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

COMPONENT-BASED SOFTWARE IS BECOMING AN INCREASINGLY POPULAR TECHNOLOGY AS A MEANS FOR CREATING COMPLEX SOFTWARE SYSTEMS BY ASSEMBLING OFF-THE-SHELF BUILDING BLOCKS. HOWEVER, MANY OF THE COMPONENT-BASED METHODOLOGIES THAT USE LARGE COMPONENTS FAIL TO ADDRESS ISSUES OF SIZE, REAL-TIME PERFORMANCE, POWER, AND COST, AS WELL AS PROBLEMS ASSOCIATED WITH THE CONFIGURATION PROCESS ITSELF. THESE ISSUES ARE CRITICAL FOR USING COMPONENTS IN EMBEDDED SYSTEMS.

L. Fernando Friedrich
Federal University of Santa Catarina, Brazil

John Stankovic
Marty Humphrey
Michael Marley and
John Haskins Jr.
University of Virginia

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 to meet the requirements of specific applications or application domains. The general idea is to provide lower cost and higher performance by eliminating general-purpose operating system 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 application is unique, their success generally depends on the same criteria, such as cost-effective variations and 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 operating system altogether by implementing all the functionality directly, or by developing an in-house operating system, can limit flexibility and be expensive. However, a recent survey suggests that these approaches are used by 66 percent

0272-1732/01/$10.00 © 2001 IEEE

of the embedded systems in Japan,[1] primarily because a suitable alternative does not readily exist. Therefore, a key challenge is to provide an operating system with a high degree of tailorability to support the embedded system with the required functionality.

An example of a hypothetical product that would benefit from tailorable operating system support is an embedded environmental control system for a smart home. This system would control lights, temperature, and appliances by using dedicated microcontrollers (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 operating system for the target microcontrollers that only contains those features necessary for the stated requirements of the application. For instance, in this scenario there is no need for a file system or for processes, networking, or 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 should be able 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 market, and need for reconfiguration. A solution approach receiving increased attention is component-based software for embedded systems.

Can this methodology be used effectively to support the delivery of cost-effective, high-quality, operating system-like software entwined with application code for embedded reconfigurable systems? Obviously there are many advantages to component-based design. For example, it minimizes the amount of new code that must be written when an application is developed and also provides the means for composition; 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 selection, parameterization of components, analysis, and choice of proper hardware and memory layouts are not fully addressed. Finally, some future products will be multifunction; as such, dynamic mode switching and environment-specific tailorability will be needed.

Component software

The literature reflects a lack of agreement and an overloading of component software terminology, offering a number of definitions of what a component is or should be.[2-6] For instance, Clemens Szypersky 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. [7]

For 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 used for third-party composition and having no persistent state (there is no differentiation between the component and its copies).

The majority of the definitions point out certain characteristics that are worth repeating. First of all, terms in the literature that 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 tech-

nology, although that usually is the case. It could contain traditional procedures or it might be realized using any other approach and provide its functionality using any technology. Also, the term *unit* 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 can 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.

Component size

Although the terms used to refer to a component do not give any indication about its size, 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 its size. On the other hand, a component could be designed to provide maximum reuse capabilities with a likely increase of context dependencies. As both approaches present inconveniences, there has to be some balance in order to come up with the right size 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 precondition to defining components and their granularity. A system can readily be partitioned into units of varying size and coherence. Second, 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.[7] 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 could be an entire file system.

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 invoked by other components. 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 nonfunctional requirements, such as performance.

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, which 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 more information about a component to determine its behavior.[8] In this case, besides the basic contract, which is composed of the functions, parameters, and possible exceptions, an interface through its specification model can contain another three levels of contract: behavioral, synchronization, and quality of service.

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 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, such as polymorphism, information hiding, and safety also are meaningful. 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.[7] 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 software engineering for design, implementation, and maintenance will continue to be used, new tools will be necessary. Today, most of the tools concentrate on component assembly 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 repeatable wherever a modification is necessary regarding the availability of future versions of components.

Finally, there needs to be some kind of environment that supports components conforming to certain standards and allows instances of these components to be attached into the component environment. This infrastructure should establish environmental conditions for component instances and 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), COM (Component Object Model), DCOM (Distributed COM), and Sun's JavaBeans.[9-12] As reusable components have been a trend in software engineering for some time, Corba, COM, DCOM, and JavaBeans all address these concerns. These systems serve important, but different needs than the ones addressed in this article. They provide a kind of macroscopic-level infrastructure for component-based software.

More recently, the Jini architecture has been defined to address the need to plug components worldwide into networks.[13] 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 two-phase commit protocol, and various functions 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 worldwide networks and evolution.

A key ability in Jini is the dynamic plug in ability and concepts that support this capability, which 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 be propagated to clients at the time they are to be used. Jini is also serving a different need than the one addressed in this article.

Component-based operating systems for embedded applications

Examples of component software outside of graphical user interfaces and compound documents are still rare. As stated by Bertrand Meyer, "An area that is crying out for component-based development is the nec plus ultra of software: operating systems."[14] 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 operating systems 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 is meant to investigate how the component software methodology has been used to provide configurable operating systems for embedded systems. Based on some of the terms and concepts on component software presented earlier, we try to identify how each one of the systems deals with the following issues, along with identifying some special features regarding embedded systems:

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

Academic systems

We are interested in systems built in academia that address issues such as configuration and reconfiguration, composition of operating systems, component-based software for operating systems, and operating system components for embedded applications. Recent projects such as Exokernel[15] and SPIN[16] provide some form of operating system configurability/extensability, allowing the operating system to be tailorable to application 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 operating system construction through composition. Therefore, they all try to define operating system components. However, they use different design approaches and infrastructures.

Early systems like Choices and OS-Kit address operating system configuration and customization issues as well as component software for operating systems.[17-18] Choices uses a complex object-oriented framework to build a full operating system. In contrast, OS-Kit provides a set of operating system components that can be combined to configure an operating system. OS-Kit does not supply any rules to help build an operating system. More recent systems such as Coyote address the problem of configuration and reconfiguration using approaches not based on object-oriented technology.[19] Coyote is focused on communication protocols. However, its ability for reconfiguration might be adopted for operating system and embedded application areas, hence we cover it here.

Another recent system, PURE, is explicit-

ly concerned with providing operating system components for configuration and composition of operating systems for embedded applications.[20] PURE uses an object-oriented methodology to provide different components for configuration and customization of operating systems for embedded applications. Another recent system, 2K, is more concerned with adaptability issues to allow applications to be as customizable as possible.[21] In addition, 2K is also concerned with component-based software for small mobile devices, or PDAs (personal digital assistants).

Choices. Choices is an object-oriented, customizable operating system whose main goal is to allow users to easily optimize and adapt the system for specific application behavior and workloads. To allow customization, Choices uses frameworks and subsystems. The design of Choices consists of 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 of classes such as Process, ProcessContainer, and ProcessManager that 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 time slice timer. These abstract classes can be thought of as components that can be configured to perform different roles.

Customization is achieved by allowing subclassing of the framework classes and by 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 runtime. 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 Choices viewer explores the system, scanning and browsing 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 is a domain-specific set of software components intended to facilitate construction of stand-alone systems on Intel x86 hardware. The OS-Kit 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 persistent state. To say that the Ext2 subset has no persistent state is not a contradiction. Instances of file systems maintain persistent data, and the Ext2 code subset of the OS-Kit is merely the blueprint for constructing and maintaining an instance of an Ext2 file system.

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 with 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 GNU profiler (gprof) program directly into 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 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 http://www.cs.utah.edu/projects/flux/oskit/.

Coyote. The purpose of Coyote is to support the construction of communication protocols such as atomic multicast, group remote procedure call (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. Other components are microprotocols, events, and a runtime 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 nonhierarchical construction of that layer using microprotocols. Examples of microprotocols include message ordering schemes or retransmission policies. Microprotocols are considered fine-grained modules that can be registered to handle events. Hence, in this system a component is a layer of a communication protocol stack and a microcomponent is a microprotocol 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 runtime system or by microprotocols. The runtime 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 microprotocols 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 microprotocols are in files and they are composed offline by the designer. The overall framework that supports this system is contained in read-only files. There are user-modifiable files that contain library-like functions and standard protocols and policies, and there can also be user supplied files with their own developed microprotocols. A key aspect of configurability in Coyote is the support for events and event handlers. It is the raising of an event that causes execution of a microprotocol. Reconfiguration is supported by the binding and unbinding of event handlers.

PURE: Portable, Universal Runtime Executive. The PURE system approach was 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: a program family and an object orientation. The program family concept is provides 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-orientation 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 of 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 (providing per thread synchronized operation of some critical functions), filing (providing the means to keep track of the allocated threads), and 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. CORE is made of four building units. These units can be composed in such a way as 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, or Nucleus Extension. While not explicit, the highly configurable and fine-granular structure provided in this system allows for better support of 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 purposes, PURE provides tools that let users specify their needs and requirements for the customized system. The approach uses 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* that produces the desired system. PURE is not available as source code.

2K. The 2K system is a reflective, component-based operating system whose main goal is to provide a generic framework 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 (metalevel data and methods on that data) offered at the object request broker (ORB) level to provide the means for adaptation.

A component in 2K is a dynamically loadable unit that is stored in a dynamic link library (DLL). The components can be loaded or unloaded depending on the user's needs and the garbage collection algorithms. After being integrated in the 2K environment, a PDA can select a category of components (such as spreadsheets) to interact with the components belonging to that category (Excel or Lotus, for example). Consequently, 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) 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 configuration and reconfiguration 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 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 by letting system components reason about their interactions with other components and make adaptation decisions. 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/.

Industrial systems

There are about 100 real-time, embedded operating systems on the market. Many of

them do not provide configuration capabilities or are not customizable. Others, such as QNX and VxWorks provide optional modules that can be statically or dynamically linked to the operating system.[22-23] However, these modules rely on a basic kernel and are not designed using a component-based approach. The modules are designed with no intention of being used in other environments. This survey of industrial systems includes JavaOS, Jbed, MMLite, Pebble, icWORKSHOP, and eCos. Some of these are or were 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 provide a component-based operating system approach for embedded applications. However, they use different approaches to provide interaction between components.

JavaOS, developed by Sun Microsystems and IBM, and Jbed, developed by Oberon Microsystems, are basically designed for Java technology.[24-25] On the other hand, MMLite, supported by Microsoft, uses a lightweight COM as its component infrastructure.[26] Other systems, including Pebble, IcWORKSHOP, and eCos are not based on any popular component infrastructures.[27-29] Pebble, supported by Lucent Technologies/Bell Laboratories, provides 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, application-specific operating systems (ASOSs) for application-specific standard processor (ASSP) hardware. The purpose of eCos, develop by Cygnus, is to provide an open, embedded-software infrastructure.

Java-OS. Java-OS is an operating system specifically developed for embedded systems and network computers. Actually, there are three Java-OSs available: for business, consumers, and network computers (see 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 operating system detects this fact, 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 or interprocess communication (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. While the developers of Java-OS make certain claims for Java-OS 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 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; other Java platforms that appear as candidates are EmbeddedJava and JavaCard. The basic differences between these Java-based platforms are their differences in support for threads, garbage collection, floating-point numbers, and so forth. In Jbed, the kernel is actually the Java Virtual Machine and is also called the runtime system or kernel level. The Jbed runtime system provides a thread scheduler, memory allocation, and garbage collection as a minimal system. This configuration requires up to 64 Kbytes of RAM. Other possible configurations include the minimal system with TCP/IP and a Web server (128 Kbytes required) and the minimal system with network loader and a flash compiler for translation of Java code into machine code on the target upon loading (256 Kbytes required). In terms of operating system kernel configuration, these are the options apparently available, meaning that operating system 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 a small memory footprint, real-time thread support, and deadline scheduling. However, it is not clear how Jbed 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 loaders, 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 supports a client/server model. At this layer, applications such as process control, remote diagnosis, and alarm systems are called clients. The clients use the components (embedded applications) through server programs (such as debugging, remote control, and Web servers) that 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 the operating system kernel level.

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

MMLite. MMLite is an object-based, modular system architecture that provides a menu of components for use at compile-time, link-time, or runtime to construct a wide range of applications. A component in MMLite consists of one or more objects. Multiple objects can 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 marshaling 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 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 provides 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 to 3,000 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 is 26 Kbytes on x86, and 20 Kbytes on ARM. It is not clear to what extent MMLite provides users with the ability to easily select components that the MMLite developers write, and to what extent users themselves 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 runtime performance of components due to namespace resolution and the creation and loading of proxy objects is lacking.

Pebble. Pebble is a new operating system designed to be an efficient application-specific operating system and to support component-based applications. It also supports complex embedded applications. As an operating system it adopts a microkernel architecture with a minimal privileged 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). Examples of system components are the interrupt dispatcher, scheduler, portal manager, device driver, file system, virtual memory, and so on. The Pebble kernel and the essential components (interrupt dispatcher, scheduler, portal manager, real-time clock, console driver, and idle task) need approximately 560 Kbytes 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 communication between PDs. 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 PDs provide the means to isolate the components, portals provide the means for components to communicate with each other. Instantiation and management of portals are performed 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 to fulfill applications requirements.

Based on the description of the system, there are no composition tools to provide the construction of the system. Also, there are no analysis tools to determine correctness or performance. It seems that what makes Pebble 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 (you can choose to include a file system or not, for instance) and static. The icWORKSHOP 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. Their components include 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. These components vary in functionality and size quite a bit, but they are much lower level than for Corba or even OSKit components. Users can tailor any of these predefined 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 maintaining low overhead to meet time requirements, handling interrupts quickly, performing a more global analysis of memory and time requirements, and controlling device drivers. The domain for their product is then any application and embedded system that might require a tailored real-time operating system.

Integrated Chipware provides three ready-to-run kernels as well as multiple options for various activities such as scheduling. A development tool called icBUILD is included in icWORKSHOP. 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 incorporate them 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). The eCos embedded operating system 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 lets developers extend the core components and develop new or modified components.

The building unit in this system varies from a package (a coarse grain component) to a configuration option (a 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, µITRON compatibility layer, hardware abstraction layer, C library, watchdog device, and I/O subsystem.

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 µITRON 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 µITRON package. On a lower level, options regarding the components can be configured—termed *microconfiguration*. For example, the modification of the kernel to support 16 versus 32 priority levels would be considered a microconfiguration. 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 lets the developer 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 runtime.

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 has not yet released it to the public. Although eCos is a commercial system, its source code can be downloaded from http://www.cygnus.com/ecos/.

Summary of academic and industrial systems

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 often are 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.

At the second level, component-based configuration of operating system functionality, such as that supported by OSKit, is used. Here, components deal with typical operating system functionality, but do not really attempt to support embedded systems. Finally, at the third level, fine-grained components that include operating system-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, and so forth. 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 them use components with different granularities. Analysis tools for 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 and operating system components. Among the systems that provide reconfiguration capa-

Table 1. Summary of the systems.

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 user components
JavaOS	None	Not available	Java Virtual Machine	Dynamic detection of drivers
Jbed	None	Jbed IDE	Java Virtual Machine	Downloading of user 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

bilities, a few provide it for operating system components. Table 1 provides a summary of the major issues being investigated in using component-based software for operating system and embedded applications.

Opportunities for research

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 in three main areas:

- software components themselves,
- dynamically reconfigurable hardware components, and
- the configuration process.

Software component research issues include:

- developing lightweight interfaces,
- defining metrics and developing techniques for categorizing components along memory size, execution time, fault tolerance, security, and quality-of-service dimensions (nonfunctional attributes, for example),
- developing and saving configuration information about components, (for instance, constraints such as component A must be used with component B and C and not with D, or only works on certain hardware).

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 operating system components and on the application code for both the functional and nonfunctional attributes?
- How do you 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 do you guide the developer to choose the right components to meet all the requirements of space, time, cost, power, and speed to market?
- Recognizing that analysis tools are critical, how do you analyze the resultant collection of components for correctness, for meeting deadlines, and meeting other requirements? MICRO

References

1. H. Takada, "μITRON: A Standard Real-Time Kernel Specification for Small-Scale

Embedded Systems," *Real-Time Magazine*, 1997, q3.
2. G. Booch, *Software Components with Ada: Structures, Tools and Subsystems*, Benjamin-Cummings, Redwood City, Calif., 1987.
3. O. Nierstrasz et al., "Component-Oriented Software Development," *Comm. the ACM*, vol. 35, no. 9, 1992, pp. 160-165.
4. R. Orfali et al., *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, New York, 1996.
5. J. Samentiger, *Software Engineering with Reusable Components*, Springer-Verlag, Berlin, 1997.
6. K. Short, *Component-Based Development and Object Modeling*, Sterling Software; http://www.cool.sterling.com.
7. C. Szyperski, *Component Software Beyond Object-Oriented Programming*, Addison-Wesley, ACM Press, New York, 1998.
8. A. Beugnard et al., "Making Components Contract Aware," *IEEE Computer*, vol. 32, no. 7, 1999, pp. 38-45.
9. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, revision 2.0, 1997; http://www.omg.org.
10. Microsoft Corporation and Digital Equipment Corporation, *The Component Object Model Specification*, Redmond, Wash., 1995.
11. Microsoft Corporation, *Distributed Component Object Model Protocol*, version 1.0, Redmond, Wash., 1998.
12. Sun Microsystems, JavaBeans, version 1.0, 1996; http://java.sun.com/beans.
13. K. Arnold et al., *The Jini Specification*, Addison-Wesley, Reading, Mass., 1999
14. B. Meyer and C. Mingins, "Component-Based Development: From Buzz to Spark," *IEEE Computer*, vol. 32, no. 7, 1999, pp. 35-37.
15. D. Engler et al., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. 15th Symp. Operating Systems Principles* (SOSP), ACM Press, New York, 1995, pp. 251-266.
16. B. Bershad et al., *SPIN: An Extensible Microkernel for Application-Specific Operating System Services*, tech. report, Univ. of Washington, 1994.
17. R. Campbell et al., "Designing and Implementing Choices: An Object-Oriented System in C++," *Comm. the ACM*, vol. 36, no. 9, Sept. 1993, pg. 117-126.
18. B. Ford et al., "The Flux OSKit: A Substrate for Kernel and Language Research," *Proc. 16th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1997, pp. 38-51.
19. N. Bhatti et al., "Coyote: A System for Constructing Fine-Grain Configurable Communication Services," *ACM Trans. Computer Systems*, vol. 16, no. 4, 1998, pp. 321-366.
20. D. Beuche et al., "The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems," *Proc. 2nd IEEE Int'l Symp. Object-Oriented Real-Time Distributed Computing*, IEEE Press, Piscataway, N.J., 1999.
21. F. Kon et al., "2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments," *Lecture Notes in Computer Science Series: Object-Oriented Technology ECOOP* [European Conference on Object-Oriented Programming] *Workshop Reader*, S. Demeyer and J. Bosch, eds., Springer-Verlag, Heidelberg, vol. 1543, 1998.
22. D. Hildebrand, "An Architectural Overview of QNX," *Proc. USENIX Workshop on Microkernels and Other Kernel Architectures*, USENIX: The Advanced Computing Systems Assoc., 1992, pp. 113-126; http://www.usenix.org/.
23. Wind River Systems, *VxWorks Programmer's Guide*, Alameda, Calif., 1995.
24. T. Saulpaugh and C. Mirho, *Inside the JavaOS Operating System*, Addison Wesley, Reading, Mass., 1999.
25. Oberon Microsystems, *Jbed Whitepaper: Component Software and Real-time Computing*, tech. report, 1998; http://www.oberon.ch.
26. J. Helander and A. Forin, "MMLite: A Highly Componentized System Architecture," *Proc. 8th ACM Special Interest Group on Operating Systems European Workshop* (SIGOPS), ACM Press, New York, 1998, pp. 96-103.
27. E. Gabber et al., "The Pebble Component-Based Operating System," *Proc. USENIX Annual Technical Conference*, USENIX Assoc., 1999, pp. 267-282; http://www.usenix.org/
28. Chipware, Integrated Chipware IcWorkShop, 1999; http://www.chipware.com/.
29. Cygnus, *eCos: Embedded Cygnus Operating System*, tech. white paper, 1999; http://www.cygnus.com/ecos.

L. Fernando Friedrich is a faculty member at the department of computer science, Federal University of Santa Catarina, Brazil. His research interests are in operating systems, real-time computing, and parallel and distributed computing. Friedrich received his PhD in engineering from the Federal University of Santa Catarina. He is a member of the Brazilian Computing Society (SBC) and the Society for Computer Simulation International.

John Stankovic is the BP America Professor and chair of the Computer Science Department at the University of Virginia. His research interests are in real-time and embedded systems focusing on operating systems and scheduling. Stankovic received his PhD from Brown University. He is a fellow of the IEEE and the ACM and serves on the board of directors for the Computer Research Association.

Marty Humphrey is a research assistant professor in the Computer Science Department at the University of Virginia where he also codirects the Legion project (for middleware that creates a cluster of computers to work on a single problem). His technical interests include operating systems (embedded, real time, wide area), real-time systems, wide area and distributed systems, and computational grids. Humphrey received his PhD from the University of Massachusetts. He is a member of the IEEE and the ACM.

Michael Marley is an embedded systems programmer with Lucent Technologies. His research interests include embedded and real-time systems. He received the BSEE and MSEE from Southern Methodist University and the MCS from the University of Virginia. He is a member of the IEEE.

John Haskins Jr. is pursuing his doctorate at the University of Virginia Department of Computer Science where he earned his MCS in 2000. He earned his BS in computer science from Georgia Tech in 1997. Between the conclusion of his undergraduate studies and the beginning of graduate studies, he worked as an adjunct research staff at the IDA Center for Computing Sciences. His technical interests include computer architecture and operating systems.

Direct questions and comments about this article to L. Fernando Friedrich, Federal University of Santa Catarina, Department of Computer Science, Florianópolis, SC, Brazil, 88070-900; fernando@inf.ufsc.br.