Mutex">
        <member id="Mutex"/>
        <member id="Semaphore"/>
        <member id="Condition"/>
    </family>
 ...
</EPOSConfig>
```

Figure 5.10: Fragments of EPOS catalog of composition rules.

EPOS abstractions and a description of the target platform that serve as additional constraints for the configuration procedure. In the current implementation, composition rules are represented in the *eXtensible Markup Language* (XML) [W3C98]. An associated *Document Type Definition* (DTD) enables a flexible use of the rules catalog by diverse tools.

Some fragments of the rule catalog concerning the dinning philosophers program analyzed earlier are depicted in figure 5.10. Families of abstractions are declared with the `family` element, while family members are declared with the `member` element in the scope of their families. Attribute `default` of the `family` element designates the family member to be used in case the configurator does not have enough arguments to make a selection.

The order in which members are declared in a family designates their cost, the first being the cheapest. Setting the `type` attribute of a `member` element to “Sole” indicates that the member cannot be used simultaneously with other family members. In the case of `Thread`, however, the `type` attribute of `family` set to “Incremental” allows the configurator to replace a cheaper member with a more expensive one. For example, if a compilation unit is satisfied with the “Exclusive” member, but another requires “Cooperative”, the latter is used for the entire application.

Both `family` and `member` elements can be assigned composition rules in the form described in section 4.5.1.2. Attributes `pre` and `pos` are used for this purpose. In figure 5.10, member “Exclusive” of the “Thread” family depends on the member with the

```
Framework {
    constructor(Id const &);
    constructor(Framework const &);
    id(void);
    valid(void);
}
...
Mutex {
    constructor(void);
    lock(void);
    unlock(void);
}

Semaphore {
    constructor(int);
    p(void);
    v(void);
}

Condition {
    constructor(void);
    lock(void);
    unlock(void);
    wait(void);
    signal(void);
    broadcast(void);
}
...
```

Figure 5.11: Fragments of EPOS catalog of system interfaces.

same name from the "Task" family, and family "CPU_Scheduler" requires the "Concurrent" member of the "Thread" family.

An interpreter for this configuration language was implemented in the context of a graphical front-end for EPOS configuration tools [Röm01]. Besides being able to guide the process of tailoring EPOS to a particular application, this visual tool delivers a feature-based interface to manually configure EPOS. It shall be extended in the future to accomplish a *management console* for SNOW, from which hardware and software maintenance tasks will be performed.

The interfaces of the abstractions listed in the catalog of composition rules are automatically collected in a catalog of system interfaces applying the syntax analyzer to EPOS component repository. A stretch of EPOS catalog of system interfaces is depicted in figure 5.11. The special interface `Framework` comprises operations that are implicitly supplied by the component framework to all abstractions. This entity is manually inserted in the catalog.

In order to select the best realization for each inflated interface referenced by the application, the configurator crosses the output of the syntactical analysis with the catalog of interfaces. Realizations with interfaces that *contain* the required signatures are probed considering the order specified in the catalog of composition rules. When one is found that matches the required signatures, the respective composition rules are checked. If no conflicts are detected, the inflated interface is bound to that realization, otherwise, the search continues. As explained in section 3.6, the impossibility to find an adequate realization reveals either an application flaw, or a member of a dissociated family that has not yet been implemented.

Regarding the dinning philosophers program, `Synchronizer` would be bound to the cheapest realization that features a default constructor and includes methods `lock` and `unlock`, that is, `Mutex`. In turn, the `Thread` inflated interface would be bound to `Concurrent`, since `Exclusive` does not provide the required constructor. As explained in section 4.6, the `Cooperative` member of the `Thread` family has a lower overhead than `Concurrent`, but it is solely elected by method `pass`, which is not realized by `Concurrent` (this was arranged reversing the order of both abstractions in the catalog of composition rules).

A configuration produced by EPOS configurator consists of *selective realize keys*, which designate the binding of inflated interfaces of abstractions and scenario aspects, and *configurable feature keys* that designate the configurable features that must be enabled for abstractions and scenario aspects. This set of keys is translated by EPOS *generator* to `typedefs` and `Traits` structures that control the operation of the component framework metaprogram throughout the compilation of an EPOS instance. Indeed, the generator cannot be considered separately from the compiler, since actual generative tasks, such as the adaptation of abstractions to scenarios and the composition of abstractions, are performed by the compiler as it executes the metaprogram.

5.4 Summary

Clustering commodity workstations as a cost-effective alternative to expensive massively parallel processors has been explored for quite a long time. Recently, improvements on commodity microprocessors and interconnects begun to make clusters competitive also in absolute performance, suggesting that both hardware technologies are about to merge. Regarding software, however, the dedicated run-time support systems used on MPPs show considerable advantages over the commodity workstation operating systems typically used on clusters. In order to compensate the difference, commodity systems are often patched with specialized subsystems, in particular user-level communication. However, the inflexible structure of such operating systems severely restricts optimizations.

The prototype implementation of EPOS, which aims firstly at verifying design decisions, was strongly motivated by the possibility of introducing an application-oriented operating system to the high-performance cluster computing scene, hence the decision of implementing it for the SNOW cluster of workstations. This cluster consists of 24 Intel iX86-based processing nodes interconnected with a MYRINET high-speed network and connected to a server through an FAST ETHERNET service network.

The implementation of EPOS for SNOW was conducted as a last refinement of design, rather than an isolated phase. Constant feedback helped to enhance design specifications while yielding better support for implementation decisions. Abstractions, hardware mediators, scenario aspects, component framework, setup and init utilities, and configuration tools were implemented to produce two versions of the prototype: one that runs "natively" on SNOW, and the other that runs as a "guest operating system" on LINUX.

The implementation of the `Myrinet_Network` abstraction benefited from the LANAI processor on the MYRINET interface card to shape an asymmetric multiprocessor configuration with the main processor, enabling the network to be modeled as a bounded buffer. In order to communicate, application processes write message descriptors on the memory shared by both processors. Messages are then autonomously processed by the portion of `Myrinet_Network` abstraction that runs on the NIC, which profits from the parallel operation of MYRINET hardware components to implement a communication pipeline.

The syntax analyzer implemented for SNOW is able to identify EPOS abstractions that are needed to support the execution of applications written in C++. It uses the compiler indicated by the user to compile the application with all inflated interfaces left unbound, thus producing a symbol table with undefined references to EPOS abstractions. The symbol table is subsequently processed to produce a list of the operations invoked for each abstraction.

The catalog of composition rules used by EPOS configurator was written in XML, with and associated DTD supporting a flexible use by diverse tools. By consulting this catalog, which embeds the relative cost of abstractions, the configurator is able to select the components that better match the requirements collected by the analyzer. The output of this configuration process are selective realize and configurable feature keys, which are subsequently translated in `typedefs` and `Traits` structures by the generator. The fact that EPOS generator does not manipulate the source code of abstractions and scenario aspects directly—the component framework metaprogram does it in behalf of the generator—allowed this simple implementation. However, working together, generator and framework are able to adapt and compose selected abstractions to produce application-oriented instances.

Chapter 6

Discussion

This chapter discusses application-oriented system design, identifying its highlights and limitations. Comparisons with related design strategies will be made, and the possibilities of deploying it to support the design of other kinds of software than application-oriented operating systems will be considered. A similar study is subsequently presented for the EPOS system. The chapter is closed with a discussion about the perspectives of further development and deployment of the ideas proposed in this dissertation.

6.1 Application-Oriented System Design in the Realm of Software Engineering

Application-oriented system design is a software development methodology that addresses issues concerning the engineering of statically configurable system-level software, in particular application-oriented operating systems. The method approaches static configurability by guiding the decomposition of the envisioned domain into software components that can be assembled to produce system instances according to the needs of particular applications, thus filling the gap left by all-purpose operating systems.

As a methodology specially conceived to support the development of system-level software, application-oriented system design presents many advantages when compared to less specific methodologies. One important factor ignored by other methodologies is that operating system abstractions often enclose entities that span three different domains: hardware, operating system, and application. If these domains are not properly bridged, a situation may arise in which system components exhibit excellence at one level, but disappoint at another. By considering application-specific views of domain entities and separating scenario aspects from abstractions, application-oriented system design consistently addresses this issue.

Regarding operating system design peculiarities, many conventional design methodologies presuppose a single paradigm. At the begging of the "object revolution", it was often claimed that object-oriented methods that could prevent "design slips toward other paradigms" were able to produce better designs. Nowadays, the combination of paradigms is commonly practiced by software designers, which profit from each paradigm's strengths to deal with distinct design issues. System-level software makes the limitations of a single paradigm more evident, since modeling some physical devices, in particular with regard to synchronization and timing, challenges any individual paradigm. Aware of this, application-oriented system design combines techniques from different paradigms (e.g. aspect separation and variability analysis), as well as making room for spontaneous combinations.

Several design methods are being used to guide the development of component-based software, but not many of them have been conceived explicitly for this purpose. The most common deficiency of such methods is the absence of a domain analysis and decomposition strategy. Consequently, many software components are being defined in the scope of specific systems (instead of a domain) and do not achieve the expected degree of reusability. Application-oriented system design features a well-defined process to decompose a domain into abstractions that capture application-specific perspectives of each domain entity. Instead of modeling monolithic abstractions, commonality and variability analysis build families of abstractions. Abstractions strongly connected by commonalities are gathered in the same family, while variability analysis renders the particular characteristics of each family member.

Another common deficiency of those design methods is the inability to guide the separation of scenario aspects from abstractions. Scenario-specific abstractions usually can only be deployed within the scenario for which they have been conceived, limiting reusability. Application-oriented system design emphasizes the separations of execution scenario aspects from abstractions, extending variability analysis to determine whether a variation is inherent to the family or whether it originates from one of the execution scenarios considered. Scenario-independent abstractions—which can be reused in a variety of scenarios—are the outcome of this procedure.

Another important advantage of application-oriented system design is that it fosters the definition of software components during domain decomposition, while most other methods cover the issue later during the specification of a physical model. Complications to match abstractions and components often arise with those methods, especially with regard to cross-component properties and excessively large components. Therefore, each application-ready abstraction in an application-oriented system design is modeled to yield exactly one software component that can be independently deployed. System-wide properties that crosscut component boundaries are modeled separately as scenario aspects.

The separation of concerns promoted by application-oriented system design helps to produce less complex software artifacts, improving most software quality metrics. However, the subdivision of abstractions into families could affect maintainability, since the number of software artifacts produced can be quite high. Nevertheless, modeling scenario aspects, each of which applies to several abstractions, and configurable features that replace the specialization of all individual members of a family by a single construct, should keep the number of elements in an application-oriented system design at manageable levels.

A large number of components could also impair usability, for application programmers might not be able to select and compose so many items. Therefore, application-oriented system design assigns every family an inflated interface, which allows whole families to be used as single abstractions. Furthermore, application-oriented system design exploits inter-family relationships to model reusable software architectures in the form of component frameworks. These frameworks prevent users from carrying out erroneous compositions.

The scenario adapter concept is another strength of application-oriented system design. It provides a controllable mechanism to apply scenario aspects to abstractions, sustaining the separation of concerns pursued during domain decomposition. Scenario adapters act as agents that mediate the interaction between scenario-dependent clients and scenario-independent abstractions. These constructs can be implemented with static metaprogramming techniques to yield a low-overhead adaptation and composition mechanism that can be used to build component frameworks.

An application-oriented operating system designed according to the guidelines of application-oriented system design can be tailored to particular applications through the selection of abstractions, scenario aspects, and configurable features that are arranged in a component framework. In addition, the inflated interface binding mechanism used to select system parts can be controlled externally, allowing for the automation of the tailoring process. Tools can analyze applications and available system components to determine the most adequate system configurations.

Notwithstanding the robustness of its concepts, application-oriented system design certainly has limitations. Noticeably, application-oriented system design requires users to be familiarized with object-oriented design, from which it inherits a series of concepts, but then defies some of its basic notions with scenario aspects and inflated interfaces. This might bring confusion to the mind of some designers. Concepts such as families of abstractions and component frameworks are more intuitive and should be easily accepted.

Nevertheless, conflicts between object-oriented design and application-oriented system design are insignificant when compared to what may be caused by inviting an operating system designer to consider application-orientation. Typical operating system designs are strongly constrained between hardware and standardized application program interfaces, rendering many software engineering techniques inapplicable. The removal of the superior “lid” by application-oriented system design may leave some designers aimless. In this situation, the higher complexity of application-oriented system design concepts may be unmanageable.

Indeed, the complexity of an application-oriented system design may exceed that of designs produced with other methodologies, for high scalability is achieved with a considerable increase in the number of software artifacts. These artifacts are individually simpler than those otherwise produced, being easier to specify, implement, and maintain. However, the complexity of the system as a whole will certainly exceed that of a monolithic design. Nevertheless, the adoption of UML as design notation enables application-oriented system design to be conducted with the aid of a variety of CASE tools that feature powerful mechanisms to manage complex sets of software artifacts.

6.1.1 Comparison with other Methodologies

Application-oriented system design will be subsequently compared with the design methods discussed in chapter 2. Although some of those methods do not envision domain engineering, and do not explicitly address issues concerning operating system design, they feature techniques that can be used to build application-oriented operating systems. This comparative study aims at identifying similarities, advantages, and disadvantages of application-oriented system design with regard to those methods.

Family-based design: Application-oriented system design shares several common aspects with *family-based design* [section 2.2.1]. Both methodologies emphasize *separation of concerns*, deploying commonality and variability analysis as the basic domain decomposition strategy. Domain entities whose commonalities are more significant than variations are grouped in *families*.

However, family-based design and application-oriented system design disagree on some important points. Family-based design does not put limits on what family members can be. A family-based design could come up with families of elementary functions alongside families of complex abstractions. This seeming flexibility makes it difficult to match logical and physical design models in the context of a component-based system, for some logical entities yield an excessive number of components, while others cross component boundaries. Application-oriented system design copes with this issue guiding the specification of application-ready abstractions that directly correspond to components.

Furthermore, family-based design does not emphasize the separation of *scenario aspects*, interpreting them as ordinary variability that yields new family members. This severely affects the reuse of a family in new scenarios, besides complicating maintenance with a substantial increase in the number of family members.

Application-orientation is addressed in the context of family-based design by *incremental system design* [section 2.2.1.1], which fosters the modeling of families with a *minimal basis* to which successive *minimal extensions* are applied all through the application. In comparison to application-oriented system design, this strategy has a major shortcoming: the *minimal* criterion for extensions, in addition to the lack of scenario aspect separation, can result in a family containing an exaggerated number of elements. Consequently, management and use become complicated, and matching the application's needs impractical. For instance, the *thread* family tree could comprise dozens of levels from its patriarch co-routine to a remotely accessible, multiprocessor- and multitask-ready descendant.

Compared to the *family-oriented abstraction, specification, and translation* variant of family-based design, application-oriented system design is in disadvantage in what regards the representation of commonalities. That method utilizes *application-oriented languages* to quickly specify commonalities, which are subsequently regarded as design secrets. In this way, domain decomposition can concentrate on variability analysis. Application-oriented system design does not contemplate such languages, so the modeling of commonalities demands more effort. Nevertheless, specifying such languages in advance to domain decomposition can be a defying task.

Object-oriented design: Application-oriented system design is a multiparadigm design method centered on object-orientation. Therefore, many *object-oriented design* [section 2.2.2] techniques are reused in an application-oriented system design. Besides common aspects inherited from family-based design, both methodologies abstract domain entities as *objects*. Commonalities lead objects to be gathered in *classes*, while variability builds class hierarchies.

Nevertheless, class hierarchies constitute just one of the strategies to build a family in the realm of application-oriented system design. Representing a *dissociated family*—whose members are usually strongly connected by semantics, but lack a common structure—as a class hierarchy leads advanced family members to incorporate unnecessary elements, consequently compromising application-orientation. Moreover, the fragile base class of such an hierarchy often fails to carry on family extensions, having to be reformulated to accommodate members not initially considered, sometimes even affecting other family members. In contrast, the factorization of a dissociated family as guided by application-oriented system design causes commonalities to be gathered in a *common package* (instead of a base class). Elements of a common package are individually reused by each family member.

Both methodologies recognize the importance of giving different members of a family a common interface, so it can be handled as a single abstraction. However, object-oriented design's answer to the question, *polymorphism*, has considerable shortcomings when compared to application-oriented system design's *inflated interfaces*. First of all, the inflated interface concept does not exclude polymorphism. An inflated interface allows the designer to decide whether the common interface assigned to a family of abstractions implies in subtyping relationships between their members or not. This is especially important to preserve domain correspondence in dissociated families, whose members define independent types. Polymorphism would require such families to be modeled as a class hierarchy with an artificial base class that is not sustained by the organization of corresponding entities in the domain.

Considering implementation, representing the common interface of a family as an inflated interface, instead of a polymorphic abstraction, has two main advantages:

- Inflated interfaces result in less run-time overhead and better memory utilization, for no indirect procedure call mechanisms are necessary and only effectively used family members are included in a system configuration (the exact type of a polymorphic object is only defined at run-time, so all connected types have to be included).
- Inflated interfaces can be externally bound at compile-time without modifications to the source code, while polymorphism shifts issues concerning family member selection to run-time. Selecting a family member at run-time, however, may require complex pieces of configuration tools be embedded in the application.

Another significant difference between application-oriented system design and object-oriented design concerns scenario aspects. While application-oriented system design isolates them from abstractions, object-oriented design models scenario-specific specializations of abstractions. Such specializations can seldom be reused in a different scenario. Moreover, if the implementation of a scenario aspect needs to be modified, the corresponding specializations of all families of abstractions designed to perform in that scenario have to modified. A single element would be modified in an application-oriented system design.

Replacing the modules of an object-oriented design's physical model with components can usually be straightforwardly accomplished. However, object-oriented design does not directly address inter-family relationships that cross the boundaries of modules, so achieving components that can be individually reused may require a large effort from designers. Likewise, object-oriented design does not feature techniques to specify how components can be composed to yield a concrete system. Application-oriented system design addressed both issues with *component frameworks,* which model reusable system architectures that emanate from inter-family relationships.

Collaboration-based design: One important concept of *collaboration-based design* [section 2.2.3] incorporated by application-oriented system design is that abstractions can play different roles in different contexts. However, this concept is deployed quite differently in both methodologies. In collaboration-based design, objects are explicitly modeled considering the roles they can play in predicted collaborations. Application-oriented system design explores the concept to model variations in the behavior of abstractions that originate from different execution scenarios. Therefore, scenario aspects can be understood as externally defined roles of abstractions. When a scenario aspect is applied to an abstraction with a scenario adapter, the abstraction "starts to play the corresponding role", thus collaborating in a scenario.

When a software development methodology is deployed to engineer a domain, instead of a single system, reusability assumes a broader connotation. Predicting which fractions of a domain will be covered by each system is usually not possible, therefore domain entities must be modeled to be reused in disregard of each other, yielding independently deployable components. Collaboration-based design was one of the first methodologies to acknowledge this necessity, guiding the specification of independently deployable collaborations. Application-oriented system design extends this notion to model independently deployable abstractions.

The design of EPOS described in chapter 4 covers abstractions and scenario aspects in the domain of high-performance dedicated computing. The engineering of such a complex domain revealed that specifying independently deployable abstractions, more than a source of improvements, is a necessity. Modeling abstractions to be reused independently of each other, independently of scenario aspects, and independently of predefined system architectures, drastically simplifies their design, improving overall quality.

When EPOS begun to be designed, it was not clear that such a level of independence could be achieved, so more traditional designs were first considered. For instance, an object-oriented framework, in which abstractions extend framework entities. Such a design would have created unnecessary dependencies that would complicate reusability. Another possibility considered was to deploy external tools, much in the sense of aspect-oriented programming, to modify EPOS components in order to enable them to be reused in specific scenarios. Such tools became superfluous for EPOS, partially because of the static metaprogramming techniques adopted, but mainly because of the degree of independence promoted by application-oriented system design, which allowed abstractions to be adapted without external manipulation.

Subject-oriented programming: Comparing *subject-oriented programming* [section 2.2.4] with application-oriented system design would make little sense, for both methods cover very different segments of the software engineering spectrum. While application-oriented system design targets the engineering of the operating system domain as a collection of reusable abstractions that can be configured to match the requirements of specific applications, subject-oriented programming extends object-oriented programming to handle issues concerning the extension of existing software without modifying the original source, and the decentralized development of classes.

Nevertheless, the subject-oriented programming observation that different clients may be interested on different aspects of a class, leading to *subjective views* of it, meets application-oriented system design domain decomposition strategy, with *subjects* being represented either as members of a family of abstractions or as scenario aspects, and scenario adapters *reconciling subjective views* much like subject composition does in subject-oriented programming.

It would be more appropriate to compare subject-oriented programming tools to those introduced by EPOS to support automatic configuration [section 4.6]. Clearly, the subject-oriented programming project at IBM Research [IBM01] is a much larger project than this dissertation, with tools that are far more elaborate. Nonetheless, both set of tools rely on externally defined composition rules and deploy statically metaprogramming techniques. Subject-oriented programming C++ support is able to derive abstract description of components (*subject labels*) from their source code. A similar description is obtained for EPOS applying the syntactical analyzer to the component repository. Labels are subsequently composed, in accordance with composition rules written in a specific declarative language, to produce a result label that describes the composed subject. EPOS relies on a catalog of rules written in XML for the same purpose. Both processes end with a generator processing a composite description.

Nonetheless, subject-oriented programming tools were conceived to operate with arbitrarily designed systems, while EPOS tools are intrinsically dependent from an application-oriented system design. This allows EPOS tools to analyze target applications in order to select the components that better serve them, an unknown notion to subject-oriented programming tools.

Aspect-oriented programming: Similar to subject-oriented programming, *aspect-oriented programming* [section 2.2.5] does not really feature a design methodology. The idea of isolating non-functional aspects in reusable constructs is not sustained by a systematic domain decomposition strategy. Moreover, functional aspects are given secondary importance by aspect-oriented programming techniques. Nevertheless, aspect-oriented programming emphasis on aspect separation is upheld by application-oriented system design.

Furthermore, aspect-oriented programming techniques and tools can be useful to implement application-oriented system designs. In this case, *aspects* would implement *scenario aspects*, and *weavers* would play the role of *scenario adapters*. This would bring one major advantage over the original scheme: describing aspects with special *aspect languages* could considerably reduce efforts and improve correctness. However, the effort to implement such tools and to specify an aspect language would only be worthwhile in the long term. Besides, the correctness enhancements obtained by specifying aspects with a more adequate language can vanish in the face of the nuisance of congregating different languages, processed by different tools, at generation-time.

6.1.2 Support for other Kinds of Design

Application-oriented system design has been conceived aiming at engineering application-oriented operating system as a collection of statically configurable software components. However, many of the concepts introduced by application-oriented system design can be useful to design other kinds of software as well.

Dynamic configurable operating systems: Application-oriented system design does not exclude dynamic configurability. Some statically composed abstractions include dynamically reconfigurable elements, as with EPOS `CPU_Scheduler` abstraction described in section 4.3.2.3. Deploying application-oriented system design to model a full-dynamically reconfigurable system would also be possible, but some important issues regarding this kind of system are not covered by the method.

Apparently, deploying application-oriented system design as-is, and following a particular implementation discipline could achieve the desired effect. Inflated interfaces would be implemented as abstract base classes, and abstractions would be implemented as subtypes of their family's interface. A framework element, similar to `handle` in the context of EPOS (see section 4.5.1), would build an *abstract factory* [GHJV95] for abstractions, allowing different members of a family to be dynamically created. In this context, all abstractions would be included in the system at compilation-time to be freely deployed at run-time. If incremental loading is desirable, a strategy similar to the one explained in [DSS90] could be easily devised thanks to the degree of independence of abstractions.

However, supporting dynamic reconfiguration in the style of *reflective systems* would be a major challenge, since application-oriented system design does not address some important issues concerning the subject. In order to support the systematic construction of reflective operating systems, a design method would have to guide the modeling of abstractions considering their *meta-level* representation and the meta-level interactions that drive dynamic reconfiguration.

All-purpose operating systems: Deploying application-oriented system design to model an all-purpose operating systems should be straightforward. Application-oriented system design could be applied to engineer an all-purpose operating system as if it was an application-oriented operating system, differentiating only during the generation of a system instance. In other words, any application-oriented operating system can be configured to perform as an all-purpose operating system. Such instances would be generated including all members of uniform and dissociated families plus the most comprehensive members of incremental families and full merged members of combined families. Likewise, all scenario aspects and configurable features would be activated. In this way, all the functionality modeled for the system would be available to generic applications.

Although such a system would not be distinguishable from a generic operating system from the application point of view, engineering it according to application-oriented system design directives would bring system developers all the advantages of component-based software, especially reusability and maintainability.

Applicative software: Application-oriented system design could also be used to model applicative software, conferring it the same scalability of an application-oriented operating system and all the benefits of a component-based realization. However, some concepts of application-oriented system design originate from the necessity of bridging hardware entities with application abstractions. This does not hold for applicative software. It is likely that an application design would comprise less dissociated families and more uniform ones, for many dissociated families arise from hardware idiosyncrasies. Furthermore, a slightly different criterion to identify scenario aspects and configurable features would be necessary in this case, since environmental and built-in properties might be difficult to distinguish in a smoother scenario (i.e. a scenario that does not span distinct domains). This has also been acknowledged by aspect-oriented programming [MLTK97].

6.2 EPOS in the Realm of Operating Systems

EPOS (*Embedded Parallel Operating System*), the experimental application-oriented operating system developed in the scope of this dissertation, is the outcome of an application-oriented decomposition of the high-performance dedicated computing domain. It consists of a collection of abstractions and scenario aspects that can be arranged in a component framework to produce system instances according to the necessities of particular applications.

Notwithstanding the intricacy of the target domain, the separation of concerns promoted by application-oriented system design greatly simplified the specification of EPOS individual elements, much in a “divide and conquer” strategy. The implementation of a prototype for the SNOW cluster made it clear that the increase in complexity resulting from successive partitioning is well worth paying. Compared to ABOELHA[1], EPOS is far more flexible and maintainable. ABOELHA’s μ-kernel has been designed to be quite compact, suggesting that further decomposition was not necessary. However, its monolithic, scenario-dependent abstractions are extremely difficult to maintain. Often, minor modifications in one abstraction trigger undesirable side-effects on other “apparently independent” abstractions, causing the whole system to undergo a new extensive test phase.

Having followed the directives of application-oriented system design to decompose the domain of high-performance dedicated computing, EPOS was able to specify autonomous families of abstractions. These abstractions concentrate on essentials of the domain entities they abstract, while environment-related properties were separately captured as scenario aspects. In addition, a factorization process collected family commonalities in common packages and configurable features. This thorough partitioning produced software components that can be independently deployed and maintained, hence minimizing adverse side-effects.

Capturing elements of reusable software architectures in a component framework also contributed to make EPOS highly scalable. Unlike traditional object-oriented frameworks,

[1]ABOELHA is a former operating system developed by the author [FAPS96]. It was designed around a μ-kernel that constitutes a run-time support substrate for distributed objects.

the EPOS component framework does not require abstractions to conform to specific implementation directives, or to incorporate special constructs. This design allowed the framework static metaprogram to be successively streamlined without affecting abstractions and scenario aspects. The resultant component framework is more intelligible, has a null intrinsic overhead, and sustains a scalable software architecture that enables EPOS to support applications of assorted complexity.

Another fundamental design decision was the isolation of non-portable elements within the sphere of hardware mediators and the setup utility. This isolation allowed abstractions that interact with hardware, such as `Thread` and `Address_Space`, to be modeled in multiple levels, each one dealing with specific issues. Even though EPOS has not yet been ported to other hardware platforms, the guest-level implementation for LINUX corroborated many design aspects concerning portability, since most hardware mediators were redefined for that implementation with no impact on abstractions and scenario aspects.

As a domain engineering venture, EPOS aspires reaching out to “all” entities in the target domain. This aspiration, however, is definitely unachievable, since an application domain is not an exact concept that can be absolutely characterized, but a subjective set of entities, concepts, techniques, etc. Although EPOS features an ample perspective of the high-performance dedicated computing domain, certainly many particularities have not yet been covered. For instance, the extensive set of abstractions, hardware mediators, and scenario aspects that makes up the process abstraction in EPOS, when subjected to the criticism of other system designers, revealed deficiencies in regard to some particular application use-cases.

As currently modeled, EPOS is unable to deliver the benefits of the `Exclusive_Thread` abstraction [section 4.3.2.2] to applications executed by a single thread per processor on a multiprocessor node (it presupposes a single thread per node). However, suppressing memory management and scheduling would be possible in such cases and would considerably enhance performance. Fortunately, the application-oriented system design of EPOS allows for extensions. The `Thread` family could be enriched with a `Processor_Exclusive` member that would extend `Exclusive` to provide for the missing support. The `Task` family could be similarly extended to allow multiple single-threaded tasks to benefit from the proposed optimizations[2].

EPOS features a set of configuration tools that are able to automatically tailor the system to specific applications, identifying and configuring necessary abstractions and scenario aspects and subsequently generating a customized system instance. This automatic configuration strategy is of great value for users, who are freed from much of the burden of system configuration. Nevertheless, it is restricted to application requirements that can be expressed through syntax, falling back to manual configuration otherwise. Com-

[2]In platforms that do not provide address relocation mechanisms, the abstraction `Processor_Exclusive_Task` would be assisted by the configuration tools to issue appropriate base addresses for tasks, thus shaping a `Mutual_Flat` member for the `Address_Space` family.

pared to systems that only support manual configuration, this is an advance, but EPOS configuration tools could be extended to support more sophisticated approaches.

Complementing the syntactical analysis of applications with data-flow analysis, for example, could render more accurate application requirements. Some system operations explicitly invoked by the application, especially regarding synchronization, could be proven unnecessary by such analysis. They could be automatically suppressed.

Another aspect of the existing configuration tool set that could be improved is the tie-break criterion: when application requirements are fulfilled by two or more members of a family of abstractions, the one with the lowest overhead is taken. This cost model disregards significant properties of abstractions that could lead to better configurations. The uniform family of scheduling policies maintained by `CPU_Scheduler` [section 4.3.2.3] is particularly affected by the adopted model, because, being a uniform family, its members are syntactically equivalent. Some applications give syntactical hints about the scheduling policy that should be used (e.g. explicitly assigning priorities to threads, changing the time-slice length, designating a user-defined thread to perform the role of scheduler), but many do not. In such cases, the overhead criterion elects `FCFS` as the scheduling policy, since it has the lowest run-time overhead. Obviously, this arbitrary decision is often inadequate. Evaluating the configuration as a whole to collect “hints” about tied abstractions would increase the ratio of optimal configurations. For instance, evidences of concurrency, such as task creation and memory sharing, could increase the cost of `FCFS`, eventually electing another policy.

6.2.1 Comparison with other Systems

Few operating systems describe the more abstract levels of their designs, usually concentrating on details about outstanding services. In this context, identifying systems that emphasize application-orientation becomes difficult. Nonetheless, some operating systems have been identified that render an interesting comparative study with EPOS.

CHOICES The CHOICES object-oriented operating system [CJR87, CIM92] at the University of Illinois at Urbana-Champaign has been designed as a hierarchy of object-oriented frameworks and implemented in C++. Each of CHOICES frameworks corresponds to a subsystem (e.g. virtual memory, file system, etc) that can be customized through the use of inheritance. System resources, policies, and mechanisms are represented as objects in the context of the corresponding framework.

CHOICES embodies the notion of customizing the operating system to support particular hardware configurations and applications. However, compared to EPOS, CHOICES components are large and not very customizable. Indeed, CHOICES would be better understood from the perspective of all-purpose operating systems, since its complex subsystems are not specialized with regard to applications needs. Moreover, CHOICES monolithic abstractions are difficult to reuse separately, because they incorporate framework elements and scenario aspects.

ETHOS The ETHOS reflective operating system [Szy92] at the Swiss Federal Institute of Technology covers extensible objected-oriented programming from the hardware up to the applications. It has been modeled as a strongly typed hierarchy of abstractions, for which default implementations exist.

While ETHOS explores dynamic configurability, EPOS is essentially a statically configurable system. Nevertheless, both systems agree about the limitations of inheritance to implement a customizable design. ETHOS pursues extensibility restricting the use of inheritance in favor of forwarding, with *directory objects* acting as proxies to extensions. In this way, extensions (modules) can be dynamically loaded. EPOS uses a similar forwarding strategy with its statically metaprogrammed scenario adapters, so that abstractions incorporate scenario aspects without having to inherit them. Furthermore, application-oriented system design supplies EPOS with a range of alternatives to traditional class hierarchies, including dissociated families, scenario aspects, and static metaprogramming. Combined, they prevent unnecessary dependencies and foster independent deployability.

PEACE The PEACE parallel operating system [SP94a] at GMD-FIRST follows the guidelines of family-based design. PEACE embodies the notion of application-orientation as it embraces a particular domain, namely parallel computing, with a set of predefined configurations aimed at particular application classes. The high-performance demanded by parallel applications was supplied by PEACE with a thrifty implementation concerned in making resources available to applications with as little overhead as possible. Several of these concepts have been reused in EPOS.

Nevertheless, PEACE design is essentially incremental, exporting a large number of elements that result from the successive extension of primordial ones. Selecting the proper elements to assemble an application-oriented operating system is a major difficulty in this context. Indeed, PEACE's ability to match up application requirements hinges on an intricate set of conditional compilation flags, in practice being restricted to a small set of predefined configurations. Application-oriented system design allows EPOS to reach a similar degree of configurability at the application side while keeping the number of exported elements (abstractions and scenario aspects) at manageable levels. Furthermore, inflated interfaces and scenario adapters mostly dispense with conditional compilation flags.

FLUX The FLUX [FBB$^+$97] operating system toolkit at the University of Utah consists of a *blackbox* framework and a set of components (object files) organized in libraries. In order to configure an operating system, users choose between libraries and object files, which are subsequently linked to produce a system instance. Therefore, FLUX emphasizes the reuse of unmodified binary components.

The notions of non-invasive framework and independent reuse explored by FLUX are upheld by EPOS. However, the preference for binary components diverges from application-oriented system design, since it prevents many customizations and hinders important concepts such as scenario aspect separation and functional factorization. Furthermore, FLUX components are relatively coarse, comprising device drivers and complete subsystems (e.g. file system), and can hardly be reused outside the scope of all-purpose computing.

PURE The PURE [SSPSS98] system at the University of Magdeburg defines a large set of fine-grained components that can be arranged to build system abstractions in the realm of embedded computing. Consequently, it shares many goals with EPOS as regards application-orientation and dedicated computing. However, PURE pursues such goals in essentially different ways.

PURE is mostly a family-based design, whose incremental families of abstractions are realized as class hierarchies. This strategy was inherited from PEACE [SP94a] and brings about the configurability and usability pitfalls discussed earlier: PURE exposes its internal structure to applications, delivering intermediate abstractions that are not application-ready. PURE recognizes that managing such a large set of components is beyond most users' grasp and is currently working on user-driven, feature-based tools to automate the process [BSPSS00]. EPOS, on the other hand, bets on application-driven configuration, delivering a set of tools that is able to configure application-specific instances automatically. However, this improvement in usability and maintainability is achieved at the expense of granularity[3]: EPOS does not match PURE's degree of configurability.

6.3 Perspectives

The application-oriented system design method proposed in this dissertation is able to guide the construction of highly customizable run-time support systems as an arrangement of adaptable software components. Such application-oriented operating systems can be tailored to specific applications through the selection of appropriate system abstractions that are adapted to match the environment in which applications perform, and subsequently "plugged" into a component framework.

The adequacy of application-oriented system design to conduct the design of application-oriented operating systems was corroborated by EPOS with the engineering of the high-performance dedicated computing domain. Application-oriented system design made it possible to decompose that domain into an extensive set of application-ready abstractions, which can be reused to build a variety of run-time support systems. Indeed,

[3]A reasoning about component granularity is presented in section 2.3.2.

EPOS attested that managing a complex set of small software artifacts that preserves domain correspondence is more productive than managing a small set of complex monolithic artifacts.

However, both application-oriented system design and EPOS are open projects with interesting possibilities for further research. In the realm of *software engineering*, CASE tools could be explored to increase the efficiency of application-oriented system design. Many elements of an application-oriented system design result from the detailing of the same domain entity, leading to mechanical replication that could be carried out automatically by such tools. For instance, the operations declared for each EPOS abstraction appear on a number of constructs, including the abstraction's interface, its family's inflated interface, scenario adapter, proxy, and agent. Ideally, these constructs should be automatically generated from an abstract description[4], which would also serve configuration purposes.

If such abstract descriptions of an application-oriented operating system components are boosted towards *formal methods*, verifying the correctness of component assemblages could be supported. The partitioning promoted by application-oriented system design yields individually simpler components that are more suitable to formal specification than conventional operating systems "components". Furthermore, formal methods could guide automatic configuration tools towards the optimal system configuration for an application.

In the realm of *operating systems*, an extensive implementation of EPOS would deliver dedicated applications services of unprecedented quality, whereas they originate from an application-oriented perspective of the dedicated computing domain. Moreover, EPOS scalable design allows for the quick incorporation of novel ideas, thus providing an ideal platform to experiment with operating systems.

Outside the scientific ring, however, the possibilities for application-oriented operating systems are not as promising. Besides the resistance of users to new ideas in the operating system field, many existing applications have little chance to profit from the advances of an application-oriented operating system. This became evident with the attempt to bring existing parallel applications to execute on the SNOW cluster under the control of EPOS. The dependencies of such applications on all-purpose operating systems are so deeply rooted that porting them to EPOS would have resulted in porting so many services that EPOS would degenerate into an all-purpose operating system [Pik00].

Curiously, there are no apparent reason for a parallel application, and dedicated applications in general, to show such dependencies. A deeper insight on an ordinary MPI application, such as those in the NAS Parallel Benchmark [NAS97], elucidates the issue. Applying EPOS analyzer to those applications with the aid of a "dummy" MPI implementation—that only designates EPOS abstractions needed to implement each service without actually implementing them—produces a very restricted set of abstractions: `Flat_Address_Space` (no memory management), `Exclusive_Task` and `Exclusive_Thread` (no process management), `Communicator`, `Chronometer`,

[4] EPOS currently deploys a loose set of `awk` scripts to generate a skeleton for new abstractions.

and `Console`. However, after compiling those abstractions (most of them are written in FORTRAN) and linking them against the MPI library, the list of dependencies grows close to a full POSIX system, including dynamic memory management, process management, and file I/O.

Therefore, enabling existing applications to profit from an application-oriented operating system would result in redesigning countless standardized application interface implementations from the perspective of application-orientation. To complicate things even more, many applications have been designed considering the low-quality services delivered by ordinary operating systems, and would not benefit from an application-oriented operating system even if all required APIs were available. A typical example would be applications that reorganize algorithms to cope with the high latency associated with ordinary system services. Supplying such applications with a low-latency system would be of no help.

Deploying application-oriented operating systems to give run-time support to *domain-specific* and *application-oriented* languages could be an alternative to break the kernel of standardized interfaces that suffocate the operating system. Nonetheless, such languages seem to be trapped in a similar paradox: though recognizably superior than all-purpose languages to handle specific applications, they seldom win the race against established all-purpose languages such as C++ and JAVA. It is probable that forthcoming commercial applications will only benefit from the advances of an application-oriented operating system if they succeed in reducing the dependencies on standardized application program interfaces.

Nevertheless, a new “strong partner” might be soon embracing the “fight” for application-orientation together with operating systems and programming languages. Highly customizable processors, and hardware in general, are about to reach the market [WTS$^+$97, Har01]. Such processors are able to undergo extensive reconfiguration procedures to optimize the path between functional units, to create new instructions, and even to “instantiate” functional units according to application needs. In this scenario, a handheld could assume the form of several devices simply by loading the appropriate application. When used as a mobile phone, the corresponding application would be loaded with its embedded application-oriented operating system, pushing the processor towards a *digital signal processor*. Using it as a digital TV set would reconfigure the hardware to include support for stream decoding and image rendering. However, such sophisticated features will be underused in the realm of general-purpose operating systems, since their plurality prevents ordinary features from being deactivated for the benefit of those actually needed by applications.

Bibliography

[ABB$^+$86] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: a New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, Atlanta, U.S.A., June 1986.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[And92] Thomas Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 92–94, Key Biscayne, U.S.A., April 1992.

[Aus99] Matthew H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[Bac87] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1987.

[Bau99] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[BC89] Kent Beck and Ward Cunningham. A Laboratory for Teaching Object-Oriented Thinking. *ACM SIGPLAN Notices*, 24(10):1–6, October 1989.

[BCF$^+$95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BDF$^+$97] William J. Bolosky, Richard P. Draves, Robert P. Fitzgerald, Christopher W. Fraser, Michael B. Jones, Todd B. Knoblock, and Rick Rashid. Operating System Directions for the Next Millennium. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 106–110, Cape Cod, U.S.A., May 1997.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[BFM$^+$00] Danilo Beuche, Antônio Augusto Fröhlich, Reinhard Meyer, Holger Papajewski, Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On Architecture Transparency in Operating Systems. In *Proceedings of the 9th SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, pages 147–152, Kolding, Denmark, September 2000.

[BGM86] Peter Behr, Wolfgang Giloi, and W. Mühlenbein. Rationale and Concepts for the SUPRENUM Supercomputer Architecture. In *Proceedings of the IFIP Working Conference on Highly Parallel Compters for Numerical and Signal Processing Aplications*, Nice, France, March 1986.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BO92] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[Boo87] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjaming-Cummings, 1987.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[BRB98] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, November 1998.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, 1999.

[BSP$^+$95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, U.S.A., December 1995.

[BSPSS00] Danilo Beuche, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Streamlining Object-Oriented Software for Deeply Embedded Applications. In *Proceedings of the TOOLS Europe 2000*, Saint Malo, France, June 2000.

[Bur95] Colin J. Burgess. Software Quality Issues when choosing a Programming Language. In *Proceedings of the Third International Conference on Software Quality Management*, volume 2 of *Software Quality Management III*, pages 25–31, Seville, Spain, April 1995.

[Cah96] Vinny Cahill. Flexibility in Object-Oriented Operating Systems: A Review. Technical Report TCD-CS-96-05, Trinity College Dublin, Dublin, Ireland, July 1996.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CEG$^+$98] Krzysztof Czarnecki, Ulrich Eisenecker, Robert Glück, David Vandevoorde, and Todd Veldhuizen. Generative Programming and Active Libraries. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloß Dagstuhl, Germany, April 1998.

[Che84] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–43, April 1984.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6):37–45, November 1998.

[CIM92] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[CJR87] Roy H. Campbell, Gary M. Johnston, and Vincent F. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[CMSW94] W. Robert Collins, Keith W. Miller, Bethany J. Spielman, and Phillip Wherry. How Good Is Good Enough?: an Ethical Analysis of Software Construction and Use. *Communications of the ACM*, 37(1):81–91, January 1994.

[Cop98] James O. Coplien. *Multi-Paradigm Design for C++*. Addison-Wesley, 1998.

[CPL$^+$97] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[CSP91] Jörg Cordsen and Wolfgang Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 24–28, Palo Alto, U.S.A., October 1991.

[DBM98] Peter Druschel, Gaurav Banga, and Jeffrey C. Mogul. Better Operating System Features for Faster Network Servers. *Performance Evaluation Review*, 26(3), December 1998.

[dC72] Luís Vaz de Camões. *Os Lusiadas*. Imprensa Regia, Lisbon, Portugal, 1572.

[Den65] Jack B. Dennis. Segmentation and the Design of Multiprogrammed Computer Systems. *Journal of the ACM*, 12(4):589–602, October 1965.

[Deu89] L. Petter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 System. In Ted J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 2. ACM Press, New York, 1989.

[Dij65] Edsger Wybe Dijkstra. Cooperating Sequential Processes. Technical Report EWD-123, Technical University of Eindhoven, Eindhoven, The Netherlands, 1965.

[Dij68] Edsger Wybe Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dij69] Edsger Wybe Dijkstra. Notes on Structured Programming. In *Structured Programming*. Academic Press, London, U.K., 1969.

[Dij71] Edsger Wybe Dijkstra. Hierarchical Ordering of Sequential Processes. In *Operating Systems Techniques*, pages 72–93. Academic Press, 1971.

[Dra93] Richard P. Draves. The Case for Run-time Replaceable Kernel Modules. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 160–164, Napa, U.S.A., October 1993.

[Dro93] Ralph Droms. *Dynamic Host Configuration Protocol*. Bucknell University, Lewisburg, U.S.A., October 1993. RFC 1531.

[DSS90] Sean Dorward, Ravi Sethi, and Jonathan E. Shopiro. Adding New Code to a Running C++ Program. In *Proceedings of the C++ Conference*, pages 279–292, San Francisco, U.S.A., April 1990.

[FAPS96] Antônio Augusto Fröhlich, Rafael B. Avila, Luciano Piccoli, and Helder Savietto. A Concurrent Programming Environment for the i486. In *Proceedings of the 5th International Conference on Information Systems Analysis and Synthesis*, Orlando, USA, July 1996.

[FBB$^+$97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[Fel92] E. W. Felten. The Case for Application-specific Communication Protocols. In *Proceedings of Intel Supercomputer Systems Technology Focus Conference*, pages 171–181, April 1992.

[FTSP00] Antônio Augusto Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations. In *Proceedings of 8th International Conference on High Performance Computing and Networking*, pages 417–426, Amsterdam, The Netherlands, May 2000.

[GBSP96] Wolfgang Giloi, Ulrich Brüning, and Wolfgang Schröder-Preikschat. MANNA: Prototype of a Distributed Memory Architecture With Maximized Sustained Performance. In *Proceedings of the Euromicro PDP96 Workshop*, 1996.

[Gen89] W. M. Gentleman. Managing Configurability in Multi-installation Real-time Programs. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 823–827, Vancouver, Canada, November 1989.

[GGD01] Jens Gerlach, Peter Gottschling, and Uwe Der. A Generic C++ Framework for Parallel Mesh Based Scientific Applications. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Lecture Notes in Computer Science, San Francisco, U.S.A., April 2001. Springer.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJ97] Robert Glück and Jesper Jørgensen. An Automatic Program Generator for Multi-Level Specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.

[Gog96] Joseph A. Goguen. Parameterized Programming and Software Architecture. In *Proceedings of the Fourth International Conference on Software Reuse*, pages 2–11, Orlando, U.S.A., April 1996.

[Hai86] Brent Hailpern. Multiparadigm Languages and Environments. *IEEE Software*, 3(1):6–9, January 1986.

[Har01] Reiner W. Hartenstein. Coarse Grain Reconfigurable Architectures. In *Proceedings of the Sixth Asia and South Pacific Design Automation Conference 2001*, Yokohama, Japan, January 2001.

[HCG$^+$82] K. Hwang, W. J. Croft, G. H. Goble, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates. A Unix-based Local Computer Network with Load Balancing. *IEEE Computer*, 15(4):55–66, April 1982.

[HFC76] A. Nico Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-oriented Systems. *ACM SIGPLAN Notices*, 25(10):169–180, October 1990.

[HK61] David J. Howarth and Tom Kilburn. The Manchester University Atlas Operating System, Part II: User's Description. *Computer Jorunal*, 4(3):226–229, October 1961.

[HO93] William H. Harrison and Harold Ossher. Subject-oriented Programming (a Critique of Pure Objects). In *In Proceedings of the 8th Conference on Object-oriented Programming Systems, Languages and Applications*, pages 411–428, Washington, U.S.A., September 1993.

[Hoa74] Charles Anthony Richard Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hol92] Ian M. Holland. Specifying Reusable Components Using Contracts. In *Proceedings of the European Conference on Object-oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 287–308, Utrecht, The Netherlands, June 1992. Springer.

[Hol93] Ian M. Holland. *The Design and Representation of Object-oriented Components*. PhD thesis, Northeastern University, Boston, U.S.A., 1993.

[HP91] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[IBM01] IBM Research. *Subject-Oriented Programming*, online edition, May 2001. [http://www.research.ibm.com/sop/].

[Int95a] Intel. *Pentium Pro Family Developer's Manual*, December 1995.

[Int95b] Intel. *Tool Interface Standard: Executable and Linking Format Specification (version 1.2)*, May 1995.

[ISO81] International Organization for Standardization. *Open Systems Interconnection - Basic Reference Model*, August 1981. ISO/TC 97/SC 16 N 719.

[JCJO93] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Oevergaard. *Object-oriented Software Engineering: a Use Case Driven Approach*. Addison-Wesley, 1993.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June 1988.

[JLMS98] Mehdi Jazayeri, Rüdiger Loos, David Musser, and Alexander Stepanov. Generic Programming. In *Report of the Dagstuhl Seminar on Generic Programming*, Schloß Dagstuhl, Germany, April 1998.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. *ACM SIGPLAN Notices*, 27(10):63–76, October 1992.

[Joh97] Ralph E. Johnson. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[JS80] Anita K. Jones and Peter M. Schwarz. Experience Using Multiprocessor Systems - A Status Report. *ACM Computing Surveys*, 12(2):121–165, June 1980.

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

[KFG$^+$93] H. Kopetz, G. Fohler, G. Gruensteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schuetz, A. Vrchoticky, and R. Zainlinger. Real-Time System Development: The Programming Model of MARS. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 190–199, Kawasaki, Japan, March 1993.

[KHPS61] Tom Kilburn, David J. Howarth, R.B. Payne, and Frank H. Sumner. The Manchester University Atlas Operating System, Part I: Internal Organization. *Computer Jorunal*, 4(3):222–225, October 1961.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[KR84] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1984.

[Lar00] Special Section on Component-Based Enterprise Frameworks. In Grant Larsen, editor, *Communications of the ACM*, 43(10):24–66, October 2000.

[Lie96] Jochen Liedtke. On Microkernel Construction. *Operating Systems Review*, 29(5):237–250, January 1996.

[LMK89] Samuel J. Leffler, Marshall Kirk McKusick, and Michael J. Karels. *The Desing and Implementation of The 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[LT91] Henry M. Levy and Ewan D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software — Practice and Experience*, 21(1):77–90, January 1991.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with Abstract Data Types. *ACM SIGPLAN Notices*, 9(4):50–59, April 1974.

[Mah94] Umesh Maheshwari. Extensible Operating Systems. Area exam report, MIT Laboratory for Computer Science, Cambridge, U.S.A., November 1994.

[MCRL89] Peter Madany, Roy H. Campbell, Vincent F. Russo, and D. E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 311–328, Nottingham, U.K., March 1989.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive Plug-and-play Components for Evolutionary Software Developmen. In *Conference on Object Oriented Programming Systems Languages and Aplications*, pages 97–116, Vancouver, Canada, October 1998.

[MLTK97] Kim Mens, Cristina Videira Lopes, Bedir Tekinerdogan, and Gregor Kiczales. Aspect-Oriented Programming Workshop Report. In *Workshopes of the European Conference on Object-oriented Programming'97*, pages 483–496, Jyväskylä, Finland, June 1997.

[MS89] David R. Musser and Alexander A. Stepanov. Generic Programming. In *Proceedings of the First International Joint Conference of ISSAC and AAECC*, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

[MT86] Sape J. Mullender and Andrew S. Tanenbaum. The Design of a Capability-based Distributed Operating System. *The Computer Journal*, 29(4):289–300, 1986.

[MT00] University of Mannheim and University of Tennessee. *TOP500 Supercomputer List*, online edition, November 2000. [http://www.top500.org].

[NAS97] Numerical Aerospace Simulation Systems Division. *The NAS Parallel Benchmarks*, online edition, November 1997. [http://www.nas.nasa.gov/Software/NPB/NPB2Results/index.html].

[OKH$^+$95] Harold Ossher, Matthew Kaplan, William H. Harrison, Alexander Katz, and Vincent J. Kruskal. Subject-Oriented Composition Rules. *ACM SIGPLAN Notices*, 30(10):235–250, October 1995.

[OKK$^+$96] Harold Ossher, Matthew Kaplan, Alexander Katz, William H. Harrison, and Vincent J. Kruskal. Specifying Subject-Oriented Composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[OMG01] Object Management Group. *Common Object Request Broker Architecture*, online edition, January 2001. [http://www.corba.org/].

[Org72] Elliott Organick. *The Multics System: an Examination of its Structure*. MIT Press, Cambridge, U.S.A., 1972.

[Oxf92] Oxford University Press. *The Oxford English Dictionary*, second edition, 1992.

[PAB$^+$95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 267–284, Copper Mountain, U.S.A., December 1995.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[Pes97] Carlo Pescio. Template Metaprogramming: Make Parameterized Integers Portable with this Novel Technique. *C++ Report*, 9(7):23–26, 1997.

[PHW76] David Lorge Parnas, Georg Handzel, and Harald Würges. Design and Specification of the Minimal Subset of an Operating System Family. *IEEE Transactions on Software Engineering*, SE-2(4):301–307, 1976.

[Pik00] Rob Pike. Systems Software Research is Irrelevant. Online, February 2000. [http://cm.bell-labs.com/who/rob/utah2000.ps].

[PKC97] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages: Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2), June 1997.

[Pla95] P. J. Plauger. The Standard Template Library. *C/C++ Users Journal*, 13(12):20–24, December 1995.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1):11–32, January 1988.

[PPD$^+$95] Rob Pike, Dave L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom and. Plan 9 from Bell Labs. *Computing Systems*, 8(2):221–254, 1995.

[PT98] Loic Prylli and Bernard Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of the International Workshop on Personal Computer based Networks of Workstations*, Orlando, USA, April 1998.

[RAA$^+$88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4):305–370, 1988.

[RAB$^+$92] Trygve Reenskaug, Egil P. Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Nêss-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pâl Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-oriented Systems. *Journal of Object-oriented Programming*, 5(6):27–41, October 1992.

[RBLP91] James Rumbaugh, Michael Blaha, William Lorenson, and William Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[RFS$^+$00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, U.S.A., October 2000.

[Röm01] Sascha Römke. Ein XML-basiertes Konfigurationswekzeug für Betribssysteme am Baispiel EPOS. Studienarbeit, Otto-von-Guericke-Universität, Magdeburg, Germany, 2001.

[Rub97] Alessandro Rubini. *Linux Device Drivers*. O'Reilly, Sebastopol, U.K., 1997.

[Sam97] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer, 1997.

[San01] Sandia National Laboratories. *The ASCI Red SuperComputer*, online edition, March 2001. [http://www.sandia.gov/ASCI/Red/].

[Sch86] Wolfgang Schröder. *A Family of UNIX-Like Operating Systems – Use of Processes and the Message-Passing Concept in Structured Operating System Design,*. PhD thesis, Technical University of Berlin, Berlin, Germany, 1986.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 1996 Symposium on Operating System Design and Implementation*, pages 213–227, San Diego, U.S.A., October 1996.

[SGP98] Abraham Silberschatz, Peter Galvin, and James Peterson. *Operating Systems Concepts*. John Wiley and Sons, fifth edition, 1998.

[Sha48] Claude Elwood Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July 1948.

[Sma99] Yannis Smaragdakis. *Implementing Large-scale Object-oriented Components*. PhD thesis, University of Texas, Austin, U.S.A., December 1999.

[SP94a] Wolfgang Schröder-Preikschat. PEACE - A Software Backplane for Parallel Computing. *Parallel Computing*, 20(10):1471–1485, 1994.

[SP94b] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, Englewood Cliffs, U.S.A., 1994.

[SS95] Christopher Small and Margo Seltzer. Structuring the Kernel as a Toolkit of Extensible, Reusable Components. In *Proceedings of the 1995 International Workshop on Object Orientation in Operating Systems*, pages 134–137, Lund, Sweden, August 1995.

[SSPSS98] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

[Str86] Bjarne Stroustrup. C++ Programming Language. *IEEE Software (special issue on Multiparadigm Languages and Environments)*, 3(1):71–72, January 1986.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, 1997.

[Str00] Bjarne Stroustrup. Wrapping C++ Member Function Calls. *The C++ Report*, 12(6), 2000.

[SUN01] SUN Microsystems. *The Source for Java Technology*, online edition, January 2001. [http://java.sun.com/].

[SV98] Clemens Szyperski and Rudi Vernik. Establishing System-Wide Properties of Component-Based Systems: A Case for Tiered Component Frameworks. In *In Proceedings of the Workshop on Compositional Software Architectures*, Monterey, U.S.A., January 1998.

[Szy92] Clemens A. Szyperski. *Insight Ethos: On Object Orientation in Operating Systems*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1992.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-orineted Programming*. Addison-Wesley, 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[Ten00] David Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[TFC00] IEEE Task Force on Cluster Computing. *Cluster Computing White Paper*, Mark Baker, editor, online edition, December 2000. [http://www.dcs.port.ac.uk/˜mab/tfcc/WhitePaper].

[TG89] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Operating System Review*, 23(5):159–166, December 1989.

[THIS97] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer, April 1997.

[Tho78] Ken Thompson. UNIX Implementation. *The Bell System Technical Journal*, 57(6):1932–1946, July 1978.

[Tie88] Michael Tiemann. Wrappers: Solving the RPC problem in GNU C++. In *Proceedings of the C++ Conference*, pages 343–361, Denver, U.S.A, October 1988.

[Tie99] Gilles Pokam Tientcheu. Communication Strategies to Support Medium/Fine Grain Parallelism on SMP Workstations Clustered by High Speed Networks. Diplomarbeit, Technical University of Berlin, Berlin, Germany, December 1999.

[TR74] Ken Thompson and Dennis M. Ritchie. The UNIX Timesharing System. *Communications of the ACM*, 17(7):365–375, 1974.

[TTG93] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors: a Summary. *ACM Performance Evaluation Review*, 21(1):272–274, June 1993.

[Van97] Michael VanHilst. *Role-oriented Programming for Software Evolution*. PhD thesis, University of Washington, Seattle, U.S.A., September 1997.

[vECGS92] Thorsten von Eicken, David E. Culler, S. C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992.

[Vel95] Todd L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[VN96a] Michael VanHilst and David Notkin. Using C++ Templates to Implement Role-Based Designs. In *Proceedings of the Second International Symposium on Object Technologies for Advanced Software*, volume 1049 of *Lecture Notes in Computer Science*, pages 22–37, Kanazawa, Japan, March 1996.

[VN96b] Michael VanHilst and David Notkin. Using Role Components to Implement Collaboration-based Designs. *ACM SIGPLAN Notices*, 31(10):359–369, October 1996.

[W3C98] World Wide Web Consortium. *XML 1.0 Recommendation*, online edition, February 1998. [http://www.w3c.org].

[WBvE96] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Low-latency communication over fast ethernet. In *Euro-Par 1996*, volume 1123 of *Lecture Notes in Computer Science*, pages 187–194, Lyon, France, August 1996. Springer.

[Weg86] Peter Wegner. Classification in Object-oriented Systems. *ACM SIGPLAN Notices*, 21(10):173–182, October 1986.

[Wei95] David M. Weiss. Software Synthesis: The FAST Process. In *Proceedings of the International Conference on Computing in High Energy Physics*, Rio de Janeiro, Brazil, September 1995.

[WG92] Niklaus Wirth and Jurg Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, Reading, U.S.A., 1992.

[Wir71] Niklaus Wirth. Program Development with Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[WTS$^+$97] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, September 1997.

[Yok92] Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the ACM Conference on Object-oriented Programming Systems, Languages and Aplications*, pages 414–434, Vancouver, Canada, October 1992.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. Scheduling Tasks with Resource Requirements in Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, 13(5):564–577, 1987.