Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 117-139

SUBJECT-ORIENTED OPERATING SYSTEM DEVELOPMENT BASED ON SYSTEM PREDICATE CLASSES [1]

Á. Balogh and Z. Csörnyei

(Budapest, Hungary)

Abstract. Portable and scalable operating systems must satisfy very different needs on very different hardware architectures. Therefore they consist of several modules implementing different functionalities, and the actual system is composed of a subset of all these modules. The purpose of the modules may be very different, e.g. device drivers, security subsystems or user interfaces, but they work in the same computer system using the same hardware and software resources, so they must conform to predefined interfaces to cooperate with each other. However, these interfaces are usually defined to be very restrictive to avoid enormous complexity of the operating system. In this paper we describe a new approach for operating system design that allows a more flexible composition of modules without increasing the complexity of the system. Our suggested solution is the application of the *subject-oriented programming* [17] where different modules are considered as subjects which work with hardware and software resources as objects. Objects are instances of *system predicate classes*, our recently published technology for effective application of object-oriented language elements in system programming. We also suggest implementation techniques avoiding performance loss.

1. Motivation

A modern operating system is no longer a single and uniform software product: it must serve very different needs, most of their criteria contradicting. There-

[1]Supported by the Hungarian Ministry of Education under Grant GVOP-3.2.2.-2004-07-0005/3.0 ELTE IKKK

fore the system software has to be flexible enough to be executed in different hardware environments and to match specific requirements. Hardware environments cover a large scale from personal mobile devices or embedded systems through desktop computers to large database servers, while specific requirements vary from maximal efficiency to top-level security. To achieve such flexibility, the operating system consists of more or less tightly cooperating low-level software components, such as device drivers, security services, user interface drivers etc., most of them optional or interchangeable.

A classic operating system structure is the so called multi-layer kernel, where the software is divided into layers, with well defined interfaces between them. The lowest layer usually consists of device drivers, customizable to match the underlying hardware architecture. Similarly, the user interface in the topmost layer is replaceable as well. Also intermediate layers contain flexible parts, such as file system drivers which work on top of mass storage device drivers. However, forcing operating system components into layers restricts the flexibility of the system. Beyond a specific point of scalability, there is no good order of layers, the different components have to mutually cooperate to provide the requested services, and layering them would restrict this cooperation. Furthermore, multi-layer operating systems have poor performance because of the restricted protocols between their components. This is the main reason why multi-layer operating systems are hardly ever used in practise.

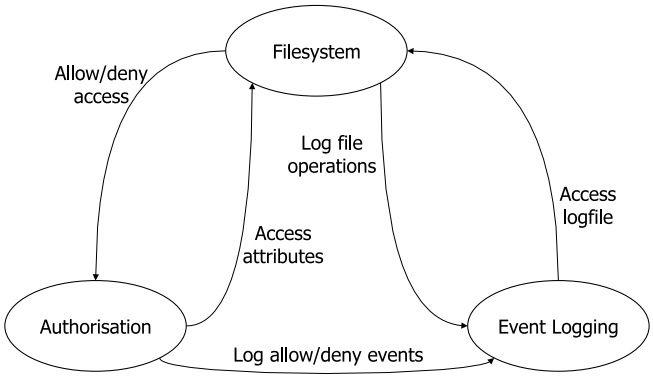

Figure 1. Cooperation between three components of an operating system

File-access is a good example to show the weakness of multi-layered kernels, as it involves several components depending on the actual security policy. The file system is handled by the file system driver, which is the main component responsible for the file-access. A modern operating system usually supports more different file systems (e.g. *Linux* supports all popular file systems on PCs), thus it

contains more file system drivers. In multi-user operating systems access to files is controlled by access rights. However, different security models (e.g. DAC, MAC etc.) use different authorization strategies, which is implemented by a specific, often interchangeable part of the operating system, the authorization subsystem. This module is independent of the file system, but is able to cooperate with it in different file operations, e.g. opening for reading, writing, file deletion or renaming. The file system device driver does not have to deal with the security model in use, but must be able to handle the file in the actual file system. Similarly, the authorization subsystem does not need to know the details of the underlying file system, but is responsible for decisions whether a specific file access mode is allowed on a given file. It is obvious, that the two components (file system driver and authorization subsystem) cannot be arranged in layers, since the file system makes the authorization subsystem check for permission, while the latter queries the file system driver for file attributes. There may also be a third component, an event logger or auditing subsystem, which is of course optional as well, since some systems e.g. mobile devices do not have enough resources for logging, and also do not need to do it. The event logger may log specific kinds of events in the system, thus its services are used by both the file system and the authorization component which invoke its routines for logging allowed and denied file accesses as well as operations on the file. However, the log itself is written into a file, thus the event logger uses the services of the file system as well. Figure 1 shows the ways of cooperation between them. It is obvious, that these three components of the operating system cannot be placed on top of each other.

Modular software design allows more complex structuring of program components than layering. Modules can provide services to each other mutually, the connection between them is described by interfaces. Every module can be replaced by another one, which conforms to the same interface, thus the modular structure is flexible as well.

In an open source system, the modules may be provided by the programming language. It is up to the user to select the necessary modules, and make (compile and link) the kernel of them. Similar method was applied in early *Linux* kernels (before version 2.0) and in the various *BSD* kernels as well. The main drawback of this solution is that it requires from the user special skills that many do not have. Another issue is that kernel recompilation must be followed by complete system restart, which cannot be applied in systems supposed to run continuously.

A more user-friendly solution is the application of run-time modules, which are precompiled, but can be linked to the kernel in run-time. The core operating system contains hooks, which are function calls to invoke a subroutine in a module. Some of these function calls are optional, if the module is not present, they invoke an empty function. This solution is implemented for example in *Linux* kernels version 2.0 and later.

However, these classic kinds of modularization (both programming language and run-time) are too restrictive, since the fixed places of the hooks in the code of the core system limit the set of the services that can be performed by the modules. For example, a *Linux* security module deciding whether an operation is allowed on an object is called after a successful DAC check, which is part of the core system. If the DAC check fails, thus it denies the operation to be performed the security module does not have the possibility to override this decision, because the hook is placed after the built-in check.

The application of the object-oriented paradigm for operating system development is a new technology, still in experimental phase, but very promising. However, it does not address our problem at all, since objects represent rather resources than services. For example, object is a file, with its own operations for opening, closing, reading, writing etc., but neither the whole file system, nor the authorization nor the auditing subsystem.

The solution which we suggest in this paper is based on the paradigm of *subject-oriented programming*. Although the technology was originally invented for high-level application development, the concepts behind it are also applicable for building of flexible operating systems. In a subject-oriented operating system, subjects replace modules, while objects represent the resources similarly to object-oriented operating systems. These objects are instances of *system predicate classes*, a recently introduced language tool that allows low-level data structure types to be handled as classes. The operating system itself is composed of different subjects, customized to the actual needs. The composition is performed on subjects in object-code, thus no recompilation is needed to change the structure of the system. The various composition rules ensure maximal flexibility and scalability of the system.

The remaining of the paper is organized as follows. In Sections 2 and 3 we overview the main concepts of *subject-oriented programming* and *system predicate classes*. Section 4 describes, how these concepts can be applied for design of an operating system. In Section 5 we present the most important composition rules to be used in a subject-oriented operating system. Section 6 introduces a new term: *subject interfaces*, to further increase the flexibility of our proposed technology. Suggestions for efficient implementation are given in Section 7. Finally, Section 8 discusses related work while Section 9 concludes.

2. Subject-oriented programming – Overview

In the general *subject-oriented programming* paradigm, the state and behaviour of the objects is shared among multiple program components, the so called subjects. This is one of the main differences compared to the classic

object-oriented paradigm, where the whole object is defined in a single module. Another very important difference is, that subjects are executable as standalone programs as well, thus the partial definition of their objects are complete in themselves, they do not explicitly require services provided by other subjects. However, when two or more subjects are composed together, the partial definitions of the same object are merged. This composition is done by the subject composer, a tool that works on the target code, thus no recompilation is needed. A large variety of composition rules is available to determine the representation and the semantics of such composition. The basic elements of subject oriented programming are shown on Figure 2.

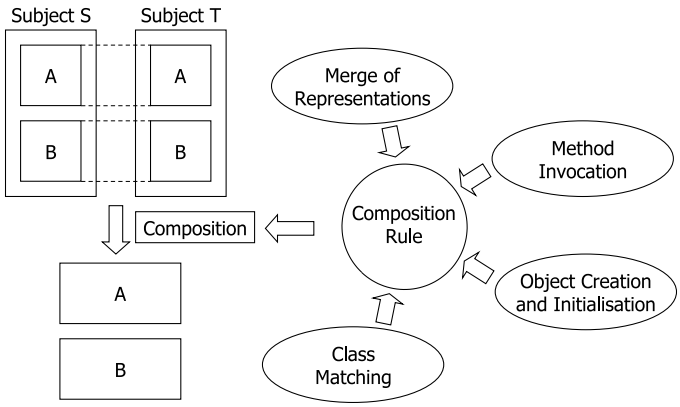

Figure 2. Overview of the *subject-oriented programming* paradigm

2.1. Subjects and objects

In the model being used, a subject is a collection of state and behaviour specifications, a perception of the world by a particular application or application component. They are not classes or objects, but describe some of the state and behaviour of them. The actual objects are a composition of these partial definitions. The objects do not have intrinsic state and behaviour, all properties are defined in various subjects. Subjects operate separately on these shared objects, without knowing the details associated with those objects by other subjects.

An executing instance of a subject is called subject activation. A cooperating group of subjects is called subject composition. The composition rule determines the representation, state and behaviour of the objects composed by multiple definitions in the different subjects.

Similarly to *object-oriented programming*, objects are members of classes.

Classes may inherit from each other, composing class-trees. These class trees may be different in different subjects even for the same class. The state of an object is retained in state variables, while behaviour is specified by the means of methods.

2.2. Interactions among subjects

Although subjects are standalone running components of the software, working independently of each other, they are not completely isolated. In the *subject-oriented programming* paradigm various forms of interactions may occur between subjects:

- a request for function or state change to be supplied by another subject;
- performance of an activity in which another subject might participate;
- notification of an occurrence which may be of interest to another subject;
- use of subject's behaviour as part of the "larger" behaviour of another;
- sharing of state.

The first four of these forms describe a shared behaviour of the object between subjects, while the last one a shared state. Shared state can be implemented as shared representation, thus affected instance variables must be mapped to the same memory location. However, to implement shared behaviour, the composition rule defines the semantics of shared method invocation.

2.3. Composition rules

A composition rule describes how the object's representations defined in different subjects are to be merged to one single representation, and the semantics of invocation of shared methods. A method call in one subject may trigger invocation of a method with the same name of the very same object in another subject. The composition rule describes the exact semantics how this situation must be handled.

The simplest composition rule, called *merging* is the following. The method is dispatched for each subject activation in arbitrary order (but not in parallel). In each subject the local inheritance graph is used for method dispatching. If the method returns values, they must be all identical or an exception is raised. If the method has in-out parameters, they may be set by one method and used by the next. However, the operations performed on them should be commutative.

An alternative composition rule, called *nesting*, treats compositions as scopes, and allows calls to propagate beyond scopes only if specified explicitly. This composition rule does not handle subjects at the same level, but one subject is nested into the other one. By using this rule subsequently for more subjects,

a nesting-tree evolves, where subjects in the lower levels are nested into their parent. In such trees, invocation of a method is dispatched downward in the tree from the node where it was called. Additionally, a subject at a given level may import a call from one of its children. In this case, the call is dispatched to the remaining children of the importing subject as well. The import can also be done successively, thus the parent of that subject may also import that call, so the dispatch can be propagated even to the top of the tree. However, in the absence of an explicit import the dispatch of the method invocation is restricted to the particular subtree.

2.4. Object creation and initialization

Similarly to the operation invocation, object creation and initialization is shared as well. First, a common object identifier is reserved. Then each subject allocation determines which class the object belongs to and reserves space for its state. Finally, initial values are written in the fields.

For this last step, initialization of the state of the object the composition may choose from two approaches. First, called *immediate initialization* fills default values to the fields immediately upon creation. Second possibility, *deferred initialization* does it first when the subject responds to an operation request of that object.

2.5. Class matching

Different subjects classify the same objects differently. It may also happen, that a subject has not classified an object at all. In this case, upon invocation of an operation, the subject must decide which method to call. There are many possible approaches to perform this classification. However, as they do not play any role in our case, where objects are instances of predicate classes (see below), we do not discuss them in this paper.

3. System predicate classes – Overview

Predicate classes are a special approach to the concept of classes in *object-oriented programming*, where instead of an external type identifier or a pointer to the virtual method table a Boolean function on the state of the object determines its dynamic type. This allows an object's representation to contain only the fields explicitly defined by the programmer. Furthermore, *predicate classes* make possible for an object to change its dynamic type if its state is modified implying the change of the truth value of its classifying predicate.

System predicate classes are *predicate classes* with fixed representations, able to store system data structures, e.g. registers, hardware-defined data structures etc. They may contain special fields, denoted as *reserved*, which may be overlapped in their descendant classes. This enables *polymorphism-by-value*, which means that a variable of the type of the superclass may store an instance of its subclass as well. Most high-level *object-oriented* languages allow such polymorphism only for the reference, not for the variable itself. Dispatcher methods evaluating the predicates and jumping to the appropriate method body are generated during link time to enable modular software development. These jumps eliminate long call-chains, giving the technology very good performance results, since their time costs contain only the execution time of the simple jump instructions.

System predicate classes allow four different kinds of single inheritance depending on whether there are new fields overlapping reserved fields or new fields at the end of the object, while multiple inheritance permits almost arbitrary combinations of these four kinds. A summary about them is presented on Figure 3.

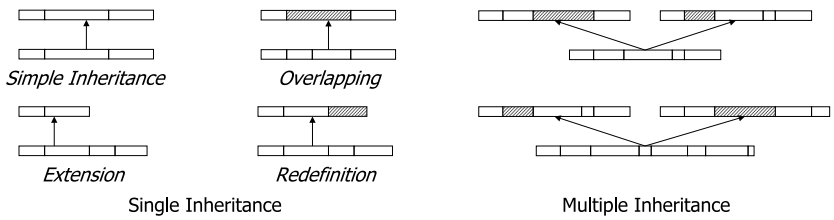

Figure 3. Overview of the various kinds of single and multiple inheritance of *system predicate classes*

4. Low-level subject-oriented design

The subject of this paper is the adoption of the above presented *subject-oriented* paradigm for operating system development. It is obvious, that optional and interchangeable operating system component may be considered as subjects, while system resources, such as various descriptors, memory pages, files, processes etc. can be represented as objects. However, there are several special requirements in operating system programming, which do not play role in high-level application development. This implies that the above described technology cannot be used in system programming without change.

The special requirements in operating system development can basically divided into two main groups. First, operating system is executed on the raw hardware. It cannot make use of an abstract virtual machine, run-time environment or standard library. It has to handle physical addresses, fixed representations, must access specific memory locations, ports and registers. High-level object identifiers or references, compiler created abstract representations cannot be used. Both compiler and subject composer must thus omit these high-level elements. This implies the introduction of several restrictions which we discuss further below.

The other group of special needs is related to the performance of the system code. An application programmer may decide against efficiency in favour of other properties more important in the actual situation, but an operating system must always provide as high performance as possible. Otherwise it would restrict its usability for applications where good performance is not a goal. Thus a general-purpose operating system must be efficient, which results in additional restrictions and modifications of the original paradigm.

In spite of the restrictions we have to introduce, we also have to retain most of the features of *subject-oriented programming* to benefit from it. These features include that the executing software – the operating system – is composed of precompiled subjects. Thus subject composer has to work on object code, which is one of the most challenging problems which must be solved efficiently.

4.1. Technology overview

A *subject-oriented* operating system consists of subjects, composed by predefined composition rules. Subjects are operating system modules, all of them interchangeable, and many of them optional. Typical optional subjects are auxiliary device drivers, security modules, while mandatory subjects include memory managers, process schedulers etc. The subjects are available in object code, no recompilation is needed when changing the system by replacing, inserting or removing subjects. The composition rule itself may also be changed at the operator's decision.

Subjects are not arbitrary standalone programs supplied in object code. They must be ready for composition with other subjects which requires special compilation and additional information issued with it. Although the technology itself is programming language-independent, the subject-oriented development tool has to support the language the subject is written in. It must include a preprocessor which executes the necessary modifications on the code and creates an interface (e.g. header file) of the subject in a syntax interpretable by the subject-composer. The subject is then compiled by the standard compiler of the programming language. Thus a subject ready for composition consists of the object file and the

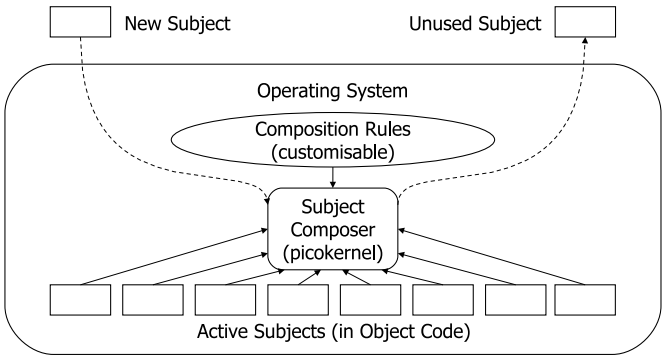

Figure 4. Technology overview of low-level subject-oriented design: the fixed part (picokernel) of the operating system is the subject composer composes new subjects into the system and removes unused ones in runtime based on user-customizable composition rules

generated interface.

4.2. Data representation

Operating systems work on data structures provided by the hardware's manufacturer, having fixed representations. This restricts both the partial definition of the classes and the subject composer, since the composed objects have to match these representations. It follows that subjects must exactly determine the offset of all fields of a class, thus reserving space for fields defined by other subjects. This is a restriction to the technology, since in the original concept subjects do not need to have any knowledge on the parts of the object defined by other subjects.

To define a (partial) class in a subject, the programmer has to determine the exact places of the fields within the object. The gaps between must be filled by reserved fields, which are basic elements of any language based on *system predicate classes*. Reserved fields allow the subject composer to compose classes together in a way, that fields overlapping each other are either the same, or a non-reserved field overlaps only reserved fields. If this is not the case, the composer returns error and rejects composition of the subjects containing the ambiguous definitions of the same class.

This latter requirement may perhaps seem to be too strict if we think on fields which are to be interpreted differently in various situations, handled by different subjects. However, our goal is to provide a tool for safe programming, and

allowing of arbitrary composition of contradicting representations would increase risk of accidental programming errors. That is the reason why we decided to introduce this restriction. However, using system predicate classes this particular problem can be solved efficiently by definition of the affected field of the class as reserved, and inherit different subclasses of it in the different subjects, overlapping the field.

Since the programming language used for the development of the subjects uses predicate-based inheritance, the composer also requires that predicates in different subjects for the same subclass are either the same or independent of each other. As independence of Boolean expressions is too complicated to check, typical requirement is that they do not contain expressions for the same fields of the class. This implies that the subject composer implements a kind of multiple inheritance across the different subjects, even if the language itself only supports single one.

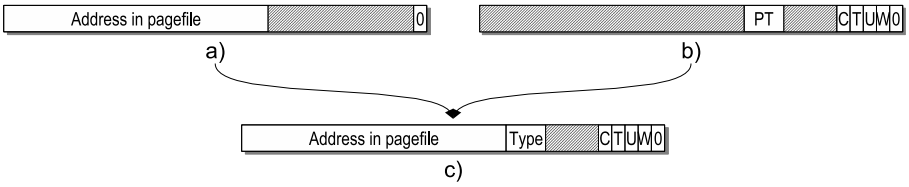

Figure 5. Composition of class for non-present page table entry defined in the paging file handler subject (a) and in the memory manager subject (b) of a subject-oriented operating system

Figure 5 shows the representation of a class for non-present pages on the *Intel x86* processor architecture in two different subjects and their composition. The last bit is always zero, since non-present page inherits in both subjects from general page with a predicate determining the last bit (present bit) as zero. Our first subject, which handles the paging file does not need any other information on the page than its address in the file (first 20 bit). However, the memory manager does not care where the page is actually stored, instead, it has to deal with other information on the actual page: its type, whether it is cached (C), whether write-through caching is enabled (T), whether it is accessible from user privilege-level (U) and whether it is accessible for writing (W). As these fields do not overlap each other, the two classes can be composed together. Other paging file-handlers and other memory managers may use different representations for non-present pages, but the fields they define may not overlap each other, except if they have the same name and type.

A special situation is, when partial definitions of the same class have different

size in different subjects. If this is the case, the subject composer extends the shorter class with reserved fields to match the size of the longest one. The most extreme case is when one subject does not define fields for a class at all, only methods: such classes match any other one. However, methods of these classes cannot access any of the fields of the object directly, only through other methods, defined in other subjects. Thus subjects defining classes without fields depend on other subjects, which contain methods accessing these fields. These subjects cannot be run as standalone programs.

A typical example for the case described above is an operating system containing a file system driver subject and an authorization subject. The latter one does not have any information about the actual data structures of the underlying file system, thus its class representing a file contains no fields at all. Instead, it asks the protection fields through private methods of the same class, but defined in the file system driver subject. This makes the authentication subject dependent of the file system, but the file system is able to run without the authentication subject, of course by providing zero security.

4.3. Method invocation

Unlike composition of the object's representation, which is always the same, the semantics for method invocation across subjects are defined by the actual composition rule. Always the same rule applies for all methods of all objects in the composed subjects.

Basically, there are three types of methods within a subject. A method is local, if there is no method with the same name and parameter signature for the class in other subjects with which the subject is composed. However, locality turns out only at composition, thus at (pre)compilation time no method can be handled as local, as we do not know what methods other subjects define for the same class.

A method of a class is common, if it is defined with the same signature in multiple subjects. This is the case, where the composition rule defines the semantics of the actual invocation.

The last case is a method, which is called in one of the subjects, but is not defined locally. This method is called external, and makes the subject dependent of another one, which contains definition of the missing method. This does not happen arbitrarily, the composition rule must explicitly declare dependency and must be checked at composition time. As the compiler does not have any information about the actual composition rule, it has to compile any subjects containing calls for missing methods. However, the subject composer has to signal error when finding missing methods without explicitly declared dependency.

4.4. Object creation and initialization

As we already mentioned earlier, an operating system is executed on the raw hardware, thus it has no language runtime environment dealing with objects identifiers in a common abstract pool. Furthermore, objects are not always created dynamically, they are often static system data structures. This results in a quite different object creation process from that what we have seen in high-level *subject-oriented programming*.

The most typical case is, if an object already exists somewhere in the memory, for example in a system table, e.g. a segment descriptor in a descriptor table. Our subject just creates a pointer to point to this object. No allocation happens, no state initialization and no constructor is called.

If an object is defined as a local variable, its initialization happens at the beginning of the block. First, where defined, default values are assigned to the fields. Then the appropriate constructor is called, depending on the parameter list given at the variable's declaration. As constructors are methods, similar rules apply for constructor invocation as for method invocation, but with some differences. As we already described, normal methods are distinguished on their parameter lists. If there is no method with the same name and same parameter list in one subject as the called one, no method is invoked in that subject, regardless of the composition rule. However, in case of constructors, the default constructor of the class is called in the subjects not containing constructor with the same parameter list as in the calling subject. Another difference is, that constructors cannot be external: a constructor with a formal parameter list matching with the actuals of the call must exist in the same subject. At dynamic allocation of objects similar things happen, but at the exact place of the allocation and not at the beginning of the block.

4.5. Class matching

Having general classes, it is difficult to determine which class an object belongs to, if its explicit class does not exist in that particular subject. Several solutions exist, but none of them is perfect, the area is a subject of research. However, the problem becomes much easier when using predicate classes, as predicates determine classification of the object.

Whenever the composition rule triggers invocation of the method in another subject, the class-tree in the original subject is searched upward for the first common class with the other subject. Then the method is dispatched in that class, using the predicates, as in the standard, non *subject-oriented* case.

Figure 6 presents a simplified example for class matching. An instance of

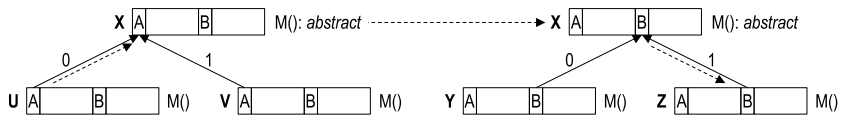

Figure 6. Direction of search at class matching

class U (thus containing zero in the field A) in the left subject, which stores one in the field B calls method M(). Let us suppose that the composition rule orders that M() has to be called in the right subject as well. However, class U does not exist in the right subject. To search for the appropriate class, and thus the appropriate method body, the class-tree of U is searched upward. Since class X, the parent class of U exists in both subjects, the method is dispatched in class X in the right subject. As we already mentioned, field B of our object is equal to one, thus method M() in class Z is executed.

5. Composition rules

Composition rules play their main role in the definition of the semantics for method invocation. The same rules apply for invocation of constructors as for invocation of normal methods, apart from the differences described above. Composition rules may also define dependencies.

A subject is dependent of another one, if one of its classes contains a call for an external method, thus a method which is not defined in that subject at all. In this case, the subject containing the call depends on any subject that provides the invoked method. To prevent programming errors, such dependencies must be defined explicitly in the composition rules in the form of an interface definition, which describes all the external methods the subject needs. If no dependency is defined in the composition rule or the interface does not describe an external method, the composition fails with an error message. Dependency and semantics of operation invocation are completely independent.

If the programming language and the actual system allows arguments to be assigned to subjects at their activation, multiple activations of the same subject may appear in a composition. Two completely identical subjects cannot be composed together, any attempt to it is considered a programming error.

5.1. Merging

The simplest and most straightforward composition rule *merging* can be used in *subject-oriented* operating system development as well. Merging always involves two subjects in a determined order, more subjects can be merged by subsequently applying the rule multiple times on the previous compositions. No single subject can be merged with more others in one single step. If two subjects are merged, the bodies of their common methods are executed sequentially after each other. Since we do not have any runtime environment but working on the raw hardware, we cannot check the commutativity of the method bodies, so we require a strict definition of the sequence of their invocation. This means that merging two subjects is not commutative either, the composition rule must determine their exact order, which is always independent of which subject has initiated the call, but in every case the method in the first subject is executed first.

5.2. Conditional merging

Similarly to standard *merging*, *conditional merging* is also applied on two subjects. However, for common methods returning discrete values its semantics is different. When calling such a method, the method body in the first subject is executed first, then its return value determines whether the body in the second one has to be executed as well. The condition is either positive or negative: in the positive case the second one is executed if the first one returned nonzero, in the negative case it happens if the return value of the first one is zero. Similarly to standard merging, one subject or one composition cannot be merged simultaneously with more others, thus conditional merging cannot be used to branch on the return value of methods.

An example, where conditional merging can be applied in operating system development is the composition of a file system driver subject with the authorization subject. As we already mentioned, this latter subject depends on a file system driver subject, since it handles several protection attributes but does not have information about their physical representation, since they vary from file system to file system. However, dependency does not determine the composition rule, since it is only related to external methods. The authorization subject has to guard file system operations, thus it has to contain methods for opening the file in different modes with the same parameter signature and return value as the file system driver. This subject is conditionally merged with the file system in such a way, that the return value of the authorization subject determines whether a file operation is allowed. For example, they both return Boolean, thus positive conditional merging is the appropriate composition rule. If the original caller of

the method gets true as return value, the file system operation was completed successfully. If it gets false, the operation failed either on access rights, thus authorization subject returned false, or on any other reason, which is the case if the file system itself returned false. To signal the exact kind of failure, the methods can either use a global error flag, similarly to POSIX's *errno*, or return an integer or enumerated type instead of Boolean, with a negative conditional merging.

5.3. Nesting

Unlike *merging*, *nesting* is a tree-like composition structure, similarly to that what we have seen at high-level *subject-oriented programming*, thus it is allowed to nest multiple subjects into another one. However, one single subject cannot be nested into more than one subject. In this case, the semantic of method dispatching is not symmetric, thus it depends on the place of the invocation. When a method is called in the nesting subject, it is dispatched to its nested subjects. However, if the call is initiated in the nested subject, it is only delegated to its nesting subject if it imports it explicitly. Then, it is dispatched to its other nested subjects, so the total scope of the call is a subtree of the nesting tree.

The order of execution in the affected subtree of subjects may be individually set for each nesting-nested pairs. If it is *pre-order*, the method in the nesting subject is executed first, while with *post-order* the method in the nested one. One single subject can nest more others with different orders of execution. In this case, first the methods are executed in the nested subjects with post-order nesting, then the one in the nesting subject, and at last the methods in the nested subjects with pre-order nesting. The order of execution of methods in the nested subjects with the same ordering rule is undetermined.

For the continuation after the execution of the method in the nested subject there are two possibilities as well. When *deep-first* order is defined, then the execution is continued with the parent or the child depending on the actual rule (pre- or post-order). When *level-first* order is defined, then the next subject is the sibling of the actual one.

5.4. Conditional nesting

Conditional nesting is the most general composition rule. It is a mixture of conditional merging and nesting. The determination of the involved subtree of subject is identical with the procedure we presented at standard nesting. However, execution of methods depend on a condition similar to conditional merging. Using the pre-order scheme, methods in the conditionally nested subjects are

executed only if the return value of the nesting subject is appropriate. This allows branches between subjects, which would be impossible with the linear conditional merging. When post-order is defined, the nested subjects are called first, and the execution of the method in the nesting subject depends on their return values. There are two possibilities: method in an *and-nesting* subject is executed, if all nested subjects returned the appropriate return value (i.e. zero or non-zero, specified for all nested subjects individually). On contrary, *or-nesting* subjects execute their method body if there is at least one nested subject whose method returned the expected result. Subjects can be nested in another one in a mixed way of standard and conditional nesting. In this case, methods in standard nested methods are always executed, and their return values at post-order invocations are not considered.

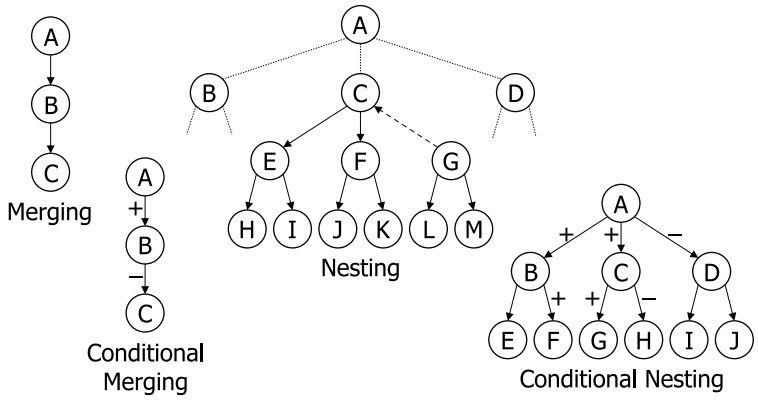

Figure 7. Examples of composition rules

Figure 7 shows small examples of the mentioned composition rules. Note that merging, conditional merging and nesting are all special cases of conditional nesting. The reason for introducing them was to make often used compositions easier.

6. Subject interfaces

In an operating system there may be services which can be provided by different subjects. For example, the system can offer more memory managers, from which we have to select the one best suitable to our needs. It may offer different paging file handlers as well, each independent of the actual memory manager. In such cases, the composition rule must also be changed, when one of these

subjects is replaced at next reboot.

The problem becomes more complex, when multiple subjects providing similar services are allowed to be loaded simultaneously to the system. For example, the operating system may support several file systems, each implemented by a different subject. However, authorization rules are the same in every file system. Since the authorization subject is not allowed to be merged with or nested into more than one file system, the only solution would be to nest the file system into the authorization subject. This, however, would break the structure of the operating system and make compositions unnecessarily complicated.

To solve these problems, we introduce a new term, *subject interface*. A subject interface is similar to an interface in high-level programming languages such as *Java* or *C#*, but instead of classes, subjects implement it. It contains interfaces for classes, to be defined in the subject implementing the subject interface. Subject interfaces can be used in composition rules instead of subjects. In this case, the rule defines the multiplicity of the implementing subjects as well. *Single* interfaces are implemented by at most one active subject. Typical single interfaces are process schedulers, memory managers, authoriser subsystems etc. *Multiple* interfaces may be implemented by multiple active subjects simultaneously. Typical application for multiple interfaces are file systems and various device drivers.

7. Implementation techniques

As we already discussed, performance is one of the most important issues when dealing with operating system code. The key for success of a new technology in the operating system programming is the possibility of efficient implementation. That is the reason why new paradigms invented for application development could hardly penetrate into the operating system development, instead still modern operating systems are programmed procedurally in standard *C*.

System predicate classes are intended to bring *object-oriented* programming into the lowest level of software development, thus enable efficient implementation of *object-oriented* operating systems. The key for the good performance of this new technology lies in the dispatcher subroutines, which are called normally, but after evaluating the actual predicate they jump into the appropriate method body directly. This results in the very low cost of a simple jump instruction on the actual processor as the total overhead of the technology. In this section we present how similar dispatcher subroutines can be used for *subject-oriented* method compositions.

As every method is potentially composable with another one in another subject, the straightforward way would be to use indirect addressing at each method

call, to be able to change the starting address of the method to the starting address of the dispatcher in case of composition. However, this would have a large negative impact of the performance. To prevent this, we generate a map on all the method calls during compile time and use direct calls, which results zero overhead. The composer then replaces every address for the actual methods in the map for the dispatcher. Although changing of the addresses may be slow, subject loading and unloading happens much more rarely than ordinary method calls, thus the general performance is higher than if we would use indirect addressing everywhere.

Composition of methods depends on the actual composition rule. Standard merging is very simple, no real dispatcher is needed: the called method is the first one which jumps at the end into the body of the second one. Since unconditional jump skips unnecessary stack handling, its execution time is much lower. Similarly, conditional merging can be solved without real dispatcher as well, by testing the return value and the conditionally jumping to the second method body. This must happen before the return instruction, which is executed only if the condition does not hold. The only trick in these solutions is that some space has to be left before the return instruction for the test and the conditional jump, which is jumped over if the method is not composed together with another one.

Nesting is much more complicated than merging, it needs a real dispatcher. However, since arguments of the methods are the same in this case as well, overhead of stack handling can be prevented by using conditional and unconditional jump instructions similarly to the dispatchers of *system predicate classes*.

The last thing to note at implementation is the work with subject interfaces which can be implemented by multiple subjects. In this case, when in composition, indirect addressing cannot always be prevented since one single subject is merged with or nested into multiple subjects through the interface. These interfaces are thus more expensive than single interfaces or direct composition of subjects.

Using this implementation technique, the general performance of a *subject-oriented* operating system is similar to that of a modular one, since the technique for loading new modules into the system is similar to that for composing new subjects to the existing ones. The main difference is, that a *subject-oriented* operating system contains more hooks for connecting the new operations than the modular one, where subroutines of new modules are called at fixed points of the code. However, these more hooks hardly decrease the performance of the system.

8. Related work

8.1. Operating systems supporting modules

Open source operating systems, e.g. *Linux* [19], [4], *FreeBSD* [18], *NetBSD* [20] and *OpenBSD* [21] belong to the most customizable systems. In their early versions this was achieved by providing everything in source code, and forcing the user to configure it according to the actual needs then recompile it. Later loadable modules were introduced that allowed customization of compiled code by loading the appropriate modules. In newest versions of their kernels, most functions are available as loadable modules.

A special family of modules, enabling high-level scalability of system security are *Linux Security Modules* [16] by *Wright*, *Cowan*, *Morris*, *Smalley* and *Kroah-Hartman*. The kernel contains hooks at given places to call functions of these modules, whenever they are present. However, *Linux Security Modules* can primarily be used to further restrict access of specific object, but hardly ever to permit access of objects inaccessible by default. Furthermore, security modules were only designed to guard objects, so their availability for auditing is limited as well.

A more general, but lower level alternative of *Linux Security Modules* are process encapsulation, allowing processes to be run in unique environments, simulating a virtual world for each process, where object access rights can be overridden in any direction, and auditing is possible as well. Among many implementations, one of the most general ones is process jailing [11] by *van 't Noordende*, *Balogh*, *Hofman*, *Brazier* and *Tanenbaum*.

8.2. Subject-oriented programming

Subject-oriented programming (*SOP*) [17], [12], [6], [13], [14], [7], [9], [10], [15], [8] and [5] by *Harrison*, *Ossher* et al. is an advanced programming paradigm for software development. Its main goal is to further improve object-oriented application development by allowing software components, called *subjects* to partially define objects, and later composing them together by a special software tool. The main difference between original *SOP* and that described in this paper is the abstraction level. Since the original one was invented for high-level application development, it works with hidden abstract data representation in a high-level running environment. Contrarily, the tool proposed in this paper allows full control of the representation of the objects, and is able to run on the raw hardware.

8.3. System predicate classes

System predicate classes [3] and [1] were introduced by the authors to enable several features of object-oriented programming at the lowest level of software. With the help of overlappable reserved fields, predicate-based inheritance and polymorphism-by-value they provide a powerful *OOP*-like tool that can be used even for operating system development. An application of the technology for the programming language *C* is *SysObjC* [2].

9. Conclusion

In the paper we presented a new technology that allows development of highly flexible operating systems having good performance, while not requiring advanced skills from users, like e.g. kernel compilation. To achieve this, we combined two existing technologies, *subject-oriented programming* and *system predicate classes*. While the first one was originally intended to use in high-level application development, the latter one was especially invented for system programming. As *subject-oriented programming* is primarily based on *object-oriented programming*, which is a high-level paradigm as well, *system predicate classes* allow using most of its features, e.g. classes, inheritance, virtual methods, encapsulation, polymorphism also at system level.

The main benefit of our suggested technology is that it allows flexible combination of operating system components, which results in a highly configurable and customizable operating system, able to run in various platforms from embedded systems to large scale servers and suitable for very different applications from real-time systems through multimedia to database management. The main difference between a subject-oriented operating system kernel and a modular one is that the latter has a fixed core, only configurable through fixed parameters or by recompilation, while in a subject-oriented system every part of the software is replaceable. Further difference is, that modules have fixed hooks where their functions are called, while subject-oriented composition rules may compose any method bodies together. The main benefit of our technology is that it achieves this flexibility without significant performance loss, when using the implementation techniques that we suggested.

Unfortunately, beside these benefits our technology has a drawback as well: it cannot be used in existing operating systems. Subject-oriented paradigm requires a complete new design of an operating system. However, we hope that because of its promising benefits, operating systems that make use of our technology will appear in the next couple of years. To help this, we also plan to develop an experimental system, that allows us to further improve the presented technology.

References

[1] **Balogh Á. and Csörnyei Z.,** Multiple inheritance of system predicate classes, *Pure Math. Appl.,* **17** (3-4) (2006), 205-227.

[2] **Balogh Á. and Csörnyei Z.,** SysObjC: C extension for development of object-oriented operating systems, *PLOS 2006, San José, CA, 2006*, ACM Press, Article No. 5.

[3] **Balogh Á. and Csörnyei Z.,** Objects and polymorphism in system programming languages: A new approach, *Periodica Politechnica Ser. Electrical Engineering* (to appear)

[4] **Bovet D.P. and Cesati M.,** *Understanding the Linux Kernel*, 3rd edn., O'Reilly, 2005.

[5] **Clarke S., Harrison W., Ossher H. and Tarr P.,** Subject-oriented design: Towards improved alignment of requirements, design and code, *OOPSLA '99, Denver, Colorado, USA*, ACM Press, 1999, 325-339.

[6] **Harrison W. and Ossher H.,** Subject-oriented programming (a critique of pure objects), *OOPSLA '93, Washington D.C., USA*, ACM Press, 1993, 411-428.

[7] **Harrison W., Kilov H., Ossher H. and Simmonds I.,** From dynamic supertypes to subjects: A natural way to specify and develop systems, *IBM Systems Journal*, **35** (2)(1996), 244-256.

[8] **Harrison W., Ossher H. and Tarr P.,** *Using delegation for software and subject composition,* Research report RC 20946, IBM Thomas J. Watson Research Center, 1997.

[9] **Kaplan M., Ossher H., Harrison W. and Kruskal V.,** Subject-oriented design and the watson subject compiler, *OOPSLA '96 Subjectivity Workshop, San Jose, CA, USA*, ACM Press, 1996.

[10] **Mili H., Ossher H. and Harrison W.,** Supporting subject-oriented programming in Smalltalk, *TOOLS USA '96, Santa Barbara, CA, USA*, Prentice Hall, 1996.

[11] **Noordende G., Balogh Á., Hofman R., Brazier F. and Tanenbaum A.S.,** *A secure and portable jailing system*, Technical report IR-CS-025, Vrije Universiteit, Amsterdam, 2006.

[12] **Ossher H. and Harrison W.,** Combination of inheritance hierarchies, *OOPSLA '92, Vancouver, British Columbia, Canada*, ACM Press, 1992, 25-40.

[13] **Ossher H., Harrison W., Budinsky F. and Simmonds I.,** Subject-oriented programming: Supporting decentralized development of objects, *7th IBM Conference on Object-Oriented Technology, Santa Clara, CA, USA*, IBM Press, 1994, 1-13.

[14] **Ossher H., Kaplan M., Harrison W., Katz A. and Kruskal V.,** Subject-oriented composition rules, *OOPSLA '95, Austin, TX, USA*, ACM Press, 1995, 235-250.

[15] **Ossher H., Kaplan M., Katz A., Harrison W. and Kruskal V.,** Specifying subject-oriented composition, *Theory and Practice of Object Systems,* **2** (1996), 179-202.

[16] **Wright C., Cowan C., Morris J., Smalley S. and Kroah-Hartman G.,** Linux security modules: General security support for the linux kernel, *11th USENIX Security Symposium, San Francisco, CA, USA, 2002.*

[17] **IBM,** SOP web page `http://www.research.ibm.com/sop/`, 1993.

[18] **FreeBSD Foundation**, FreeBSD web page `http://www.freebsd.org/`, 2006.

[19] **Linux Online**, Linux web page `http://www.linux.org/`, 2006.

[20] **NetBSD Foundation**, NetBSD web page `http://netbsd.org/`, 2006.

[21] **OpenBSD Foundation**, OpenBSD web page `http://openbsd.org/`, 2006.

Á. Balogh
Department of Algorithms
and their Applications
Eötvös Loránd University
Pázmány Péter sét. 1/C
H-1117 Budapest, Hungary
`bas@elte.hu`

Z. Csörnyei
Department of Programming
Languages and Compilers
Eötvös Loránd University
Pázmány Péter sét. 1/C
H-1117 Budapest, Hungary
`csz@inf.elte.hu`