A Survey of Configurable Operating Systems

Jean-Charles Tournier
University of New Mexico

November 24, 2005

Contents

1 Introduction **3**

2 Classification criteria **3**
2.1 Programming paradigm 4
2.2 Granularity 5
2.3 Depth 5
2.4 Time 5
2.5 Integrity 6
2.6 Application domain 7

3 Evaluation of configurable operating systems **7**
3.1 Choices 7
3.2 eCos 8
3.3 Exokernel 9
3.4 K42 10
3.5 Linux 11
3.6 MetaOS 12
3.7 MMLite 13
3.8 OSKit 14
3.9 Pebble 15
3.10 Scout 16
3.11 Spin 17
3.12 Think 18
3.13 TinyOS 19

4 Comparison **20**

5 Analysis **21**

6 Conclusion **23**

1 Introduction

The need to deal with configurable operating systems is not new and several research projects have tried, and are trying, to deal with this issue. Indeed, a number of past studies [10, 34, 4, 14, 32, 38, 54, 57] demonstrated that, from small constrained embedded systems to high performance computing, general purpose operating system negatively impacts application performance. As an example, [10] showed that a dedicated operating system for high performance systems transfers messages between nodes from 4 to 10 times faster than Linux does.

Although the word *configurable* is clearly defined as the fact *"to fashion by combination and arrangement"* [2], it covers three distinct notions in operating system research. Indeed, configurability deals with kernel extension, as well as kernel customization and kernel adaptation. An extension adds some functions to an initial kernel, while a customization is the fact to tune the kernel to suit a particular application or set of applications. At last, adaption is the property to be able to adjust a kernel's structure or behavior to new conditions. More generally, these three notions refer to the specialization of a kernel to an application, a set of applications or a given hardware in order to improve the overall system's performance. From these definitions, the number of concerned operating system research projects is even bigger as many projects address at least one of these issues. Indeed, as a fixed operating system is too restricted and limited, most of the actual projects provide some configuration facilities. Moreover, such operating systems are not restricted to dedicated systems, as even general purpose operating systems address this issues. As an example, the Linux operating system has evolved from a fixed kernel to a configurable one thanks to the definition of kernel modules that allow to add, or remove, functionalities to the kernel.

In order to identify the huge number of different configurable operating systems, a survey is therefore needed. Moreover, such a survey should allow to identify the main classes of configurability and to compare operating systems from a configurability point of view. However, only few papers try to do such a survey, but the main lack of these surveys is they are either out-of-date [16, 52, 51] or are too restricted [27]. The most recent survey [19] deals only with customizability in operating systems and the classification based on the time and the initiator of customization is not enough for these kind of systems.

In this paper, we present an up-to-date survey that includes newest research projects and based on revelant criteria that has been risen when authors started to work on a configurable operating system for high-performance computing [39]. Indeed, for each configurable operating systems we have to figure out the programming paradigm, the granularity and the depth of configuration as well as the time of configuration. Moreover, it is useful to identify if there are some mechanisms that enforce the integrity of the system after a configuration and which application domains the operating system primary targets.

The rest of this paper is organized as follows. The first section motivates the criteria used in this survey and outlines the different ways operating systems tackle each of them. The second section is an evaluation of several revelant operating systems based on the evaluation criteria, while the third part provides a comparison of them. Finally, before the conclusion, a last part analyzes the evaluation and points out the main lack of the existing configurable operating systems.

2 Classification criteria

In this section, we motivate and present more in details the classification criteria on which this survey is based. We have identified six different criteria namely programming paradigm,

granularity, depth, time, integrity and application domain.

2.1 Programming paradigm

The first criterion focuses on the programming paradigm used to configure an operating system. Over the past 30 years, several programming paradigms has been used for operating system configuration from classical libraries (e.g. *Exokernel, Pebble*) to objects (e.g. *K42, MetaOS*), through modules (e.g. *Linux, Spin*) and components programming (e.g. *MMLite, OSKit, Scout, Think TinyOS*). In order to avoid confusion, we need to identify the differences between a component, an object and a module. In this survey, a component follows the definition provided by C. Szyperski [55] where a component is defined as: *"a unit of composition with contractually specified interfaces and fully context dependencies that can be deployed independently and is subject to third-party composition"*. Although a module can be also defined as a unit of composition that encapsulates data and subprograms, the main differences with a component are a module can not be instantiated and can only be composed in a flat and static way. Finally, from the previous definition, a component is an enhancement of an object as it explicitly defines its requirement and addresses deployment and configuration issues.

It must be noticed that several projects are looking at other programming paradigms such as aspect oriented [47, 18, 17, 21] or domain-specific language (DSL) ones [6, 42]. Aspect-oriented programming paradigm is a technique for specifying behaviors (or strategies) independently of the target system. A "weaver" automatically integrates the code of such an aspect into the relevant system entities. This approach separates strategies, which are programmed using aspects, from the underlying mechanisms, and thus should simplify system evolution and extension. A domain-specific language (DSL) is a programming language dedicated to a particular application domain. A DSL is more restricted than a general-purpose language, such as Java or C, but encapsulates domain expertise that can allow verification of important safety properties. However, these projects have not been surveyed because they are relatively new and immature. Indeed, as an example the first version of the weaver aspect for C++ language, called AspectC [53], has just been released at the end of october 2005. Moreover, they only allow to deal with some parts of an operating system and are not used to develop a complete system. As an example, in the case of domain specific languages [6] deals only with scheduling issues and [42] is dedicated to device driver development. Moreover, in the case of aspect oriented programming, the most advanced study [21] deals with kernel extensions over a NetBSD operating system, but doesn't provide any facility to develop an entire operating system based on aspects.

All of these paradigms tend to structure the code as much as possible and therefore have a huge impact on how operating system can be configured. However, the choice of a programming paradigm does not affect other criteria except the integrity one. Indeed, while each programming paradigm allows to tackle granularity, depth, time and application domain criteria in various different ways, the integrity criteria depends on the facilities provided by the programming paradigm. As an example, the classical interface type-checking obviously needs the interface notion which is not present in the library programming paradigm. Moreover, the programming paradigm based on framework allows to enforce the configuration of a system in order to be validated automatically. Similarly, programming paradigms that allow to formally express requirements of each part of the system (e.g. components or modules thanks to well-defined required interfaces) can also be validated automatically but in a more flexible way than frameworks as no predefined configuration is required. It means that the choice of a programming paradigm must be mainly based on what kind of validation is expected for the system.

Finally, the programming paradigm used to configure the kernel may have some consequences on the way the upper levels of the system (services and applications) have to be developed.

As an example, the *Think* component-based operating system forces the whole system to be component-based.

2.2 Granularity

The granularity criterion focuses on the size of the unit of configuration defined by the programming paradigm. This size usually fits in a range from coarse to fine grain. A coarse grain size (e.g. *eCos, OSKit, Pebble, Scout, Linux*) allows to configure an entire subsystem at once, while a fine grain size (e.g. *MetaOS, MMLite, Spin*) allows to directly configure a function of the kernel (either its data or its behavior). A third level of granularity is usually identified, namely mid-coarse grain (e.g. *Choices, K42, TinyOS*), that allows to configure a subset of the kernel functionalities.

While, most of the programming paradigms have a fix grain of configuration, the programming paradigms with hierarchical facility (e.g. *Think*) allows to vary the granularity depending on the part of the system and the need of configuration. Moreover, such facility tackles the trade-off between the granularity and the complexity of the resulting structure as it allows to define coarse grain entities made of finer-grain ones. Indeed, as mentioned by the *K42* project, the main drawback of a mid or fine-grain granularity is the complexity to understand the interactions between the system's entities.

Finally, although this criteria seems to be strongly linked to the programming paradigm one, it is not. As an example, the *MMLite* and *OSKit* operating systems are both component-based, but the first one falls in fine grain category while the second one falls in the coarse grain category.

2.3 Depth

The third criterion focuses on the depth of configuration in a operating system. It refers to the level in a operating system below no configuration is possible. A first rough classification identifies two main categories of depth, namely full and partial. A full configurable operating system (e.g. *Choices, K42, MetaOS, MMLite, Spin, Think or TinyOS*) allows to configure the whole part of a kernel including its lowest part, while a partial configurable one has a fixed set of functions that can not be configured. However, this first classification needs to be refined in order to differentiate the many different operating systems that belong to the second category. Three sub-categories of the partial one have been identified, namely partial-small, partial-medium and partial-big, in order to evaluate the size of the non-configurable part. A partial-small configurable operating system has the smallest fixed part that usually only includes an hardware abstraction or reification layer with some resources protection and multiplexing facilities. Such typical operating systems are the *Exokernel* or *Pebble* ones. A partial-medium configurable operating system has a bigger fixed part made of a subset of classical operating system services such as the scheduling, inter-process communications and interruptions management. The *eCos, OSKit* and *Scout* operating systems fit in this category. Finally, a partial-big configurable operating system fixed the most part of its kernel except the device drivers part (e.g. *Linux*).

It must be noticed that each surveyed operating system project that falls in the full configurable category has either mid or fine grain granularity. They are therefore much more configurable as each part of the system can be configured at a mid or fine-grain.

2.4 Time

The time criterion focuses on when the operating system can be configured. Classically, such a configuration may occurs at each stage of the life cycle of an operating system: it may be at

development, compilation, boot or execution stage. If an operating system provides facilities to be configured at development or compilation stages it is called a static configuration, while a dynamic configuration covers the boot and execution stages.

The interest to identify the configuration time is multiple: a configuration at execution stage allows to correct a bug or add/remove functionalities without shutting down the system. It is extremely interesting for systems that require continuity of services. However, this kind of configuration, and more specially when function is removed or modified, rises several issues such as definition of safe points (when a configuration can be done), state transfer (how to map state of a replaced function to a new one), the redirection (how the system know to interact with the new part) and the version management (how different version of a function can be used). The interest of configuration at boot stage is usually to tune the system when it knows its exact environment. A configuration at compilation stage allows the compiler to automatically configure the system, while it is done manually at development stage. During these two last stages, either compiler or developer needs to have an exact knowledge of the whole system, including the applications and the hardware.

However, most of the surveyed operating systems fall in either the development stage category (e.g *Choices, K42, Linux, MetaOS, MMLite, Pebble, Spin* and *Think*) or the execution stage one (e.g. *eCos, Exokernel, Scout* and *TinyOS*). Only the *OSKit* operating system provides facilities to configure the system at boot-time. It must be noticed that most of the projects that fall in the development time category target constrained embedded systems. Indeed, it can be explain by the fact that a run-time configuration implies a cost and some indeterminism that can not be supported by such systems. For projects that fall in the run-time class, the difference between them is based on which kind of facilities is provided in addition to the classical dynamic loader. Indeed, many projects only provide this minimal facility and everything must be foreseen and implemented by the programmer. However, some projects such as *MMLite* and *K42* provide additional facilities such as safe points identification or state transfer mechanism. Finally, although all these projects provide run-time configuration, most of them do not explicitly mention which part of the system can or can not be configured during the execution of the system.

2.5 Integrity

The integrity criterion is one of the most important one for a configurable system, while it addresses the validity of a particular configuration. Indeed, to each configuration operation, it must be evaluated if the new configuration does not introduce incompatibility (e.g. functional, security or quality of service) with the rest of the system. It must be noticed that such an evaluation must be performed whenever a configuration is possible (c.f. time criteria).

From the surveyed operating systems, two main different mechanisms are used to enforce the integrity of a given configuration. The first one is based on a semantic validation in order to validate that each required and provided services are compatible and present. This kind of integrity is achieved by interface type-checking, as well as dedicated languages (e.g. *OSKit*, *TinyOS*), constraint definitions (e.g. *eCos*) or frameworks (e.g. *Choices*) which enforce the structure of the system. The second one comes from the security domain and allows, or not, a configuration to be done and isolates each part of the system. It covers mechanisms provided by the *MetaOS* project that defines configuration policies in order to identify which part of the system can be configured. It also covers *Pebble*'s mechanism that executes each extension in a distinct protection domain in order to avoid unintentional effects of the new configuration to the rest of the system. It must be noticed that each project that falls in the latter class also provides a semantic validation thanks to the classical interface type-checking. Moreover, some projects do not have specific mechanisms that address the integrity criteria except the underlying compiler

(e.g. *Linux*, *Exokernel*).

2.6 Application domain

The last criterion deals with the application domain primary targeted by an operating system. Indeed, depending on an application domain, a configurable operating system focuses on specific issues and provides some configuration facilities related to these issues. As an example, a real-time operating system should provide different scheduling policies, while a distributed operating system should provides different memory management polices.

Three main classes have been identified, namely distributed domain (e.g. *Choices, Exokernel* or *K42*), which includes high performance and parallel systems, general purpose domain (e.g. *Linux, MetaOS, OSKit* or *Spin*) and dedicated domain (e.g. *eCos, MMLite, Pebble, Scout, Think* or *TinyOS*), which includes embedded systems. It must be noticed that no operating system has become a standard in any of the application domains. Indeed, for each domain a lot of projects are still actively running. However, it seems that the embedded domain has more configurable operating system projects than the distributed one. It can be explained by the fact that the need of application specific operating system has primary appeared in embedded systems because of the restricted resources.

3 Evaluation of configurable operating systems

In this section, several configurable operating system projects are presented. Presentation of each project is divided in two parts: the first part gives an overall view of the approach, while the second part is an evaluation based on criteria presented in the previous section. In order to evaluate these criteria, publications are usually not not enough: it is why we mainly survey projects with available source code. Moreover, as an exhaustive survey is impossible to achieve, we rather present projects with relevant answers to the criteria.

3.1 Choices

Choices [13] is a customizable object-oriented operating systems developed at the university of Illinois and started in 1987[1]. Choices is more than a classical operating system project as it defines a methodology to develop operating systems. It means that it is more a software engineering project than a pure operating system one. Indeed, Choices' architecture is organized into frameworks of objects that are hierarchically classified by function and performance. This approach allows to customize the operating system by specializing classes in the various hierarchies, and by instantiating a specific set of objects.

Programming paradigm Programming paradigm of Choices is entirely based on frameworks [20] and objects. It defines a framework as *"an architectural design for an object-oriented system"* [13]: it allows to captures design insights using entity relationship diagrams, data flow diagrams, control flow diagrams, class hierarchies, class interfaces and path expression. Moreover, a framework is a reusable entity as it can be refined (called subframework) and may have several instantiations and implementations within a system.

In order to better understand the framework approach, we describe the main framework defined by Choices. This main framework provides generalized entities and constraints to which the specialized subframeworks must conform. It introduces the notion of a *Process* (a sequence

[1]The project doesn't seems running anymore as the last code release is from 1994

of actions), the address space of a *Process* called *Domain*, and *MemoryObjects* (data) that can be accessed by a *Process* in its *Domain*. The subframeworks introduce additional entities and constraints and subclass some of the entities of the framework. Examples of such subframeworks include virtual-memory management, process management, file system and message-passing. Recursively, these subframeworks may be refined further.

Granularity The unit of configuration in Choices is object as it is the basic entity of framework. Inside framework, objects are used to model both the hardware interface, the application interface, and all operating system concepts including system resources, mechanisms and policies. More precisely, every logical or physical system entity, such as CPUs, disks, memory, schedulers, address space or locks is reified by an object belonging to a framework. It is why, Choices is classified as a mid-grain configurable operating system.

Depth The Choices structure is a typical monolithic one as only applications are in the user space. The micro-Choices project is a redesign of Choices into a micro-kernel architecture. Nevertheless, both of these projects allow to configure each layer of the operating system including the low level one. Choices is therefore classified as a full configurable operating system.

Time Choices may be configured either at development time or run-time. At run-time, Choices provides a dynamic loading mechanism that permits application and system programs to add new system services at run-time. As Choices is written in C++, a specific library has been added to the language in order to allow dynamic class loading. Nevertheless, it must be noticed that each possible dynamic configuration must be foreseen at development time in order to tackle classical dynamic configuration issues (cf. section 2.4 page 5).

Integrity The choices compile tool chain enforces type safety of frameworks and objects. Moreover, it guides the programmer in implementing customizations by requiring particular behaviors and by enforcing the use of the interface provided by the framework.

Application domain Targeted systems of Choices are distributed and parallel systems as many objects of the source code deal with memory management, message-passing and network issues.

3.2 eCos

eCos [1] is an embedded real-time configurable operating system supported by the RedHat compagny and started since 2001.

Programming paradigm The programming paradigm used by eCos is the package one as each part of the system is a package. Each package is described in a single file using the CDL language. This file usually includes details of all the configuration options and how to build the package. Developers select and configure the required packages for a specific system using a visual tool.

Granularity The unit of configuration in eCos is the package. However, it must be noticed that each package has a plenty of configuration options which allow to tune them. Package granularity can be classified as coarse grain one as packages implement a whole part of the system such as the hardware abstraction layer, the file system or the network management.

Depth The depth of configuration in eCos stops at the kernel package. This package provides the core functionality needed for supporting multi-threaded applications. It has the ability to create new threads in the system and to control over various threads, for example manipulating their priorities. Moreover, it provides a choice of schedulers and a range of synchronization primitives, as well as the support for interrupts and exceptions. Thus, eCos is classified as a partial-medium configurable operating system.

Time The configuration of an eCos system is entirely done at development time thanks to a graphical tools. This tools allows to browse each package and each option of the packages.

Integrity The graphical tool used to configure an eCos system allows to check that constraints defined by packages are respected in a given configuration. These constraints define services required by a package.

Application domain The eCos operating system targets real-time embedded systems, and therefore has been ported on embedded processors (e.g. ARM). One of its goals is efficiency from a memory viewpoint, it is why it tries to solve the tradeoff between package granularity and package costs with multiple configuration options.

3.3 Exokernel

The Exokernel project [22, 37] is developed at the Massachusetts Institute of Technology. This project addresses the problem of performance and flexibility in operating systems by providing application-level management of physical resources. The main idea of the Exokernel project is that applications better know how resources must be managed in order to get optimal performances. Exokernel therefore defines a small kernel securely exports all hardware resources through low-level interface to untrusted library operating systems. Library operating systems use this interface to implement system entities and policies.

Programming paradigm In Exokernel, the programming paradigm to allow configuration is based on library. Moreover, Exokernel is developed using the C language for efficiency reasons. The kernel, as well as the libraries are implemented in C.

Granularity The unit of configuration in the Exokernel operating system is the library. Indeed, developers have to choose, or develop, the suitable libraries for their applications. As libraries implement some subpart of the system (RPC, scheduling, memory management), Exokernel is classified as a coarse grain configurable operating system.

Depth The depth of the configuration ends at the kernel level which is a minimal layer. It must be noticed that Exokernel does not emulate hardware resources but export it. It means that library link to the kernel are platform dependent. The Exokernel minimal layer functionalities is limited to ensuring protection and multiplexing of resources. Thus Exokernel fits in the partial-small configurable operating system category.

Time The configuration of the system is performed at development time: each library is linked statically to the kernel during compilation stage.

Integrity The integrity of a given configuration is only based on the C compiler.

Application domain The Exokernel operating system targets high performance operating systems with extensibility needs.

3.4 K42

K42 [3, 7] is a recent[2] open-source and scalable operating system kernel. It targets next generation servers ranging from small-scale multiprocessors, to very large-scale non symmetric multiprocessors. The main goals of K42 is to develop a scalable, adaptable, extensible and customizable kernel. The K42 operating system is currently running on PowerPC platforms including POWER3, POWER4, POWER4+ and POWER Mac G5. Finally, the K42 project is hosted by IBM.

Programming paradigm Originally, K42 uses the *building-block* programming paradigm [5]. Nevertheless, in the latest publication [3], authors do not refer to this paradigm anymore, but rather use the object oriented one as the difference between them was too thin. K42 is therefore structured in an object-oriented manner at each layer of the system. More precisely, each resource (e.g. virtual memory region, network, file, process) of the system is managed by a *" per-instance"* object. Nevertheless, as mentioned in [3], the largest drawback of the object-oriented nature of the system is the complexity it introduces. It especially appears when one is trying to understand the interactions between the different entities of the system.

Granularity As K42 is completely designed using object paradigm, its unit of configuration is object. It implies that K42 may be classified as mid-grain from a granularity point of view. Indeed, objects implement exceptions, as well as timers, process, scheduler, files, address space, etc. However, this mid-grain decomposition leads to a huge number of objects: there are about 1800 classes in the latest version of K42[3].

Depth Each part of the K42 operating system can configured, which means that K42 is classified as a full configurable operating system.

Time K42 may be configured either at development time or at run-time [7]. Obviously, configuration at development time can be achieved thanks to the available source code, but one of the main advantage of K42 is its ability to dynamically update objects. However, in K42 dynamic configuration is limited to the dynamic update of an existing part of the system. Three kinds of update has been identified: updates that only affect code (where any data structures remain unchanged across the update), updates that affect both code and global single instance data (e.g. Linux kernel's unified page cache structure) and updates that affect multiple-instance data structures (e.g. data associated with an open socket). K42 tackles the main issues related to dynamic configuration: configurable unit is object, safe point detection is achieved thanks to mechanism similar to read copy update (RCU) in Linux [41], state tracking is performed using the factory design pattern [29] and state transfer has to be foreseen by developers. Although K42 allows dynamic configuration, there are still open issues: as an example, object interfaces can not be dynamically changed. Moreover, dynamic configuration is not supported to objects outside the kernel space, but in the same time low-level exception objects are not concerned too. Finally, new issues have been risen such as timeliness of configuration: indeed, for some configuration, such as security one, it is important to know when a configuration has completed, and to be able to guarantee that configuration will be complete within a certain time frame.

[2]The first paper has been published in 2000

[3]The evaluation has been done on 10/25/2005 using CVS

Integrity In K42, integrity of a given configuration is only based on interface's signatures. It is classically performed by compiler when configuration is done at development time. At run-time integrity is enforced because of the limitation of dynamic configuration to dynamic update.

Application domain K42 targets high-performance systems ranging from small-scale multiprocessors to very large-scale non-symmetric multiprocessors. It implies a lot of objects dealing with cache-coherence, memory management, object distribution, etc. Moreover, K42 provides its own operating system library that can be mapped in order to get K42 compatibles with the Linux API and Linux ABI.

3.5 Linux

Linux [15] is a free software Unix-like operating system kernel that was begun by Linus Torvald in 1991 and subsequently improved with the assistance of developers around the world. Linux has been originally developed for the Intel 80386 processor and has since been ported to many other platforms. At this time[4], the latest version of the Linux source code is 2.6.14.

Programming paradigm The most part of the Linux source code is written in C, along with snippets of assembly language. However, the programming paradigm used to configure the Linux kernel is the module one. The basic idea of a linux module, which is called a Linux Loadable Kernel Module or LKM, is to generate in the first instance a minimal kernel, and then to load modules according to need. A LKM is defined as an object file that contains code. Moreover, a KLM has to provide some well-defined services in order to manage its life-cycle, such as the entry functions (e.g. *init_module()*) or exit functions (e.g. *cleanup_module()*).

Granularity Although, a LKM has no predefined granularity, in practice it is used to add support for new hardware (e.g. device drivers), file systems or for adding system calls. A LKM allows therefore a coarse-grain configuration of the Linux kernel.

Depth The LKM can only be loaded on a existing Linux kernel and only some parts of the kernel can be configured. As mentioned previously, these parts are typically the device drivers parts. It means that most of the operating system can not be configured. Linux is therefore classified as a partial-big configurable operating system.

Time One of the main advantage of the LKM is to be able to configure the Linux kernel without rebooting the system. However, Linux does not provide any mechanism to tackle the classical run-time configuration issues. Although, all these issues have to be foreseen by developers, Linux forces the place where it must be tackle thanks to the entry and exit functions of each LKM.

Integrity The integrity aspect is mainly addressed by the C compiler that generates the LKM. Moreover, when a LKM is loaded all its dependencies must be solved. It can be done either manually by the developer or automatically by the Linux kernel (thanks to the *modeprobe* command). However, there is no mechanism to prevent misbehavior. As an example, KLM is well know hacker's mechanism to intsall *rootkit*.

[4] Fall 2005

Application domain The primary goal of Linux is to be a general purpose operating system such as Unix. However, over the years, it has been ported and used to a wide variety of application domains from embedded systems to high performance ones.

3.6 MetaOS

The MetaOS project [35, 36] deals with development of secure and flexible operating systems using a *meta* approach. This project is an improvement of a previous meta-object oriented operating system, namely Apertos [59], in order to avoid heavy performance penalties and coarse-grain flexibility. The main motivation to use a *meta* approach is that it should increase flexibility as it provides a clear separation between provided services and the way to access to these services. The MetaOS project started in 1996 at the university of Victoria (Canada) and ended in 1998. This project is a prospective one and only a prototype has been developed which is based on a Java Virtual Machine. Moreover, we do not have access to the source code of the project.

Programming paradigm The MetaOS project uses a *meta* programming paradigm. The *meta* term is not limited to meta-object, but it is also used for meta-interfaces and meta-spaces. Meta-objects are defined as objects that contain implementation details of other objects. Meta-interfaces are interfaces that articulate and expose key implementation details. They can contain declarative methods as well as imperative methods. Declarative methods allow retrieval of status data and selection from a range of existing implementation options, and imperative methods allow the addition of new code to an existing implementation. Finally, meta-spaces contains meta-objects that form a optimal execution environment for objects.

These differences of *meta* concept lead a structuring of the system in three parts: the base level is implemented by objects and represents applications; the level below, called meta-level, consists of meta-objects that implement classical operating system services; and the lower level, called meta-meta-space, which allocates the resources to the meta-spaces. Meta-spaces dynamically group meta-objects in order to support application with similar requirements (e.g. scheduling, memory management, networking).

Granularity Although the base entity of the *meta* programming paradigm is the object, the unit of configuration is the method thanks to the concept of meta-interfaces. Therefore, MetaOS has a fine-grain configuration unit.

Depth MetaOS should be fully configured including its kernel. Nevertheless, this criteria is difficult to evaluate as the prototype of MetaOS has only been implemented over a Java virtual machine.

Time One of the main claim to use *meta* concept is its ability to configure an operating system at run-time. Indeed, using the meta-interfaces concept, it is possible to identify and configure each part of the system at a fine-grain granularity. Nevertheless, none of the reference papers deal with classical run-time configuration issues (cf. section 2.4 page 5).

Integrity At development time, the integrity criterion is achieved by the Java programming language that is used to develop the prototype of MetaOS. At run-time, the integrity aspect is performed thanks to the meta-interfaces and meta-space manager concepts. Indeed, meta-interfaces allow to define policies in order to validate and allow any changes, while the meta-space manager has special privileges in order to dynamically group meta-objects. However, there is no integrity mechanisms to validate each combination of meta-objects.

Application domain The MetaOS project targets general purpose operating system as cost of *meta* approach is still a critical issue.

3.7 MMLite

MMLite [33] is an operating system project developed by Microsoft in 1998. The main goal of MMLite is to allow a system to be dynamically assembled into a full application system. The dynamic configuration is achieved through a original mechanism called *mutation*. Another issue targeted by MMLite concerns the efficiency of the system from a memory footprint viewpoint.

Programming paradigm MMLite uses a programming paradigm based on COM component [11]. A MMLite component is a collection of concrete classes and some glue code that ties them together. A concrete class implements one or more interfaces. An interface, also known as an abstract class, is purely abstract in that it is not tied to any particular implementation. Where an interface is purely abstract, a concrete class is concrete, i.e. it is one particular implementation. A concrete class can inherit another concrete class (also called subclassing). In that case some methods are copied from the super-class (the one inherited from) and others are implemented by the subclass (the methods override those of the super-class). Finally, an interface defines the interaction between a user of a service (the client) and the implementer of the service (the server). An interface is essentially a list of methods (functions) with their arguments and behaviors. It must be noticed that all interfaces derive from *IUnknown* interface. IUnknown implements lifetime control through reference counting and type casting with version checking through Unique Identifiers (UID).

To summarize an interface is implemented by a concrete class that in turn resides within a component.

Granularity MMLite defines a unit of configuration and a unit of composition. The unit of composition is either component or concrete class, whereas the unit of configuration is the method. Indeed, a dynamic configuration may lead to the redefinition of a method of a concrete class. Therefore, MMLite is classified as a fine grain configurable operating system.

Depth MMLite can be configured at each level as components are also used to structure the kernel. However, the way components are assembled is pre-defined as the global structure of the system must be a micro kernel one. Another example, is that a scheduler component is always required even if the system is not multi-threaded. However, MMLite is considered as a full configurable operating system.

Time The main goal of MMLite is to allow configuration at run-time. MMLite allows run-time changes to the implementation of an object, even while the object is being used thanks to the *mutation* mechanism. Mutation is the act of atomically changing an ordinarily constant part of an object, such a method implementation. Each mutation is performed by a thread which is called a *mutator*.

A mutator must translate the state of the object from the representation expected by the old implementation to the one expected by the new implementation. It must also coordinate with worker threads and other mutators through suitable synchronization mechanisms. Transition functions capture the translations that are applied to the object state and to the worker threads execution state. In order to limit the amount of metadata, execution transitions only happen between corresponding clean points in the old and new implementations.

The synchronization mechanisms suitable for implementing mutation is divided into three groups. The first one is mutual exclusion. In this case mutation cannot happen while workers are executing methods of the object to be mutated. Mutual exclusion is simple in that there is no worker state associated with the object when mutation is allowed to happen. The second group is based on transactions. In this case, mutation have to roll back the workers that are affected by mutation. Mutators and workers operate on an object transactionally and can be aborted when necessary. The last group is based on swizzling. In this case, mutators modify the state of the workers to reflect mutation. Instead of waiting for workers to exit the object or forcing them out, the third mechanisms just suspends them. The mutator then modifies the state of each worker to reflect the change in the object.

Integrity Whenever a configuration is possible, MMLite checks interfaces type in order to evaluate their compatibilities from a functional and a version point of view.

Application domain MMLite targets multimedia embedded systems, such as DirectX accelerator board, x86, ARM and VLIW processors. It leads to the need of small memory footprint. As example, a minimal kernel used in a watch has a size of 10 KB on a x86 processor.

3.8 OSKit

The OSKit project [26], from the university of Utah, aims to provide a framework and a set of modularized components for the construction of operating system kernels. It means that OSKit does not define a new operating system, but rather provides facilities to develop a specific operating system. These facilities include a component library as well as tools to compose an operating system. The project started in 1996 and the last change in the source has been done in 2002.

Programming paradigm The programming paradigm used by OSKit is the component one. It defines its own component model called *unit* [25]. This component model defines two kinds of components: the first one, called *atomic unit*, is the smallest unit of composition, while the second one, called *compound unit* is composed of other components. It means that this component model is hierarchical. Moreover, it must be noticed that component's behavior is written using C language, while component description and linkage is described using a dedicated language, called Knit [49].

Granularity Although OSKit is based on a hierarchical component programming paradigm, it must be classified as a coarse grain configurable operating system. Indeed, components implement sub-part of the system, such as memory, network or processes management, and are not used at a finer grain.

Depth OSKit is clearly classified as a partial-medium configurable operating system as it is based on a minimal kernel component. Above this component, configuration is possible in order to include several device drivers, memory manager, and so on.

Time The time of configuration in OSKit is mainly at development time, but it can be also achieved at boot-time. Nevertheless, it must be noticed that configuration is impossible at run-time. Indeed, after compilation stage, the component's structure of the system is lost and leads to a closed and monolithic system.

Integrity The integrity of a given OSKit configuration is enforced by the dedicated language in charge of composing and linking components. In addition to the classical import/export services type checking, this language, and its associated compiler, allow to check domain-specific architectural invariant defines by programmers. For example, it is possible to check that code executing without a process context will never call code that requires a process context.

Application domain The OSKit project mainly focuses on general purpose operating systems as part of the code comes from the FreeBSD operating system.

3.9 Pebble

Pebble [28, 40] is a component-based operating system developed at the Bell Laboratories. It has been designed with the goals of flexibility, safety and performance. Pebble is based on three main concepts, which are (1) a minimal privileged mode nucleus, (2) a set of replaceable system service and application components running in distinct protection domain and (3) a code generator specialized for each possible cross-domain transfer. The first version of Pebble was released in 1999.

Programming paradigm Although Pebble defines itself as a component-based operating system, it can not be classified as it according to component's definition given in section 2. Indeed, Pebble's component is never defined (no notion of interfaces, nor binding) and it would be rather classified as a library-based operating system. In Pebble, each library is implemented by a single C file and is in charge of a specific operating system service.

Granularity Unit of configuration in Pebble is therefore library. Each library implements a classical operating system service such as device drivers (e.g. serial, IDE, PCI, SCSI, ethernet or RTC), cache, interrupt handler, scheduler, files system or virtual memory management. Pebble is therefore classified as coarse-grain grain from the granularity point of view.

Depth The Pebble operating system is based on a micro-kernel called *nucleus*. The nucleus is as minimal as possible and is only responsible for switching between protection domains. Nevertheless, the nucleus can not be configured: configuration only deals with selecting operating system services implemented by libraries at higher level. Thus, Pebble is classified as a partial-small configurable operating system.

Time Pebble allows mainly to configure a system at development time. Moreover, the Pebble's reference paper [28] refers to dynamic library replacement. However, neither the reference paper nor the source code give any details on the way such a replacement is achieved.

Integrity Pebble does not have the interface concept, thus the integrity of a configuration is only ensured by the underlying programming language compiler which is C. At run-time protection is provided by hardware-enforced virtual address spaces as each library runs in a distinct protection domain. All communication between protection domains is done by the means of *portals*. A portal enforces security between protection domain as only if a portal exists between protection domain A and protection domain B can A invokes a service offered by B. The hardware memory protection prevents access to memory outside the protection domain, and limits access to the well-defined portals.

Application domain Pebble mainly addresses embedded communication devices with safety needs. It is important to notice here, that the Pebble architecture is based around the availability of hardware memory protection: it requires a memory management unit.

3.10 Scout

Scout [45, 44] is a communication-oriented operating system developed at the University of Arizona. Therefore, Scout mainly focuses on the efficiency of processing and forwarding data in systems with different characteristics. The latter requirement is performed thanks to the definition of Scout as a configurable operating system, while the first one is achieved thanks to the *path* abstraction. A *path* encapsulates the flow of data from an I/O source to an I/O sink and defines its semantic (e.g. its reliability, security or real-time behavior) as well as the collection of system resources needed to process its data.

Programming paradigm Components, which are called *routers*, are the unit of program development in Scout. Each Scout component provides a well-defined and independent functionality. Well-defined means that there is usually either a standard interface specification, or some existing practice that defines the exact functionality of a component. Independent means that each single component provides a useful, self-contained service. More concretely, a Scout component is implemented as a collection of C source files and is described using a "spec" file that lists the related source files and the accessible services.

To form a complete system, individual components are connected into a component graph: the nodes of the graph correspond to the components included in the system, and the edges denote the dependencies between these components. Two components can be connected by an edge if they support a common service interface.

Granularity A Scout component typically implements network protocols (such as IP, UDP and TCP), storage system (such as VFS, UFS or SCSI) and drivers for the various devices in the system. It means that a Scout module allows a coarse grain configuration of the operating system. In the latest source code release, the number of modules is 62 and each of them has an average of 4000 lines of C code.

Depth Scout is based on a fixed kernel that implements interrupts, exceptions management as well as low-level services. Scout is therefore classified as a partial-medium configurable operating system.

Time The configuration of Scout is only allowed at development time thanks to the definition of the module graph. Nevertheless, components are run-time entities which allow to identify the structure of the system at run-time: it is used to manage the path abstraction in order to be able to modify path dynamically.

Integrity The integrity criterion in Scout is enforced by the Scout tool chain in order to check if connected components have compatible interfaces.

Application domain Scout targets information appliances such as network devices, multimedia workstation or individual nodes of a distributed-memory multi-computer system.

3.11 Spin

SPIN [8, 48] is an operating system project developed at the University of Washington in 1995. The main goal of this operating system is to allow applications to dynamically specialize the kernel in order to achieve a particular level of performance and functionality. A specialization can add new kernel services (by linking new code), as well as replace default policies or migrate applications function to kernel address space. In SPIN a specialization is called an *extension* and its behavior is defined through the execution model.

Programming paradigm SPIN and its extension are written in Modula-3 [46], a general purpose programming language based on modules. The key features of the language include support for interfaces, type safety, modules, generic interfaces, threads and exceptions. The design of SPIN depends only on the language's safety and encapsulation mechanisms. However, applications can be written in any language.

Granularity Granularity in SPIN is defined by the extensions. Extensions are defined in terms of events and handlers. An event is a message that announces a change in the state of the system or a request for a service. An event handler is a procedure that receives the message. An extension installs a handler on an event by explicitly registering the handler with the event through a central dispatcher that routes events to handlers. Using this extension model, extension may be defined at the granularity of a procedure or may entirely replace an existing system service. Therefore, SPIN is classified as a fine-grain configurable operating system.

Depth Although the SPIN operating system is structured as a micro-kernel, the extension model allows to configure the kernel itself.

Time The main purpose of the SPIN operating system is to allow to configure the operating system at run-time. This kind of configuration is achieved thanks to the extension model as described in the granularity criterion. However, the extension model does not defines how extensions can be removed or how the state transfer between two extensions is achieved.

Integrity The integrity criterion is enforced by the Modula-3 programming language and capabilities. Indeed, Modula-3 compiler enforces interface boundaries between modules using the encapsulation idea and sensitive kernel interfaces are secured via a restricted linker and the type-safe properties of the language. Moreover, SPIN uses capabilities in order to enforce operations that can be applied to resources. Indeed, all kernel resources are referenced by capabilities. A capability is defined as an unforgeable reference to a resource which can be a system module, an interface or a collection of interfaces. Individual resources are protected to ensure that extensions reference only the resources to which they have been access. Interfaces and collection of interfaces are protected to allow different extensions to have different views on the set of available services.

Application domain The goal of SPIN is to build a general purpose operating system that provides extensibility, safety and good performance. As an example, a SPIN instance has been developed for a web server. The SPIN Web server executes entirely in the kernel address space. This allows it to access the network and the local disk with low latency. The Web server application, as well as the file system interface, entire network protocol stack, and device infrastructure are all linked into the system after it boots.

3.12 Think

Think [23, 24] is a component-based framework for operating system development started in 2002 at INRIA and France Telecom R&D. Think defines a component library that implements the main operating system services and a set of tools in order to manage component composition. One of the main Think goal is to be as flexible as possible in order to build dedicated and fully configurable operating system. In order to achieve these goals, Think uses a component approach as far as possible. Therefore, it allows a developer to build a entire operating system from the component library without being enforced into a predefined kernel structure.

Programming paradigm The Think project is clearly based on a component programming paradigm. Indeed, Think is an implementation of the Fractal [12] component model which is a high-level generic component model. This model is hierarchical and defines component as run-time entities. Therefore, a Fractal component may be made of other components and is defined through a content, a controller and a set of required and provided interfaces. A content implements the functional part of a component. In a case of a hierarchical component, a content is made of sub-components and in a case of a base component a content is made of code. A controller manages the content of a component. At least, a controller can control the life-cycle of a component and also its structural content with adding or removing sub-components. Concretely, content of lowest-level components are made of C or assembly code, while description of component composition and component hierarchy is made through an Architecture Description Language (ADL) defined by the Fractal component model. It must be noticed that the Fractal ADL aims to be flexible in order to integrate various aspects of composition depending on application domain.

Granularity In order to achieve its flexibility goal, Think adopts components everywhere. Indeed, in Think everything is a component. As it is based on a hierarchical component model, Think provides different component granularity. A component may be as fine-grain as a mutex or as coarse-grain as a device driver. However, although the systematic use of components in operating system provides benefits (e.g. reusability, flexibility and portability), there is a tradeoff to achieve between component granularity and run-time overhead as components are run-time entities. A general approach adopted by Think is that components who share the same data are not decomposed. As an example, an IP session component doesn't need to be decomposed into one component for handling input data and another one for handling output ones. Nevertheless, in Think the granularity of configuration can be from fine-grain to coarse-grain using hierarchical property.

Depth Think does not predefined the operating system structure, thus the kernel part itself can be configured. Moreover, it must be noticed that Think does not identify the classical system decomposition (i-e resources level, kernel level, services level and applications level) as each level is developed in a homogeneous way using components. Thus each level can be configured using the same mechanisms.

Time Think provides configuration facilities at both development and run-time. However, the main configuration of a system is achieved at development time. At run-time, configuration is facilitates as components are run-time entities. Moreover, the Think framework provides a dynamic loader and the concept of optional interfaces that allows a component to use a service if an only if it is present. Therefore, a component may use different services while the system evolves.

However, Think does not tackle any of the issue of run-time configuration, thus everything must be foreseen by developers.

Integrity The Think framework validates a given configuration thanks to the classical interface-type checking based on services signatures. Moreover, a configuration is validated if and only if all the mandatory required services of components are satisfied. This validation is achieved thanks to the ADL files of a system that describe which and how components are involved. At run-time, several extensions has been developed in order to validate a configuration from a quality of service point of view [58] and from a security one [56].

Application domain Although Think does not target any particular application domain, in practice it is more used for embedded systems. Indeed, the Think library has been ported to many ARM processors, but also on ST Microelectronics DSP processor, Tini Internet Interface and MindStorm Lego RCX. However, some ports has also been achieved on more powerfull microprocessor such as PowerPC G3 and G4 and Intel X86. It must be noticed that the number of ports is facilitated because it only deals with a minimum number of components which are the ones that reify the hardware.

3.13 TinyOS

TinyOS [31, 34] is an open-source operating system from the University of California designed for wireless embedded sensor networks. TinyOS provides a new component model suitable for this kind of systems and provides a library dedicated to networking and sensor data collection. Its goal is to provide mechanisms to build highly specialized operating systems with a particular focus on memory footprint.

Programming paradigm TinyOS is developed using the nesC language [30] which is a component programming language. A nesC component provides and uses interfaces which are the unique access points to the component. An interface models some services and is specified by an interface type. Interfaces are bidirectional: they contain commands and events. The provider of an interface implements the commands, while the users implements the events. Moreover, a component may have several interfaces of the same type but with different names.

The nesC language provides a hierarchical component model as component may be made of other components. In this case, components from the lowest level are called *module components*, while components from the upper level are called *configuration components*.

Granularity TinyOS provides a library of small components that implement functions such as the UART, timer. More complicated part of the system such as network management is divided into several sub-components. To give an idea, component have an average of 120 lines of code. However, TinyOS is classified as a mid-grain configurable operating system as a unit of configuration encapsulates several services.

Depth TinyOS allows to configure each level of the operating system, including the kernel itself.

Time All the system can only be configured at development time using the nesC language.

Integrity The integrity of a given configuration is enforced by the nesC compiler. It must be noticed that nesC is a unique language dedicated to component programming. Moreover, the same language is used to described component behavior as well as component configuration. Nevertheless, the integrity aspect is still limited to interface signature.

Application domain TinyOS targets wireless network embedded systems which are very constrained systems. It is why no dynamic memory allocation is allowed and the full component graph is know at compile time. These restrictions make the whole program analysis and optimization significantly simpler and more accurate. As an example, the nesC compiler eliminate unreachable code as well as inline small functions. A typical instantiation of TinyOS fits in 178 bytes of memory.

4 Comparison

This section aims to compare the surveyed operating systems in order to evaluate how much a system is configurable. The following comparison is only based on the granularity, depth, time and integrity criteria, as they qualify how much a system is configurable. Indeed, the programming paradigm is useless for the comparison as it only captures how a configuration is achieved. Likewise, the application domain criteria is useless to figure out the level of configurability of a given operating system. However, this criteria is used to compare operating systems that belong to the same application domain. The application domains are the distributed one, the general purpose one and the dedicated one, as mentioned in section 2.

Each comparison criteria has a set of possible values as identified in the section 2. For the granularity criteria, the set of values are fine, mid and coarse grain. Moreover, from a configurability point of view, a fine grain is more configurable than a mid grain which is itself more configurable than a coarse grain. The granularity criteria can therefore be expressed as:

$G = \{fine, mid, coarse\}$ and $fine > mid > coarse$

For the depth criteria, the possible values are either full, partial-small, partial-medium or partial-big. Moreover, from a configurability point of view a full operating system is more configurable than a partial-small one, which is more configurable than a partial-medium one and partial-big one. The depth criteria can therefore be defined as:

$D = \{full, \textit{p-small}, \textit{p-medium}, \textit{p-big}\}$ and $full > \textit{p-small} > \textit{p-medium} > \textit{p-big}$

For the time criteria, the only possible values are development stage, boot stage or execution stage. A development stage configuration is basic, while a boot stage one is more complicated but less than a execution stage one. Thus, time criteria is defined as:

$T = \{devpt, boot, exec\}$ and $exec > boot > devpt$

Finally, the integrity criteria has only three possibles values which are none, semantic and security. The order between these values is defined as:

$I = \{none, semantic, security\}$ and $security > semantic > none$

From these definitions, a given operating system, S_i, is therefore a subset of the union of the sets defines by the granularity (G), depth (D), time (T) and integrity (I) sets:

$S_i \subset (G \cup D \cup T \cup I)$

If $S_i = \{G_i, D_i, T_i, I_i\}$ and $S_j = \{G_j, D_j, T_j, I_j\}$ are two different operating systems, we say that S_i is more configurable than S_j if and only if $G_i \geq G_j$, $D_i \geq D_j$, $T_i \geq T_j$, $I_i \geq I_j$ and if at least one element of S_i is greater than the one in S_j (otherwise S_i and S_j are equals). Moreover, S_i is strictly more configurable than S_j if and only if S_j is a subset of S_i, i-e $S_j \subset S_i$. These two comparison definitions allows to differentiate an operating system with greater values from an operating system that covers a wider range of criteria values.

The three following graphs (cf. figure 1, 2 and 3) present a visual comparison of each surveyed operating systems in regards with their application domain. The graph in figure 1 deals with the general purpose operating systems. It shows that from these comparison criteria MetaOS and Spin are equals and are more configurable than OSKit an Linux. However, the difference between MetaOS and Spin is about the provided configuration tools and facilities which is not captured with the criteria. Likewise, graph in figure 2 shows that Choices and K42 are equals, although K42 is more convenient to use as it integrates some automatic mechanisms for dynamic reconfiguration. Finally, graph on figure 3 shows that no operating system can be identified as the most configurable one in dedicated application domain. However, from the comparison definitions, Think is strictly more configurable than MMLite as Think also provides a mid and coarse grain granularity.

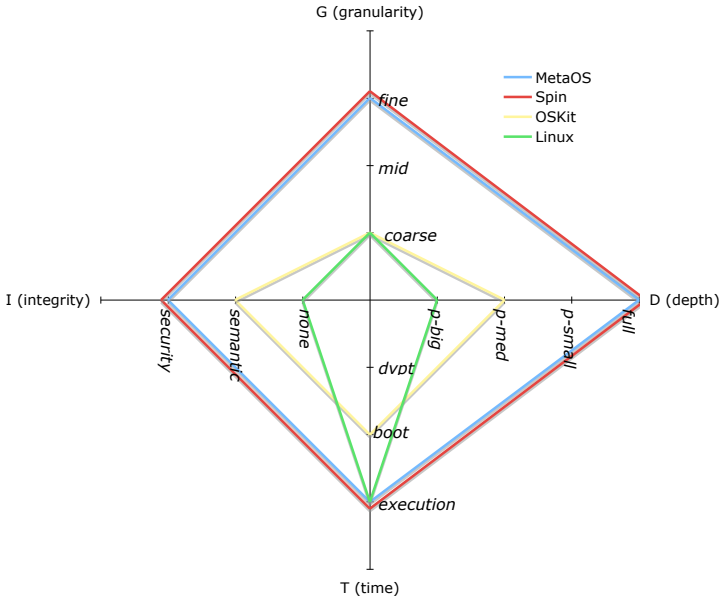

Figure 1: Comparison of general purpose operating systems.

5 Analysis

Although the previous sections demonstrated that configurability is achieved in various different ways, many aspects have still not been tackled. In this section, we presents an analysis of the previous surveyed operating systems in order to point out their main lacks to be more configurable and easier to use and validate:

- Configuration of an operating system is always seen from a functional point of view, but never from a non-functional one. Indeed, a configuration is always validated if all required services are present and compatible from a signature point of view, but not how these services are provided. Nevertheless, operating systems such as MMLite or OSKit check some fixed nonfunctional aspects such as version compatibility or architecture constraints. However, the type of checked aspects is fixed and there is no way to define which kind of aspect must be checked in addition to the functional one. A classical way to add nonfunctional

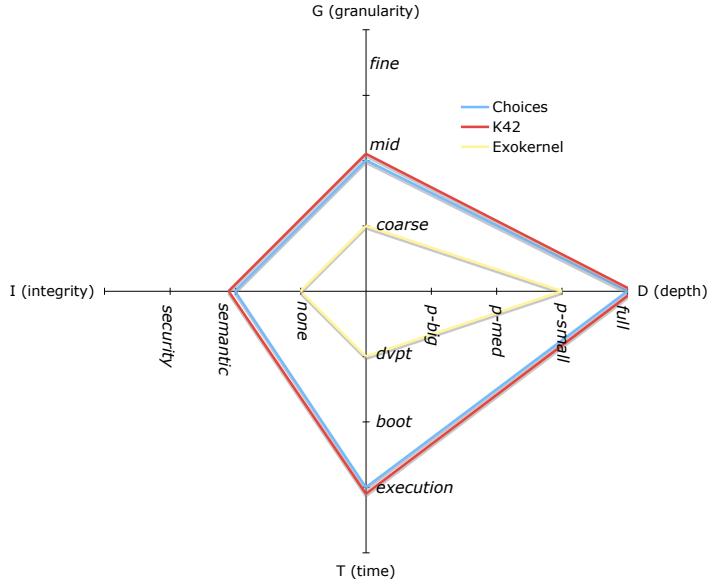

Figure 2: Comparison of operating systems for distributed systems.

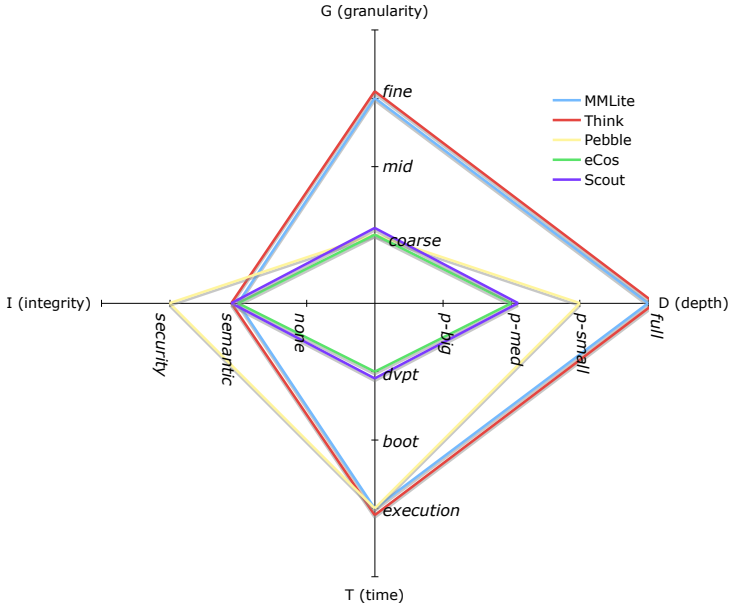

Figure 3: Comparison of dedicated operating systems.

attributes to a part of a system is to used reach interfaces [9]. A rich interface usually contains informations about method signatures, pre and post conditions, protocol specifications and quality of service requirements including security, fault-tolerance, resources or real-time requirements. This kind of issue is widely studied in classical distributed applications [50, 43]. Moreover some recent studies deal with security [56] and quality of service [58] in configurable operating system. It must be noticed that most of the studies dealing with non-functionnal aspects are based on a component programming paradigm as interfaces are clearly expressed.

- Although a lot of projects claim to deal with run-time configuration, a lot of issues are still not tackled. One of the main one is about the state transfer when a part of a system must be replaced. Indeed, this aspect must always be foreseen and programed by developers. Another issue is about which part of the system can be effectively configured at run-time. Although most of the project do not mention about this issue, the K42 project explicitly excludes code linked to low-level aspect from dynamic configuration. Finally, the last issue concerns the timeless of configuration in order to guarantee that a configuration will be achieved within a certain time frame. A further issue will concern the dynamic management of nonfunctional aspects as explained previously.

- The choice of the granularity of the unit of configuration is a tradeoff between the system complexity, possibility of configuration and run-time cost. Indeed, more units are fine grain, more the system is configurable and customizable and more it is complex. Moreover, at run-time the system needs to keep its structure explicitly in order to be able to identify or configure each part. It also means that more units are fine grain, more the cost is important. A way to tackle this tradeoff is to use hierarchical programming model such as the component one. However, it does not take into account the run-time cost as the unit of configuration at development time is the same at run-time. It should be interesting to be able to specify at development time which part of the system should be reconfigured. Although, it requires to know how the system will be used, it can avoid most of the cost by keeping only the relevant part of the structure alive at run-time.

- All development-time configurations achieved by the surveyed operating systems are done manually. In the case of applications specific operating system, the developer needs to know the exact requirements of applications. It should be interesting to have some mechanisms that can evaluate applications code and determine which operating system configuration is the most suitable. By this way, applications developers have only to focus on their applications and not on the lowest part of the system. Such mechanisms can be based either on a purely code analyzer or on design patterns. Indeed, using design patterns, it would be easier to automatically identify the way application is working and therefore its requirements from an operating system point of view. However, such design patterns must be specific to each application domain and they must be opened enough to fit a wide range of applications inside a domain.

6 Conclusion

References

[1] http://sources.redhat.com/ecos/.

[2] *The Oxford English Dictionary*. Oxford University Press, 1989.

[3] J. Appavoo, M. Auslander, M. Butrico, D. M. daSilva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating-system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[4] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 96–107, 1991.

[5] Marc A. Auslander, Hubertus Franke, Benjamin Gamsa, Orran Krieger, and Michael Stumm. Customization lite. In *Workshop on Hot Topics in Operating Systems*, pages 43–48, 1996.

[6] L.P. Barreto and G. Muller. Bossa: A DSL Framework for Application-Specific Scheduling Policies. In *Eighth Workshop on Hot Topics in Operating Systems*, 2001.

[7] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *USENIX 2005 Annual Technical Conference, General Track*, pages 279–291, 2005.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[9] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[10] Ron Brightwell, Rolf Riesen, Keith Underwood, Trammell Hudson, Patrick Bridges, and Arthur B. Maccabe. A Performance Comparison of Linux and a Lightweight Kernel. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster2003)*, 2003.

[11] K. Brockshmidt. *Inside OLE, Second Edition*. Microsoft Press, Redmond WA, 1995.

[12] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model - specification. In *http://fractal.objectweb.org*, 2004.

[13] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing choices:an object-oriented system in c++. *Communications of the ACM*, 36(9):117–126, 1993.

[14] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *Operating Systems Design and Implementation*, pages 165–177, 1994.

[15] R. Card, E. Dumas, and F. Mevel. *The Linux Kernel Book*. Wiley, 1998.

[16] Lee Carver, Ying-Hung Chen, and Theodore Reyes. Practice and technique in extensible operating systems. Technical report, University of California, San Diego, 1998.

[17] Yvonne Coady, Gregor Kiczales, and Michael Feeley. Exploring an Aspect-Oriented Approach to Operating System Code. In *Proceedings of the 4th Workshop on Object-Orientation and Operating Systems at the 15th European Conference on Object-Oriented Programming (ECOOP-OOOSW)*, 2001.

[18] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the European Software Engineering Conference (ESEC)*, 2001.

[19] G. Denys, F. Piessens, and F. Matthijs. A survey of customieability in operating systems research. *ACM Computing Surveys*, 34(2):450–468, December 2002.

[20] L.P. Deutsch. Design reuse and frameworks in the smalltalk-80 programming system. In ACM Press, editor, *Software Reusability*, volume 2, pages 55–71, 1989.

[21] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, 2005.

[22] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[23] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Usenix Annual Technical Conference*, 2002.

[24] Jean-Philippe Fassino, Jean-Charles Tournier, and Tahar Jarboui. The think component-based operating system. In Hermes Science and Lavoisier Company, editors, *Model Driven Engineering for Distributed Real-time Embedded Systems*. International Scientific and technical Encyclopedia, 2005.

[25] M. Flatt. *Programming Languages for Component Sotware*. PhD thesis, Rice University, 1999.

[26] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: a substrate for kernel and language research. *SIGOPS Oper. Syst. Rev.*, 31(5):38–51, 1997.

[27] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable component-based operating systems for embedded applications. *IEEE Micro*, 21(31):54–68, 2001.

[28] Eran Gabber, Christopher Small, John Bruno, José Brustoloni, and Avi Silberschatz. The pebble component-based operating system. In *USENIX Techincal Conference*, pages 267–282, 1999.

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[30] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[31] David Gay, Philip Levis, and David Culler. Software design patterns for tinyos. In *ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, 2005.

[32] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 27, pages 187–197, 1992.

[33] Johannes Helander and Alessandro Forin. Mmlite: a highly componentized system architecture. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 96–103, 1998.

[34] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Conference Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[35] Michael Horie, James C. Pang, Eric G. Manning, and Gholamali C. Shoja. Designing meta-interfaces for object-oriented operating systems. In *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 989–992, 1997.

[36] Michael Horie, James C. Pang, Eric G. Manning, and Gholamali C. Shoja. Using meta-interfaces to support secure, dynamic system reconfiguration. In *4th International Conference on Configurable Distributed Systems*, 1998.

[37] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russel Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Symposium on Operating Systems Principles*, pages 52–65, 1997.

[38] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas AndersonTools for the Development of Application-Specific Virtual Memory Management. Tools for the Development of Application-Specific Virtual Memory Management. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 48–64, 1993.

[39] Arthur B. Maccabe, Patrick G. Bridges, Ron Brightwell, Rolf Riesen, and Trammell Hudson. Highly configurable operating systems for ultrascale systems. In *Proceedings of the First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters*, 2004.

[40] Kostas Magoutis, José Carlos Brustoloni, Eran Gabber, Wee Teck Ng, and Avi Silberschatz. Building appliances out of components using pebble. In ACM Press, editor, *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 211–216, 2000.

[41] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[42] Fabrice Merillion and Gilles Muller. Dealing with hardware in embedded software: A general framework based on the devil language. *SIGPLAN Not.*, 36(8):121–127, 2001.

[43] Richard Monson-Haefel, Bill Burke, and Sacha Labourey. *Enterprise JavaBeans, Fourth Edition.* O'Reilly, 2004.

[44] Allen Brady Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, Todd A. Proebsting, and John H. Hartman. Scout: A communications-oriented operating system. In *Operating Systems Design and Implementation*, 1994.

[45] David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 153–167, 1996.

[46] Greg Nelson. *Systems Programming With Modula-3.* rentice Hall Series in Innovative Technology, 1991.

[47] Spinczyk Olaf and Daniel Lohmann. Using AOP to Develop Architecture-Neutral Operating System Components. In *ACM Press : Proceedings of the 11th ACM SIGOPS European Workshop (SIGOPS-EW '04)*, pages 188–192, 2004.

[48] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 201–212, 1996.

[49] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *4th Operating Systems Design and Implementation (OSDI)*, pages 347–360, 2000.

[50] Douglas C. Schmidt, Nanbor Wang, and Carlos O'Ryan. *Overview of the CORBA Component Model*, chapter VII in Component-Based Software Engineering: Putting the Pieces Together. Addison Wesley Professional, 2001.

[51] M. Seltzer, Y. Endo, C. Small, and K. Smith. Issues in extensible operating systems. Technical Report TR-18-97, Harvard University, 1997.

[52] Christopher Small and Margo Seltzer. A comparison of os extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[53] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP Extension for C++. *Software Developer's Journal*, pages 68–76, 2005.

[54] Michael Stonebraker. Operating system support for database management. In *Communications of the ACM*, volume 24, pages 412–418, 1981.

[55] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software - Beyond Object-Oriented Programming – Second Edition.* Addison-Wesley/ACM Press, 2002.

[56] Jean-Philippe Fassino Tahar Jarboui and Marc Lacoste. Applying Components to Access Control Design: Towards a Framework for OS Kernels. In *International Conference on Dependable Systems and Networks (DSN 2004)*, 2004.

[57] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 110–119, 1994.

[58] Jean-Charles Tournier, Jean-Philippe Babau, and Vincent Olive. Qinna, a Component-Based QoS Architecture. In Lecture Notes in Computer Science 3489 Springer 2005, editor, *8th International Symposium on Component-Based Software Engineering*, pages 107–122, 2005.

[59] Yasuhiko Yokote. Kernel structuring for object-oriented operating systems: The apertos approach. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742. Springer-Verlag, 1993.

	Programming Paradigm	Granualrity	Depth	Time	Integrity	Application Domain
Choices	frameworks	mid	full	dvpt & run-time	framework enforcement	parallel systems
eCos	packages	coarse	medium	dvpt	constraints definition	embedded systems
Exokernel	libraries	coarse	small	dvpt	C compiler	high performance systems
K42	objects	mid	full	dvpt & run-time	itf type-checking	high performance systems
Linux	modules	coarse	big	dvpt & run-time	C compiler	GPOS
MetaOS	objects/meta-objects	fine	full	dvpt & run-time	policies replacement	GPOS
MMLite	components	fine	full	dvpt & run-time	itf type-checking	embedded systems
OSKit	components	coarse	medium	dvpt & boot-time	dedicated language	GPOS
Pebble	libraries	coarse	small	dvpt & run-time	multiple protection domains	embedded systems
Scout	components	coarse	medium	dvpt	itf type-checking	information appliances
Spin	modules	fine	full	dvpt & run-time	Modula-3 compiler + capabilities	GPOS
Think	components	fine to coarse	full	dvpt & run-time	dedicated language	dedicated systems
TinyOS	components	mid	full	dvpt	dedicated language	sensors network

Table 1: Synthesis of the evaluated operating system projects