# A Survey of Configurable, Component-based Operating Systems for Embedded Applications

Luis F. Friedrich, John Stankovic
Marty Humphrey, Michael Marley
John Haskins

Computer Science Report No. CS-2000-03
January 5, 2000

# A Survey of Configurable, Component-based Operating Systems for Embedded Applications

Luis F. Friedrich[1] , John Stankovic, Marty Humphrey, Michael Marley, John Haskins

Department of Computer Science
University of Virginia
Charlottesville, VA 22903
{fernando,stankovic,humphrey,mem5w,predator}@cs.virginia.edu

## Abstract

With the proliferation of embedded applications, criteria such as cost-effective variations of the product, flexible operation of the product, minimal time to market, and minimal product costs become deep concerns for embedded software industries. Component-based software is becoming an increasingly popular technology as a means for the construction of complex software systems by assembling off-the-shelf building blocks providing the ability to deal with customization and reconfiguration issues. However, many of the component-based methodologies utilize large components and do not address size, real-time performance, power, and cost issues. Another main problem with component-based systems is the configuration process itself. Issues such as the selection, parameterization of components, the analysis, and the choice of proper hardware and memory layouts are not fully addressed. This paper surveys the current state of component-based software and its utilization for the construction of operating systems for embedded applications and concludes with recommendations for further research in component-based operating systems.

**Index Terms**: Components, configurable, reconfigurable, operating systems, embedded systems.

---

[1] Research Fellow from Department of Computer Science, Federal University of Santa Catarina, Florianopolis, Brazil – supported by CAPES grant no. BEX0643/98-0 and by FULBRIGHT grant no. CCI 09/12 (e-mail: lff@inf.ufsc.br).

# 1. Introduction

Since the first microkernel appeared, improving modularity and flexibility of operating systems, there has been support for Application Specific Operating Systems (ASOS). This term is often used to refer to the ability for customization and reconfiguration in order to meet the requirements of specific applications or application domains. The general idea is to be able to provide lower cost and higher performance by eliminating general-purpose operating systems features that are unnecessary for the application and possibly tailoring included features to better suit the application being developed

Embedded applications are proliferating at an amazing rate with no end in sight. They are present in many industries such as telecommunications, automotive, consumer electronics, medical instrumentation and office automation. While each embedded applications is unique, their success generally depends on the same criteria such as cost-effective variations of the product, flexible operation of the product, minimal time to market and minimal product costs. In most embedded applications, the use of general-purpose operating systems (GPOS) platforms is not applicable because it is too expensive. Embedded system requirements such as processor performance, memory and cost are so variable that a GPOS cannot meet all the needs. Other approaches such as avoiding an OS altogether by implementing all the functionality directly, or by developing an in-house OS can limit flexibility and be expensive. However, a recent survey suggests that these approaches are used by 66% of the embedded systems in Japan [Takada, 1997], primarily because a suitable alternative does not readily exist. Therefore, a key challenge is to be able to provide an OS with a high degree of tailorability in order to execute the embedded system with the required functionality.

An example of a hypothetical product that would benefit from tailorable OS support is an embedded environmental control system for a smart home. This system would control the lights, temperature, and appliances by using dedicated micro-controllers (one or more), a small amount of memory, and real-time processing capabilities. An initial design of such a product would benefit by using an OS for the target micro-controllers that only contains those features necessary for the stated requirements of the application. For example, in this scenario there is no need for a file system, processes, networking, and security and protection facilities as provided by a GPOS. Now, suppose that market conditions warrant the addition of a burglar alarm system to the smart home control system, as well as the ability for different people to program different environmental settings into the unit over the Internet. This new product requires additional functionality over the original product, namely networking components, a file system for system logging, and possibly process support. Ideally, the product designer can add this new support easily and analyze the resulting collection of software in terms of correctness, cost, and speed. In addition, the product designer would like to quickly determine if it is possible to dynamically reconfigure the original product to handle this new functionality, so that consumers do not have to replace their original environmental control system. Many other embedded applications have similar requirements of low cost, high tailorability, quick time to the market and need for reconfiguration. A solution approach receiving increased attention is component- based software for embedded systems.

Component-based software is becoming an increasingly popular technology as a means for the construction of complex software systems by assembling off-the-shelf building blocks providing the ability to deal with customization and reconfiguration issues. Can this methodology be used effectively to support the delivery of cost-effective, high quality OS-like software entwined with application code for embedded reconfigurable systems? Obviously there are many advantages for component-based design. For example, component-based design minimizes the amount of new code that must be written when an application is developed and also provides the means for composition, which are essential ingredients for rapidly deployable embedded systems. In addition, an approach dealing with customization and reconfiguration issues allows fine tuning an embedded application. However, many of the component-based methodologies utilize large components and do not address size, real-time performance, power, and cost issues. Another main problem with component-based systems is the configuration process itself. Issues such as the selection, parameterization of components, the analysis, and the choice of proper hardware and memory layouts are not fully addressed. Finally, some future products will be multi-function; as such, dynamic mode switching and environment-specific tailorability will be needed.

The main purposes of this paper are to 1) discuss components in the specific domain of operating systems and embedded systems, 2) highlight the differences between software component-based technologies for general applications and for OS and embedded systems, 3) survey some of the systems and tools that have been developed for components in this focussed domain, and 4) to suggest future research needs.

## 2. Component Software Overview

In this section, a brief overview of component software technology and terminology is provided. A definition of components and important terms currently used in component-based software are introduced. There does not seem to be complete agreement and terms are badly overloaded. The literature offers a number of definitions of what a component is or should be [Booch, 1987][Nierstrasz et al. 1992] [Orfali et al., 1996] [Samentiger, 1997] [Short, 1997]. For instance, Clemens Szypersky [Szyperski, 1998] defines a software component as

*" a* **unit** *of composition with contractually specified* **interfaces** *and explicit* **context dependencies** *only. A software component can be* **deployed independently** *and is subject to* **composition** *by third parties."*

As an example, a component provides prefabricated functional building blocks to be reused by rearranging them in new compositions. One example might be using prefabricated avionics software in a complex command and control application. In other words, components can be thought of as building blocks or units of independent deployment, third-party composition, which have no persistent state (there is no differentiation between the component and its copies).

The majority of the definitions point out certain characteristics that are worthwhile repeating. First of all, the terms from the literature used to refer to a component (*unit, piece of software,* or *abstraction)* do not indicate any particular implementation technology. For instance, there is no need for a component to contain classes and be constructed using object technology although that is usually the case. It could contain traditional procedures or it may be realized using any other approach and provide its functionality using any technology. The term *unit* also does not provide any indication about the size of the component. However, there are hierarchies of components so size can vary considerably. Second, the term *independent deployment* refers to the fact that components are typically unaware of the contexts in which they may be used. In this case, to be able to deploy a component independently means that a component needs to be well separated from its environment and other components. Therefore, a component encapsulates its constituent features, and it will never be deployed partially. This requirement usually has performance implications and is one problem when trying to employ such components in an embedded system. Third, if a component is to be used for *composition* then it has to be sufficiently self-contained with a clear specification of what it requires and provides. In other words, a component has an *interface specification* that describes what the component does, and how it behaves when its functions or services are used. Through the specification, any potential user of those functions can use the component in his application without preoccupying himself with how those functions are actually performed. Also important, is that a component can be viewed as a *white-box* or *black-box* building block depending on the *visibility* that the users have of its interface implementation. If a user has access to a component's source code then it is said to be a white-box component, since it implies some degree of extension and customization. If, on the other hand, a component is available with no source code, and may be used just as it is, it is described as a black-box component. Finally, besides the specification of provided interfaces, components also are required to specify their resource and other needs. These needs are called *context dependencies*, referring to the context of composition and deployment required.

To set the stage for a discussion of components in the OS and embedded systems domains, three key issues are now discuss in more depth: component size, component interfaces, and component tools and infrastructures.

## 2.1 Component size

Although the terms used to refer to a component do not give any indication about the size of a component, the right size is one that makes it most useful. This means a component must have some quality issues such as correctness, robustness, careful specification and so forth. Also, a component must provide the 'right' set of interfaces without restricting context dependencies. For example, a component should provide all required software encapsulated in it, but this would increase the size of a component. On the other hand, a component could be designed to provide 'maximum reuse capabilities', with a probably increasing of context dependencies. As both approaches present inconveniences, there

has to be some balance in order to come up with the right size of a component. First, component-based architectures are considered modular and so naturally layered leading to a natural distribution of functionality. This modular approach makes the dependencies more explicit helping to reduce and control them. Therefore, modularity is a sort of pre-condition to be able to define components and its granularity. A system can readily be partitioned into units of varying size and coherence. Second, in order to achieve the best granularity of components the rules governing the partitioning vary from case to case and may depend on many different aspects such as abstraction, analysis, compilation, fault containment and loading [Szyperski, 1998]. Depending on these aspects a component could have different granularity. For example, as a unit of abstraction a component could be an abstract data type such as a stack or a queue while as a unit of fault containment and loading a component would be an entire file system.

Some new forms of partitioning have been proposed. For example, aspect-oriented programming [Kiczales, 1994] proposes to partitioning programs based on various aspects they address. Then, a weaver tool merges the aspect-oriented fragments into a whole. Weaving is a complex task as mutual dependencies among the fragments need to be respected.

## 2.2 Component Interfaces

A fundamental principle of component-based design is that a component has an interface. All connections between components occur through interfaces that can be defined as a set of functions that are invoked by other components. In order to be able to guarantee component independence, component software maintains a strict separation between the interface specification and the interface implementation. The **interface specification** of a component is a well-known contract specifying how a component's functionality is accessed. In addition, the specification provides the necessary information for both those implementing the interface and using the interface. Besides functional aspects an interface specification may also contain non-functional requirements such as performance. In order to develop useful interfaces, understanding the behavior of the participants of key activities in a domain is effective. In this case, component modeling and domain modeling are helpful. A domain model sets the context for the area being studied which can be a large area or a part of a specific application. The key thing about domain models is the possibility to point out and describe important components, their relationships and the meaningful collaborations between them in the domain of interest. In component modeling, the interactions between components can be analyzed and captured which is helpful for interface specification and its implementation. Interactions between components are called collaborations that may be complex, involving many parties and an agreed sequence of actions between them.

The main elements of an interface are its list of functions with the corresponding parameters expected from its callers and the specification model that provides the means by which each service may be understood. However, it might be necessary to have some more information about a component in order to be able to determine its behavior

[Beugnard et al., 1999]. In this case, besides the basic contract which is composed by the functions, parameters and possible exceptions an interface through its specification model can contain other three levels of contract, behavioral, synchronization and quality-of-service.

## 2.3 Components Tools and Infrastructures

To be useful, components must be implemented, assembled and interact with other components. Therefore, they require tools that may be specialized to component assembly and component construction and they also require some basic support structure (infrastructure) providing the means for their interaction.

First, it is helpful to know what kind of programming languages can be used in component software development and if there are some special requirements. For example, as component programming supports incremental loading of code, late binding has to be supported because interactions with other components need to be dynamic. Other features like polymorphism, information hiding and safety are also meaningful. According to [Szyperski, 1998], languages such as C, C++, Modula-2, or Smalltalk are not truly component-oriented programming languages because they lack the support for encapsulation, polymorphism, type safety, module safety, or any combination of these. However, almost any programming language can be used for developing components.

The development of component software appears to be more dependent on supporting tools. Although most of the traditional tools of the software engineering for design, implementation, maintenance will continue to be used, new tools will be necessary. Today, most of the tools concentrate on component assembly that is normally performed by instantiating and connecting component instances and by customizing component resources. Some assembly tools assume that all component instances have a visual representation at assembly time and then use powerful graphical builder tools to assemble components. An important aspect in the assembly process is that it should be automated and able to be repeated wherever a modification is necessary regarding the availability of future versions of components.

Finally, there needs to be some kind of environment which supports components conforming to certain standards and allow instances of these components to be attached into the component environment. This infrastructure should establish environmental conditions for component instances and to regulate the interaction between component instances. All popular component infrastructures provide mechanisms that allow development in multiple languages and execution across multiple hardware platforms. Examples of such infrastructures include CORBA (Common Object Request Broker Architecture)[OMG, 1995], COM (Component Object Model) [COM, 1995], DCOM (Distributed COM) [DCOM, 1998] and, Sun's Java Beans [JavaBeans, 1996]. As reusable components have been a trend in software engineering for some time, CORBA, COM, DCOM and Java Beans are all addressing these concerns. These systems serve important, but different needs than the ones being addressed in this paper. They provide a kind of macroscopic level infrastructure for component-based software.

More recently, the Jini architecture[JINI, 1999] has been defined to address the need for world-wide component pluggability into networks. In Jini's world, the components to be plugged into a network can be large software components, entire applications, hardware devices, and embedded systems. The Jini system depends on and works with Java and consists of sets of interfaces. These interfaces include distributed events, a 2-phase commit protocol, and those involved with resource allocation and reclamation. Also included is support to aid in supplying and finding services through lookup and discovery components. The system is very open ended as it needs to be to address world-wide networks and evolution. A key ability in Jini is the dynamic pluggability and concepts that support this capability may prove useful in some aspect of application specific operating systems where such kernels must support hot swappable software. A key difference between Jini and CORBA, for example, is Jini's ability to download code to the client that is then used to communicate with the server. This approach permits changes to servers to be evolvable and propagated to clients at the time they are to be used. Jini is also serving a different need than the one addressed in this paper.

## 3. Survey of Component-based OSs for Embedded Systems

Examples of component software outside of graphical user interfaces and compound documents are still rare. As stated by Bertrand Meyer [Meyer, 1999], "An area that is crying out for component-based development is the nec plus ultra of software: operating systems." This section presents a survey on systems that refer to component-based development as a design methodology and are in some sense centered in providing configurable OS for embedded applications. The survey includes academic and industrial systems. Most systems present some sort of ability for customization and reconfiguration to meet application specific requirements. The survey intends to investigate how the component software methodology has been used to provide configurable OS for embedded systems. Based on some of the terms and concepts on component software presented in section 2, we try to identify how each one of the systems deals with the following issues:

- What is a component? How are they defined?
- Is the system capable of doing performance analysis?
- How is the composition of system made?
- How are the components connected?
- Is dynamic reconfiguration possible?

In addition, some special features regarding embedded systems are identified.

## 3.1 Academic Systems

In this part of the survey we are interested in systems built in academia which are addressing issues like (re)configuration, composition of OSs, component based software for OS, and OS components for embedded applications. Recent projects such as Exokernel [Engler, 1995] and SPIN [Bershad, 1994], provide some form of OS configurability/extensability, allowing the OS to be tailorable to applications specific requirements. However, their configurability is limited in that they define a fixed amount of functionality that must be used in all the applications. In addition, these systems are not related to component-based software or embedded systems.

The surveyed systems include CHOICES, OS-Kit, COYOTE, PURE and, 2K. Most of these systems have in common the intention to deal with OS construction through composition. Therefore, they all try to define OS components. However, they use different design approaches and infrastructures.

Early systems like Choices [Camp, 1993] and OS-Kit [Ford, 1997] address OS configuration and customization issues as well as component software for OS. Choices uses a complex object-oriented framework in order to build an full OS. In contrast, OS-Kit provides a set of OS components that can be combined in order to configure an OS. The OS-Kit does not supply any rules to help build an OS. More recent systems like Coyote [Bhatti et al., 1998] address the problem of configuration and reconfiguration using approaches not based on object-oriented technology. Coyote is focussed on communication protocols. However, its ability for reconfiguration might be adopted for OS and embedded applications areas, hence we cover it here. Recent systems like PURE [Beuche et al., 1999] are explicitly concerned with providing OS components for configuration and composition of OS for embedded applications. PURE uses an object-oriented methodology to provide different components for configuration and customization of OS for embedded applications. Another recent system, 2K [Kon et al., 1998], is more concerned with adaptability issues in order to allow applications to be as customizable as possible. In addition, 2K is also concerned with component-based software for small mobile devices called PDA's (Personal Digital Assistant). Now consider each of these systems in more depth.

### CHOICES

Choices [Camp, 1993] is an object-oriented, customizable operating system whose main goal is to allow users of the system to easily optimize and adapt the system for specific application behavior and workloads. In order to allow customization, Choices uses frameworks and subsystems. The design of Choices is made by a hierarchy of frameworks representing the conventional organization of an operating system into layers. In Choices, a framework consists of a number of classes representing system entities such as disks, memory, schedulers, and so forth. For example, for the process subsystem there is a framework that is composed by classes such as *Process*, *ProcessContainer* and *ProcessManager* which define methods responsible for implementing the functionality of the framework. The *ProcessManager* class defines

methods for creating, suspending and killing processes. It also manages a global ready queue and the timeslice timer. These abstract classes can be thought of as components that can be configured in order to perform different roles. Customization is achieved by allowing subclassing the framework classes and overriding methods. For example, a *Process* component can incarnate the behavior of one of the following subclasses: *ApplicationProcess*, *SystemProcess*, *InterruptProcess* or, *Gang*. Classes belonging to a framework communicate with each other by calling methods. The interface of a framework is used by clients, which are entities outside the framework. Dynamic code loading is also provided by subclassing at run-time. As an example, the framework device management is composed by classes such as *DeviceController* and *Device*. If a new device driver needs to be added it can be done by loading subclasses of the *DeviceController* and the *Device* classes.

Regarding configurability and composition, Choices provides an interactive graphical tool, OS View, which allows both system and user-level services to be dynamically reconfigured, customized and evaluated. The viewer of Choices explores the system scanning and browsing of all operating system objects, which are represented as graphics images. In addition, alternative services can be loaded or activated in the system. For example, in the memory management framework different page replacement policies can be interactively loaded and evaluated through performance statistics. Therefore, it also provides performance information. No special features exist for embedded systems. A licensed release of Choices can be obtained at http://choices.cs.uiuc.edu/choices/.

## OS-Kit

The University of Utah's OS-Kit [Ford, 1997] is a domain-specific set of software components intended to facilitate construction of stand-alone systems on Intel x86 hardware. The authors argue that the boring details of constructing stand-alone systems are more easily handled through the OS-Kit components, thus freeing developers to perform research on their intended area of focus.

An example of a component in the OS-Kit is the Ext2 file system - a subset of the larger Linux legacy code portion of the OS-Kit. The Ext2 subset is an independently-deployable unit with no persistant state*. The authors of the OS-Kit went to great lengths to minimize the number of interactions and dependencies between components. This increases flexibility between the components and flexibility between the components and code created independently by the kernel developer. To provide usability OS-Kit adopted a subset of COM as the basic framework allowing components to interact each other efficiently through well-defined interfaces. However, in COM-based systems components run within an address space with no protection between them.

Analysis capability in the OS-Kit is provided by the profiling component library. This segment allows the kernel developer to link an instance of the gprof program directly into

---

* *Note: To say that the Ext2 subset has no persistent state is not a contradiction. Instances of file systems maintain persistent data; the Ext2 code subset of the OS-Kit is merely the blueprint for constructing and maintaining an instance of an Ext2 file system.*

the kernel. gprof performs its data reduction and analysis of the kernel immediately before the kernel exits and produces its output to the console. A minimal API of functions is also provided to control the profiling during the execution of the kernel.

Finally, composition in the OS-Kit is left solely to the kernel developer; there are no tools to help with this. The OS-Kit is essentially a collection of code segments that must be integrated and connected manually by a third party. This system does not focus on embedded systems. OS-Kit provides open source code that can be downloaded from its web page http://www.cs.utah.edu/projects/flux/oskit/.

## COYOTE

The purpose of Coyote [Bhatti et al., 1998] is to support the construction of communication protocols such as atomic multicast, group RPC, group membership, and protocols needed for mobile computing. In Coyote, those protocols are called composite protocols and represent one of the fundamental components of the system. Others components are: micro-protocols, events and a run-time system. Composite protocols assume a typical hierarchical communications protocol stack. They are considered a coarse-grain module. Within each level of this protocol stack, Coyote supports the non-hierarchical construction of that layer using micro-protocols. Examples of micro-protocols include message ordering schemes or retransmission policies. Micro-protocols are considered fine grained modules which can be registered to handle events. Hence, in this system a component is a layer of a communication protocol stack and a micro-component is a micro-protocol used to construct a particular layer of the system. Events in Coyote are responsible for initiating execution activity within a composite protocol. They can be detected and raised by the run-time system or by micro-protocols. The run-time system is responsible for managing execution and implementation of the event mechanism. Basically, it provides a kind of storage for the messages and allows multiple micro-protocols to access them. While embedded systems might require communication services it seems that most would not employ the type of communications supported by Coyote. It would be interesting to develop a set of lower level and efficient protocols for embedded systems using the Coyote approach. On the other hand, such features as multicast and mobile computing support might be useful for some types of embedded systems.

In Coyote there are no analysis tools to determine correctness or performance. Composition tools are minimal in that protocols and micro-protocols are in files and they are composed off-line by the designer. The overall framework that supports this system is contained in read only files; there are user modifiable files which contain library like functions and standard protocols and policies; and there can also be user supplied files with their own developed micro-protocols. A key aspect of configurability in Coyote is the support for events and event handlers. It is the raising of an event which causes execution of a micro-protocol. Reconfiguration is supported by the binding and unbinding of event handlers.

## PURE-Portable, Universal Runtime Executive

The PURE [Beuche et al., 1999] system approach is developed to offer an operating system tailored to the application. The goal is to construct a highly configurable system providing the means for the application designer to choose the needed functionality. Although PURE claims not to be restricted to any application area, its main focus is on "deeply embedded systems". The term is used to refer to systems operating under extreme resource constraints in terms of memory, CPU and power consumption.

The design approach of PURE is based on two main concepts: *program family* and *object orientation*. The program family concept is used in order to provide a sort of hierarchical design in such a way that a "minimal subset of system functions" is used as a platform to implement extensions or "minimal system extensions". Object-Oriented is used as the implementation discipline.

Partitioning in Pure is based on abstractions and the units used to build a system may have different granularity and complexity. The smallest building unit is a class. Therefore, Pure can be viewed as a class library. For example, the building unit responsible for thread control is composed by 45 classes arranged in a 14-level hierarchy. Some of these classes are: *Counter* – implementing a waiting list of threads, *Schemer* – implementing thread scheduler, *Monitor* – provides per thread synchronized operation of some critical functions, *Filing* – provides the means to keep track of the allocated threads, *Active* – currently executing thread. These classes can be customized to meet application specific requirements. Although classes in Pure are claimed to be very fine grain some like the *Schemer* are coarse grain. The components in Pure are arranged in a structure made of a nucleus and a nucleus extension. The nucleus, called CORE (Concurrent Runtime Executive) is responsible for the implementation of a "minimal subset of system functions" for scheduling of interrupts and threads. The CORE is made of four building units. These units can be composed in order to provide the desired functionality. For example, one can have a minimal system only supporting low level trap/interrupt handling. Features that represent some kind of extension, called "minimal system extensions" are added to the system in the nucleus extension, called NEXT. While not explicit, the highly configurable and fine-granular structure provided in this system allows for better support embedded applications requirements such as memory and time requirements. However, it is not clear how it is determined if the requirements are met. A key aspect of embedded applications, interrupt handling, is also specifically addressed by Pure.

Pure does not provide any kind of analysis tools to determine performance. For configuration purpose Pure provides tools that allow users to specify their needs and requirements for the customized system. The approach is to use an annotation language to provide the necessary information such as dependency and attributes, for the generation tool to be able to evaluate and choose the right building units for combination. The result of the configuration process is a sort of *make* file which produces the desired system. Pure is not available as source code.

### 2K

The 2K [Kon et al., 1998] system is a reflective, component-based operating system whose main goal is to provide a generic framework in order to support adaptation in a network-centric computing environment. The ability of 2K for adaptation is based on parameters such as network bandwidth, connectivity, memory availability, communication protocols and hardware components. The 2K operating system is built on top of CORBA. It uses reflection (meta-level data and methods on that data), offered at the ORB level, in order to provide the means for adaptation.

A component in 2K is a dynamically loadable unit which is stored in a library (DLL). The components can be loaded/unloaded depending on the user needs and the garbage collection algorithms. As an example, a PDA after being integrated in the 2K environment can select a category of components such as spreadsheets in order to interact with the components belonging to that category (Excel, Lotus). Based on this it seems that the granularity of the components supported by 2K is coarse grain. However, a component can also be responsible for a discipline such as thread pool or thread per connection (100 lines of code) in order to implement a concurrency policy.

There is no analysis tool to determine correctness and performance. However, components can access the state of the system to determine if they need to adapt. The system provides (re) configuration capabilities based on prerequisite dependencies and dynamic dependencies. The prerequisites for a component are a specification of requirements such as hardware resources and software services that are necessary in order to load, configure and execute the component. Dynamic dependencies describe the dependencies between a particular component and other components in a running system. To provide this information, an object called *ComponentConfigurator* is assigned to each component. In addition, 2K enables adaptation through allowing system components to reason about their interactions with other components and make adaptation decisions. It seems like the contribution of 2K for embedded application is the mechanism it uses to provide adaptation/reconfiguration capabilities. Software and documentation related to 2K is available at http://choices.cs.uiuc.edu/2K/.

## 3.2 Industrial Systems

There are about 100 real-time, embedded operating systems in the market. Many of them do not provide configuration capabilities or are not customizable. Others such as QNX [Hildebrand, 1992] and VxWorks[WindRiver, 1995] provide optional modules that can be statically or dynamically linked to the OS. However, these modules rely on a basic kernel and are not designed using a component-based approach. The modules are designed with no intention to be used in other environments. The survey of industrial systems include JavaOS, Jbed, icWORKSHOP, MMLite, Pebble, and, eCos. Some of these are already products (JavaOS, JBed and icWORKSHOP) while others are still under development (MMLite, Pebble and eCos). A common feature of all these systems

is that they are intended to provide a component based OS approach for embedded applications. However, they use different approaches to provide interaction between components.

JavaOS [JavaOS, 1999], developed by Sun Microsystems, Inc., and IBM and JBed [Oberon, 1998], developed by Oberon Microsystems, are basically designed for Java technology. On the other hand, MMLite [Helander, 1998], supported by Microsoft, uses a light-weight Component Object Model (COM) as its component infrastructure. Other systems like Pebble [Gabber et al., 1999], IcWORKSHOP [Chipware, 1999] and, eCos [eCos, 1999] are not based on any popular component infrastructures. Pebble, supported by Lucent Technologies – Bell Laboratories, intends to provide a component infrastructure that is based on protection domains and portals. However, its main concern is not embedded systems. IcWORKSHOP, developed by Integrated Chipware, is designed for building component-based ASOS (Application Specific Operating System) for ASSP (Application-Specific Standard Processors) hardware. The purpose of eCos, develop by Cygnus, is to be an open embedded software infrastructure. Lets now consider each of these systems in more depth.

### Java-OS

Java-OS [JavaOS, 1999] is an operating system specifically developed for embedded systems and network computers. Actually, there are three different Java-OS available: for Business, for Consumers and, for Network Computers ( http://www.sun.com/javaos/ ).In general, Java-OS has a database of configuration information consisting of named Java objects. This database helps support dynamic reconfiguration. Information in this database includes application-specific settings, which devices are present, and which software components must be installed for a user. As an example, if a new device is added the OS detects this fact, it adds an entry into the configuration database and fires a configuration change event.

Java-OS consists of a boot loader, a microkernel, and a runtime. The booter loads the Java-OS (possibly off the network) and activates the microkernel. A basic microkernel is always included. The microkernel includes support for threads, low level memory management, timers, interrupts and monitoring. The microkernel does not support multiple address spaces nor IPC. The microkernel also supports a runtime environment set of services. This includes the Java virtual machine, a garbage collector, a service loader, and core classes. This can be considered a fairly large set of required functionality as compared to the icWorkshop approach described below. While Java-OS claims are made for real-time and embedded systems performance, it is not clear how one performs a global analysis on the resultant system to determine if memory, power, and timing constraints are met.

### JBed

Jbed [Oberon, 1998] is a real-time operating system with a kernel designed for embedded Java. It is considered a Java platform for real time and embedded systems. Others Java

platforms which appear as candidates are Embedded Java and JavaCard. The basic differences between the platforms are their support for threads, garbage collection, floating point numbers, and so forth. In JBed, the kernel is actually the Java Virtual Machine and it is also called the run-time system or kernel level. The JBed run-time system provides a thread scheduler, memory allocation, and garbage collection as a minimal system. This configuration requires up to 64KB RAM. Other possible configurations include the minimal system with TCP/IP and a web server (128KB) and the minimal system with network loader and a flash compiler (for translation of Java code into machine code on the target, upon loading) (256KB). It appears that in terms of OS kernel configuration these are the options available, meaning that OS kernel components are coarse grained. In addition, there are no dynamic configuration capabilities at that level. At the kernel level, JBed is concerned with some special features for real-time and embedded applications such as small memory footprint, real time thread support and deadline scheduling. However, it is not clear how it determines if requirements such as memory and timing are met. On top of the kernel level, components such as peripheral device drivers, communication device drivers, network loader and libraries are supported. These components are called embedded applications and can be downloaded on demand. Therefore, at this level JBed provides dynamic configurability. Finally, the application layer support a client/server model. At this layer, applications such as process control, remote diagnosis, alarm systems, are called clients. The clients use the components (embedded applications) through server programs such as debugging, remote control, Web server, which provide management of those components. According to JBed documentation, servers allow clients to perform remote diagnosis of embedded applications, replace components in the field and remotely control embedded systems from a PC. However, these capabilities are not available at OS kernel level.

JBed provides a development tool, JBed IDE, which provides a convenient cross-development and visualization environment for the configuration of an embedded application. At the kernel level, it seems that the configuration is just a matter of choosing one of the available configurations mentioned above.

### MMLite

MMLite [Helander, 1998] is an object-based, modular system architecture that provides a menu of components that can be chosen at compile-time, link-time, or run-time to construct a wide range of applications. A component in MMLite consists of one or more objects. Multiple objects may reside in a single namespace; when an object needs to send a message to an object in another namespace for the first time, a proxy object is created in the sending object's namespace that transparently handles the marshalling of parameters.

A unique aspect of MMLite is its focus on support for transparently replacing components while these components are in use (Mutation). MMLite uses Component Object Model (COM) interfaces, which in turn support dynamic reconfigurability on a per-object and per-component basis. However, COM does not provide protection between the components, in MMLite it is not clear if it provide isolation between components or not. The base menu of the MMLite system contains components for heap

management, dynamic on-demand loading of new components, machine initialization, timer and interrupt drivers, scheduler, threads and synchronization, namespaces, file system, network, and virtual memory. These components are typically very small (500-3000 Bytes on x86), although the network component is much larger (84,832 Bytes on x86). The resulting MMLite system can be quite small: the base system on x86 is 26KB, and 20KB on ARM. It is not clear to what extent the goal of MMLite is to provide users the ability to easily select components that they -- the MMLite developers -- write, and to what extent users themselves are intended to define and utilize their own new components. Although there has been an apparent emphasis on developing minimal-sized components (in number of bytes), analysis tools regarding the run-time performance of components due to namespace resolution and the creation and loading of proxy objects is lacking.

## Pebble

Pebble [Gabber et al., 1999] is a new operating system designed for both efficiently being an application- specific operating system and to support component-based applications. It also intends to support complex embedded applications. As an operating system it adopts a micro-kernel architecture with a minimal priviledged mode nucleus that is only responsible for switching between protection domains. The functionality of the operating system is provided by operating system user-level components (servers). These components can be replaced, augmented, or layered. The programming model is Client/Server, client components (applications) request services from system components (servers). Example of system components are the interrupt dispatcher, scheduler, portal manager, device driver, file system, virtual memory, etc. For instance, the Pebble kernel and the essential components (interrupt dispatcher, scheduler, portal manager, real-time clock, console driver and the idle task) need something like 560KB of memory. Components are like processes, each one executes in its own protection domain (PD). In Pebble a PD includes a page table and a set of portals. Portals provide the communication between PD's. For example, if there is a portal from $PD_1$ to $PD_2$ then a thread executing in $PD_1$ can invoke a specific service (entry point) of $PD_2$. Therefore, components communicate through transferring threads from one PD to another using portals. The PD concept together with the portal concept can be understood as a component infrastructure, while Pebble PD's provides the means to isolate the components portals provide the means for components to communicate each other. Instantiation and management of portals are done by an operating system component, Portal Manager. For instance, the instantiation process involves the registration of a server (any system or application component) in a portal and the request of a client for that portal. In Pebble, it is possible to dynamically load and to replace system components in order to fulfill applications requirements.

Based on the description of the system there are no composition tools in order to provide the construction of the system. Also, there are no analysis tools to determine correctness or performance. It seems that what makes Pebble as an operating system for embedded applications is its capability for dynamic configurability and its ability to safely run untrusted code.

## icWORKSHOP

Real-time operating system vendors have provided tailorable kernels for some time. However, the degree of tailorability has been at a fairly high level (e.g., you can choose to include a file system or not) and static. The icWORKSHOP [Chipware, 1999] from Integrated Chipware allows the rapid customization of a real-time operating system from small granule components. The system components are collected into a toolkit called icPARTS. A list of their components includes: tasks, queues, timers, file management, clocks, semaphores, error handling, I/O, sockets, interrupt handling, IPC, floating point, mutexes, pipes, condition variables, buffer management, schedulers, board configuration classes, linked lists, memory management, networking, kernel locks, and directories. From this list you can see that components vary in functionality and size quite a bit, but they are much lower level than you might find for CORBA components or even for OSKit components. Users can tailor any of these pre-defined components or add new ones. For example, a user might add an avionics or telecom specific component. While not explicit, the fine granularity involved in this system allows for better control over issues such as low overhead to meet time requirements, handling interrupts quickly, performing a more global analysis of memory and time requirements, and control over device drivers. The domain for their product is then any application and embedded system that might require a tailored real-time operating system.

They do provide 3 ready to run kernels as well as multiple options for various activities such as scheduling. icWORKSHOP also includes a development tool called icBUILD. This tool contains support for composing an application specific operating system and various visualization and performance monitoring capabilities. The claim is that it is geared for custom real-time development. This claim seems to stem from the level of components available as well as a software logic analyzer that helps in performance measurement. Browsers and editors are supplied to let a designer view and modify components and incorporated into a functioning kernel via button clicks (according to the company). Debuggers and statistical profilers are part of the analysis capabilities. It does not seem that this system supports dynamic reconfigurability.

## eCos - Embedded Cygnus Operating System

eCos [eCos, 1999] is an embedded operating system that was designed and constructed to provide a standardized framework to be used in the creation, extension and configuration of embedded system software. The extensibility of the system is achieved through the use of open published APIs. This allows developers to extend the core components and develop new or modified components.

The building unit in this system varies from a package (coarse grain component) to a configuration option (very fine grain component). The configurability of the system is achieved by selecting package options as well as including and configuring components within a package. The system is composed of packages such as the kernel, the uITRON compatibility layer, hardware abstraction layer, C library, watchdog device and I/O Sub-system. Each package can contain any combination of configuration options,

components, or even another package. For example, options such as 'NULL is a good pointer' and 'Return Error Codes for Bad Params' as well as components such as semaphores, mailboxes, event flags, alarm handlers, and version information can be combined for the package uITRON compatibility layer. In addition, eCos supports configurability at multiple levels. On the highest level, components can be switched in and out. For example, semaphores could be included or not in the uITRON package. On a lower level, options regarding the components can be configured - which they have termed micro-configuration. For example, the modification of the kernel to support 16 versus 32 priority levels would be considered a micro-configuration. And then on the lowest level, the source code itself can be modified to make even the smallest of changes.

The eCos system also has a good deal of tool support. It includes a configuration tool that can be used to select components and configure those components to get the precise functionality that is desired. The configuration tool provides a graphical user interface that allows the developer to easily select options and configure the system. Once the system is configured, the tool generates a build tree containing header files that is used to build the system. During this process, the tool does dependency checks of the components that are included in the system, as well as dependency checks and component requirements for the application code. It should be apparent from the way in which a system is built that this is only a static framework. There is no support for dynamically loading components during run time.

In addition to the configuration tool, eCos provides test cases for each configurable feature. The test cases can be run to verify the validity of the system. These test cases are automatically linked into the system during build time. Currently eCos has developed an automated testing infrastructure, but have not released it to the public at this point. Although eCos is a commercial system, its source code can be downloaded from http://www.cygnus.com/ecos/.

## 3.3 Summary of Academic and Industrial Systems

After reading all the above summaries it is possible to make the following observations. While somewhat simplistic, to better understand software components for embedded systems, it is possible to consider component based systems at three levels. At the highest level systems such as CORBA, COM, and JavaBeans often use large scale components. Entire systems or subsystems are used and the components themselves are often applications. The cost of communications between interfaces is generally high, but due to the large scale aspects of the components this is not usually a problem. These types of components are not the main focus of this paper. At the second level, component based configuration of OS functionality such as that supported by OSKit are used. Here components deal with typical OS functionality, but do not really attempt to support embedded systems. Finally, at the third level, fine grained components that include OS-like functions and user supplied application specific functions such as those for avionics or telecommunications are used. The components attempt to support low overhead interfaces, sensors, actuators, fast interrupt handling, etc. Therefore, regarding the size of

the components it is possible to conclude that there is no common definition. Based on the reviewed systems, most of the them use components with different granularities. Analysis tools on how well the configured system will meet time, memory and power constraints seem very limited. A few systems provide profiling and debugging capabilities. Tools for the actual functional configuration seem good for some of the industrial products. Overall, it seems that research is needed more in the development of configuration tools and accompanying analysis than in the construction of the components themselves.
Regarding the infrastructure supporting the connection of the components, many systems are based on a kernel approach. There is a kernel that is responsible for the connection of the components. In this case, the size and functionality of the kernel is variable. Other systems use COM-based interfaces and Java Virtual Machine as a mean to connect the components. In most systems the components are classified as user (U) and operating system (OS) components. Among the systems that provide reconfiguration capabilities, a few provide it for OS components.

Table 1 provides a summary of the major issues being investigated in using component-based software for OS and embedded applications.

| System | Analysis Capabilities | Composition Tools | Infrastructure | Reconfiguration capabilities |
|---|---|---|---|---|
| CHOICES | OS View | OS View | OO-based Framework | Dynamic loading of OS-classes |
| OS-Kit | Profiling | Not available | COM-based Framework | Not supported |
| COYOTE | None | Minimal | x-kernel | Dynamic change of Event handlers |
| PURE | None | Annotation language | Minimal kernel OO-based | Not supported |
| 2K | None | Not available | CORBA-based Reflective ORB | Dynamic loading of U-components |
| JavaOS | None | Not available | JVM | Dynamic detection of drivers |
| Jbed | None | JBed IDE | JVM | Downloading of U-components |
| MMLite | None | Not available | COM-based Framework | Replacement of OS-components |
| Pebble | None | Not available | Minimal kernel | Dynamic loading of OS-services |
| IcWORKSHOP | Debugging and profiling | IcBUILD | Kernel-based | Not supported |
| Ecos | Test cases to verify validity | GUI generated header files | Kernel-based | Not supported |

Table 1. Summary of the systems

## 4. Research questions

While we are beginning to see many projects and products addressing the need for component based development for embedded and real-time systems, key research questions still exist. We separate the research questions into 3 main areas:

- Software components themselves,
- Dynamically reconfigurable hardware components, and
- The configuration process.

Software component research questions include:

- development of lightweight interfaces,
- defining metrics and developing techniques for categorizing components along memory size, execution time, fault tolerance, security, and QoS dimensions (i.e., non-functional attributes), and
- development and saving of configuration information about components such as constraints (e.g., component A must be used with component B and C and not with D; only works on certain hardware, etc.).

Dynamically reconfiguring hardware components will become more prevalent with the availability of new FPGAs. While this will increase the flexibility and performance of embedded systems, it gives rise to the following research questions:

- what impact will the changed hardware have on the OS components and on the application code for both the functional and non-functional attributes, and
- how to determine when the performance gain of dynamic configuration is worth the cost.

The configuration process is perhaps the area that has received the least attention, but it is critical. Some of the key research questions are:

- how to guide the developer to choose the "right" components to meet all the requirements of space, time, cost, power, and speed to market, and
- how to analyze the resultant collection of components for correctness, for meeting deadlines, and meeting other requirements. Analysis tools are critical.

## 5. References


[Bershad, 1994] Bershad B., Chambers C., Eggers S., Maeda C., McNamee D., Pardyak P. Savage S., Sirer E. (1994) SPIN – An Extensible Microkernel for Application-specific Operating System Services. University of Washington Technical Report 94-03-03.

[Beuche et al., 1999] Beuche D., Guerrouat A., Papajewski H., Schroder-Preikschat W., Spinczyk O., and Spinczyk U. (1999) The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. *Proceedings of 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France.

[Beugnard et al., 1999] Beugnard A., Jezequel J., Plouzeau N. and, Watkins D. (1999) Making Components Contract Aware. *Computer*, **32**(7), 38-45.

[Bhatti et al., 1998] Bhatti N., Hiltunen M., Schlichting R., and Chiu W. (1998) Coyote: A System for Constructing Fine-Grain Configurable Communication Services. *ACM Transactions on Computer Systems*, **16**(4), 321-366.

[Booch, 1987] Booch G.(1987) *Software Components with Ada: Structures, Tools and Subsystems*. Benjamin-Cummings, Redwood City, CA.

[Camp, 1993] Campbell R., Islam N., Madany P., and Raila D. (1993) Designing and Implementing Choices: an Object-Oriented System in C++. *Communications of the ACM*, September 1993.

[Chipware, 1999] Integrated Chipware IcWorkShop (http://www.chipware.com/).

[COM, 1995] Microsoft Corporation and Digital Equipment Corporation (1995) *The Component Object Model Specification*. Redmond, Washington.

[DCOM, 1998] Microsoft Corporation (1998) *Distributed Component Object Model Protocol, version 1.0*. Redmond, Washington.

[eCos, 1999] Cygnus (1999) eCos – Embedded Cygnus Operating System. Technical White Paper (http://www.cygnus.com/ecos).

[Engler, 1995] Engler D., Kaashoek M.F. and O'Toole J. (1995) Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proceedings of the 15th SOSP*, Copper Mountain, CO.

[Ford, 1997] Ford B., Back G., Benson G., Lepreau J., Lin A., and Shivers O. (1997) The Flux OSKit: A Substrate for Kernel and Language Research. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France.

[Gabber et al., 1999] Gabber E., Small C., Bruno J., Brustoloni J., and Silberschatz A. (1999) The Pebble Component-Based Operating System. *Proceedings of the USENIX Annual Technical Conference*. Monterey, California, USA.

[Helander, 1998] Helander J. and Forin A.(1998) MMLite: A Highly Componentized System Architecture. *Proceedings of the Eighth ACM SIGOPS European Workshop*. Sintra, Portugal.

[Hildebrand, 1992] Hildebrand D. (1992) An Architectural Overview of QNX. *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*. Seattle, WA.

[JavaBeans, 1996] Sun Microsystems (1996) *JavaBeans, version 1.0*. (http://java.sun.com/beans).

[JavaOS, 1999] Saulpaugh T. and Mirho C. (1999) *Inside the JavaOS Operating System*. Addison Wesley, Reading, Massachusetts.

[JINI, 1999] Arnold K., O'Sullivan B., Scheifler R.W., Waldo J., and Wollrath A.(1999) *The Jini Specification*. Addison-Wesley

[Kon et al., 1998] Kon F., Singhai A., Campbell R. H., Carvalho D., Moore R., and Ballesteros F. (1998) 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. *Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*. Brussels, Belgium. July 1998.

[Meyer, 1999] Meyer B. and Mingins C. (1999) Component-Based Development: From Buzz to Spark. *Computer*, **32**(7), 35-37.

[Nierstrasz et al.,1992] Nierstrasz O., Gibbs S., and Tsichritzis D. (1992) Component-oriented software development. *Communications of the ACM*, **35**(9), 160-165.

[Oberon, 1998] Oberon Microsystems (1998) Jbed Whitepaper: Component Software and Real-time Computing. *Technical Report*. Zurich, Switzerland (http://www.oberon.ch).

[OMG, 1997] Object Management Group (1997)*The Common Object Request Broker: Architecture and Specification*, Revision 2.0, formal document 97-02-25 (http://www.omg.org).

[Orfali et al., 1996]Orfali R., Harkey D., and Edwards J. (1996) *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, New York.

[Samentiger, 1997] Samentiger J. (1997) *Software Engineering with Reusable Components*. Springer-Verlag, Town.

[Short, 1997] Short K. (1997) Component Based Development and Object Modeling. Sterling Software (http://www.cool.sterling.com).

[Szyperski, 1998] Szyperski C. (1998) *Component Software Beyond Object-Oriented Programming*. Addison-Wesley, ACM Press, New York.

[Takada, 1997] Takada H. (1997) μITRON: A Standard Real-Time Kernel Specification for Small-Scale Embedded Systems. Real-Time Magazine, issue 97q3.

[WindRiver, 1995] Wind River Systems, Inc. (1995) *VxWorks Programmer's Guide*.