VEST
A Toolset For Constructing and Analyzing Component Based Embedded Systems[1]

John A. Stankovic
Department of Computer Science
University of Virginia
Charlottesville, VA 22906
stankovic@cs.virginia.edu

Abstract. Embedded systems are proliferating at an amazing rate. To be successful these systems must be tailored to meet many cost, time to market, performance, and dependability constraints. Building and tailoring embedded systems by using domain specific components has promise. However, in composing components the most difficult issues are ensuring that non-functional properties such as real-time performance and dependability are being met. The VEST toolkit is aimed at providing a rich set of dependency checks based on aspects to support embedded system development via components.

1 Introduction

Embedded systems are proliferating at an amazing rate with no end in sight. In 1998 only 2% of all processors where used for general purpose workstations and 98% for embedded systems. The percentage of processors used for workstations is rapidly approaching 0% for the year 2001. Success of embedded systems depends on low cost, a high degree of tailorability, quickness to market, cost-effective variations in the product, and sometimes flexible operation of the product. The reliability of these products and the degree of configurability are paramount concerns. Currently, there is no efficient way to build software for these systems. The use of component based software for constructing and tailoring these systems has promise. However, most components are too heavyweight and don't explicitly address real-time, memory, power and cost constraints. What is required is a new type of component that is lightweight and relates to the physical and real-time properties of embedded systems.

The first part of the solution is developing the new components themselves. This problem is where most people have spent their energies. While this is a necessary step, it is the easiest step and it ignores fully addressing how the components interact with other components or how they fit into a component infrastructure. The second problem is that while significant work has been done in developing CORBA [36], DCOM [24], Jini [1] components, much less has

[1] This work was supported, in part, by the DARPA PCES program under grant F33615-00-C-3048.

been done with components suitable for embedded systems. Third, most tools available for the configuration process provide little more than linking or an extended "make" capability. It is the designer who has to know everything about the components and their (often hidden) constraints. For example, a designer may know that the combination of 2 seemingly unrelated components leads to unpredictable delays in network protocol processing. What is needed are tools that support the specification of embedded system requirements followed by knowledgeable and helpful construction of the embedded system with careful analysis of component dependencies as well as the time, memory, power, and cost constraints. The final product must offer as many guarantees as possible along many dimensions including correctness, performance and reliability. Our work is focusing on the development of effective composition mechanisms, and the associated dependency and non-functional analyses for real-time embedded systems. Our solutions are embodied within a toolkit called VEST (Virginia Embedded Systems Toolkit).

We are attempting to develop techniques that are general enough to construct and perform analysis checks on any of the following types of systems:

- a static and fixed embedded system at low cost (e.g., software for a wrist watch),
- networked and long lived embedded systems that can be made more evolvable by hot swapping both hardware and software (e.g., air traffic control), and
- a reflective runtime environment to support a high degree of adaptability (e.g., air traffic control or even a set top box that sometimes acts to download movies, at other times act as a PC, or to run games).

The main purposes of this paper are threefold: to present the design of VEST, to highlight various types of dependency checks including aspects, and to illustrate the special issues involved with composing embedded real-time systems.

2 The Design of VEST

VEST (Virginia Embedded Systems Toolkit) is an integrated environment for constructing and analyzing component based embedded and real-time systems. Generally speaking, the VEST environment is designed to help people select or create passive components (collections of code fragments, functions, classes, HW, etc), compose them into a system/product, map them onto runtime (active) structures such as processes or threads, map them onto specific hardware, and perform dependency checks and non-functional analyses to offer as many guarantees as possible along many dimensions including real-time performance, and reliability[2]. The VEST environment is comprised of the following main parts:

- DOME (Domain Modeling Environment) [17], a tool-set which is an extensible collection of integrated model editing and analysis tools supporting a Model-Based Development approach to system/software engineering.

[2] VEST does not support formal proof of correctness.

- A library system composed of 5 libraries: domain application components, middleware components, OS components, aspects, and the product under development library.
- An interactive composition and analysis tool which supports:
  - Tailored infrastructure and product creation via domain specific software/hardware composition
  - Mapping of passive software components to active runtime structures (processes/threads)
  - Mapping of processes/threads to hardware platforms
  - A set of dependency and aspect checks
  - Ability to invoke off-the shelf or home grown analysis tools such as real-time and reliability analyses.

Using VEST the designer can

- Create a new component which entails not only adding the source code, but also adding a significant amount of reflective information. The reflective information for source code objects include categories such as interface information, linking information, location of source code itself, graphical information for displaying/representing the source object on the VEST graphical editor canvas, and other reflective information of the source object needed to analyze cross cutting dependencies. Examples of this latter category are presented in section 3.
- Choose a component from one of the libraries.
- Graphically compose components so that a hierarchy of components can exist.
- Use Glue Code to connect components, if necessary.
- Map components to processes so that the active part of a composed system can be designed and analyzed.
- Bind processes to HW. Only after this step can we truly do the real-time analysis since execution times are highly platform dependent.
- Perform dependency checks including aspects. This is one area where we believe VEST makes a major contribution. Previous systems did not have enough support for cross-cutting dependencies and this is one specific goal for VEST.
- Invoke off-the-shelf analysis tools to analyze a configured system.

To better support real-time and embedded systems, the VEST component library contains both software and hardware objects. The software objects can themselves be further divided into two subgroups: source objects that describe a specific code written in a traditional language like C, and higher-level objects that describe groupings of source objects. The hardware objects can also be divided into descriptions of a specific piece of hardware and descriptions of groupings of hardware components. We are supporting the following hardware devices: processors, memories, caches, DSP, busses, A/D and D/A, actuators and sensors. Sets of reflective information exist for each of these device types.

Processes and threads are separate entities in VEST and also have reflective information describing their attributes.

To briefly provide some idea of the graphical interface the tool buttons available to a designer include invoking/using/composing: Functions, Classes, Components, Threads, Processes, Devices, Composed Devices, Glue Code, Select and Move, and Dependency Connectors are provided for users to describe entities' dependency graphically (we provide both a Precedence Connector and a Call-Graph Connector). Attributes can be added to each connector, e.g., attributes can be added to a call-graph connector such as 1) Call instance: Which call instance this connector stands for; 2) Return type: Return parameter type; and 3) Input parameter list. A Mapping/Binding connector is used to map a passive component to a thread/process or bind a thread/process to a piece of hardware.

From the VEST graphical interface users can create a new library, add/delete components to a library, browse for a component, and search for a component by name or by keyword. Finally, through a menu item they can invoke the off-line analysis tools.

3 Dependency Checks and Aspects

One goal of VEST is to provide significant support for various types of dependency checking. Dependency checks are invoked to establish certain properties of the composed system. This is a critical part of embedded and real-time system design and implementation. For purposes of doing research, we have separated dependency checks into 4 types:

- Factual
- Inter-component
- Aspect
- General

Factual The simplest, component-by-component dependency check is called factual dependency checking. Each component has factual (reflective) information about itself including:

- WCET (worst case execution time)
- memory needs
- data requirements
- interface assumptions
- importance
- deadline or other time constraints
- jitter requirements
- power requirements (if a hardware component),
- initialization requirements
- environment requirements such as
    - must be able to disable interrupts
    - requires virtual memory

    - cannot block
- the code itself

The above list is extensible depending on the application area. A factual dependency check can include using this information across all the components in a configuration. For example, a dependency check might involve determining if there is enough memory. This can be very easy; add up the memory requirements of each component and compare to the memory provided by the hardware memory component. If memory needs are variable, a slightly more sophisticated check is needed that accounts for worst case memory needs. The greater number of factual dependency requirements that are checked by VEST the more likely it is that simple mistakes are avoided.

Inter-component Inter-component dependencies refer to pairwise component checks. This includes:

- call graphs
- interface requirements including parameters and types of those parameters
- precedence requirements
- exclusion requirements
- version compatibility
- software to hardware requirements, and
- this component is included in another.

The above list is extensible. Given a set of components configured for some purpose the inter-component checks also tend to be easy. For example, that modules conform to interface definitions can easily checked. That correct versions are being used together and that no two components that must exclude each other are included, also can be easily checked. The greater the number of inter-component dependency requirements that are checked, the more likely that simple mistakes are avoided.

Aspects Aspects [19] are defined as those issues which cannot be cleanly encapsulated in a generalized procedure. They usually include issues which affect the performance or semantics of components. This includes many real-time, concurrency, synchronization, and reliability issues. For example, changing one component may affect the end-to-end deadlines of many components that are working together. Many open questions exist here. How can we specify the aspects? Which aspects are specific for a given application or application domain and which are general? How does the tool perform the checks? Are the checks feasible for certain problems?

It is our belief that aspects include an open ended (unbounded) set of issues. Therefore, we cannot hope to be complete, rather we need to identify key aspects for embedded systems and create the specification and tools to address as many of these issues as possible. The more aspects that can be checked, the more confidence in the resulting composed system we will have. However, by no means do we claim that the system is *correct*, only that certain specific checked errors are not present.

Consider an example of aspects relating to concurrency. In looking into this issue, we have divided the problem into 4 parts: what are the concurrency concepts that need to be expressed, what mechanisms exist for supporting those concepts (the components), how can we express the concepts and components, and how are the dependency checks themselves performed?

Concurrency concepts include: synchronous versus asynchronous operation, mutual exclusion, serialization, parallelism, deadlock, atomicity, state (running, ready, blocked), events (triggers), broadcasts, coordinated termination, optimistic concurrency control, and multiple readers - single writer. This list can be extended, perhaps indefinitely.

Concurrency mechanisms include: locks, binary and counting semaphores, wait-list management, timers, messages, monitors, spin locks, fork-join, guards, conditional critical regions, and precedence graphs. At this time we have not developed any language for expressing the concurrency concepts or mechanisms.

Related to aspects, dependency checks seem to come in two types. One is an explicit check across components in the current configuration. Examples include: suppose a given system has only periodic tasks and a change is made to add aperiodic tasks. A particular dependency check may be able to identify all those components that had previously assumed that no aperiodics would exist. Another example involves a system that has no kill primitive. Adding a kill primitive imposes a key requirement on all other tasks that they cannot be killed while they are in the middle of a critical section.

The second type of aspect checking is actually deferred to associated analysis tools. For example, checking that real-time constraints are met is best done by collecting the right information and passing it to a more general analysis tool. Four such tools are described in section 4.

General The fourth type of dependency check we refer to as a *general* dependency check. It is not clear that this category is needed and it may eventually be folded into the other three. However, global properties such as that the system should not experience deadlock, or that hierarchical locking rules must be followed, or a degree of protection is required seem different enough that, for now, we are treating these checks as a separate category. Note that it is not important what category a certain dependency check falls into, only that such a check is made. However, each category may require fundamentally different solutions. Aspects, while they can be global, seem to focus more on how code changes affect other modules, while the general dependency category focuses on global requirements. For example, a dependency check for the absence of deadlock might rely on a Petri-net analysis (see also non-functional analyses section below).

3.1 Examples of Dependency Checks

The current version of VEST implements the following dependency checks.

A Memory Size Check: Browse through a system (all the graphical icons on the editing pane or reachable from that pane), adding up the memory requirements. Browse through the system, adding up the total memory provided by the hardware memory modules. Compare the memory required with the available memory and inform users of the result. Note: Besides modules whose memory requirement values are available, the number of modules whose memory requirement values are not available is also counted and reported to users.

Power Consumption Check: Browse through a system (all the graphical icons on the editing pane or reachable from the pane), adding up the power requirements and the available power, and summarizing the number of modules whose power requirement value are not available. Display the power consumption result.

Interface Compatibility Check: For each call graph connector in the system, check whether its interface is compatible with its calling component's interface. For those mismatched call graph connectors, mark them in red and inform the users.

All Call-Graphs Check: Browse through a system to check all of its components' calling dependencies. For a components calling dependency list, look through the whole system to see if components it calls are missing from current system. If some components are missing from current system, inform users.

Exclusion Constraints Check: This name-based checking will check any generic exclusion violations among the system. Browse through a system to check all of its components' exclusion constraints. For a component's exclusion list, look through the whole system to see if there is an exclusion violation, i.e., other components that it excludes also compose current system. If an exclusion happens in current system, inform users.

No Dynamic MM Check: This checks a specific exclusion constraint, i.e., a dynamic memory management constraint. Users can use it to check if dynamic memory management is excluded by a system (i.e., some components in the system exclude the use of any dynamic memory management components).

Buffer Size: Browse through a system, for each buffer, add up its total buffer consumption based on its rate of execution. Browse through the system, for each buffer, add up its total buffer production based on its rate of execution. Compare each buffer's total production with its total consumption. If its total production is larger than total consumption, messages might be lost. Inform users.

4 Non-functional Analysis Tools

Our configuration tool needs to scan the newly composed system, extracting the appropriate reflective information needed for a particular analysis. It then passes that information to the analysis tool in the correct format. A tool may be an off-the-shelf tool or one supplied with our system. To make this clear, consider the following four examples.

Consider that we want to perform a real-time analysis that takes exclusive access to shared resources into account. See Figure 1. To perform such an analysis requires making many decisions such as we must know all the components (hardware and software), the mapping of software to runtime active structures, the mapping of software to hardware, the scheduling algorithm being used, and the workload. The component reflective information must include worst case execution times for that software on specific hardware as well as what data that software needs to share in exclusive mode. All of this information can be part of the configuration tool set. Given knowledge of all these things it is then possible to use the Spring scheduling off-line algorithm [45], designated as H() in the figure, to compute a feasible schedule where all deadlines and exclusion constraints are met, or be told that no such schedule was found. See Figure 2. In figure 2 it is shown that task A is scheduled on one CPU, while tasks B and C are on a second CPU. Resources R1 through R5 are additional resources required by these tasks and they are allocated for time periods that *avoid* conflicts. For example, tasks A and B are scheduled to run concurrently and they use a disjoint set of resources R2 and R3, respectively. Alternatively, tasks B and C are scheduled to run sequentially because they conflict over resource R3. In this schedule all the tasks make their deadlines, designated D(i) i=A,B,C in the picture. In general, if no valid schedule is found the designer can take various actions such as adding hardware and then reiterate the analysis.

A second example would be to perform actions very similar to the above scenario except the designer makes assumptions about using the rate monotonic analysis approach with priority ceiling and invokes the schedulability analysis for that paradigm. This, for example, is being done in MetaH.

A third example involves reliability analysis. Off-the shelf tools exist for computing the reliability of a given system configuration. Such an analysis is usually quite restrictive. Yet, what ever level of sophistication that the off-the-shelf tool supports, that can be supported here. For example, if we assume fail stop hardware, we know the probability of failure of each hardware component and the HW configuration, we can input this to a tool such as Galileo and determine the reliability. This, of course, does not consider software faults.

Tools based on Petri-nets analysis exist to evaluate various concurrency properties of a system. Again, our configuration tool can extract the right information from the reflective information about the newly composed system, put it into the format expected from a Petri-net analysis tool and obtain the analysis benefits of that tool.

5 Current Status

VEST version 0 has been implemented. Version 0 has basic library support, support for functions as components, various examples of dependency checking, and the capability to invoke external off-the-shelf tools. It has basic graphic support and is built on DOME. VEST version 1 is currently being implemented. It will contain the following:

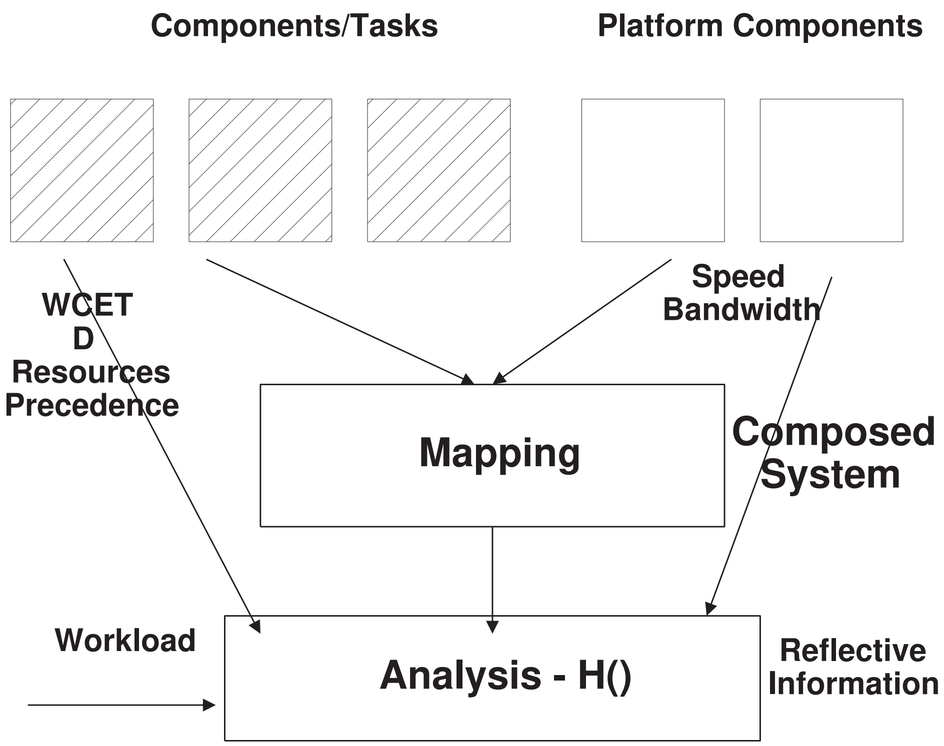

Fig. 1. Real-Time and Concurrency Example.

- Support for objects and classes including multiple interfaces.
- Support for configurable components so that a designer can instantiate a library component with specific parameters for their product.
- Support for events as first class entities and how those entities relate to aspects.
- Direct support for language independent aspects.
- Additional library support for the management of 5 libraries: base application code library, middleware library, OS library, aspect library, and new product library.
- Addition of more dependency and aspect checking.

6 State of the Art

The software engineering field has worked on component based solutions for a long time. Systems such as CORBA [36], COM [23], DCOM [24], and Jini [1] exist to facilitate object or component composition. These systems have many advantages including reusability of software and higher reliability since the components are written by domain experts. However, CORBA and COM tend to

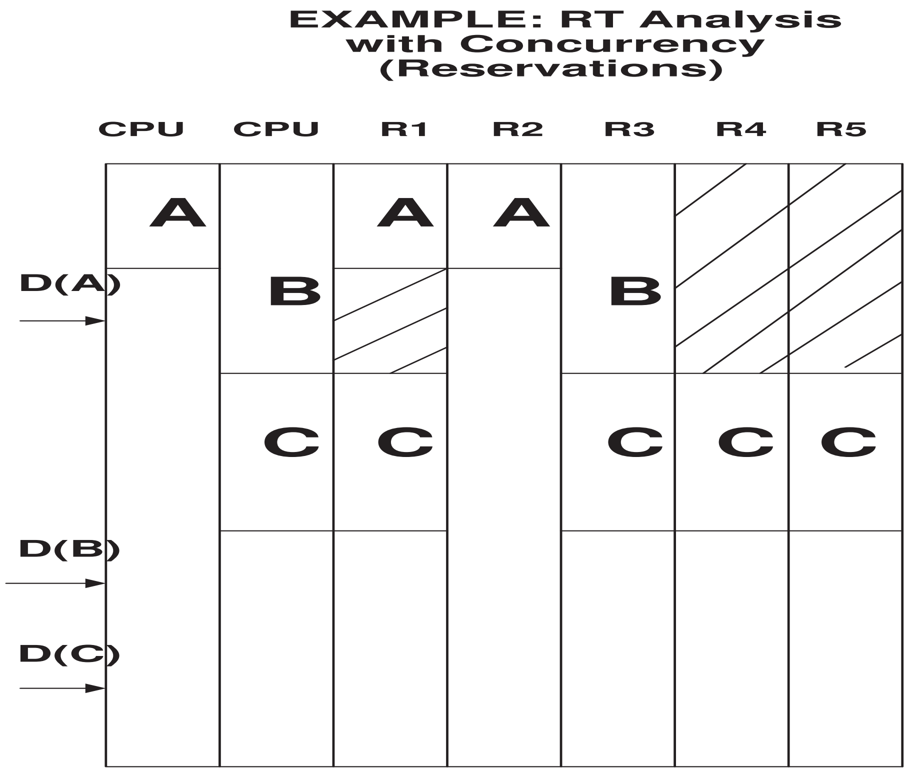

Fig. 2. Feasible Schedule.

produce heavyweight components that are not well suited to embedded systems, and none of these systems have adequate analysis capabilities. Also, the generality and large size of component libraries makes these general solutions difficult to use. However, using components in more focussed domains such as found in many embedded system applications seems promising. Often a high degree of tailorability is needed to make a product successful in the marketplace. This tailorability may be required in the application code, the middleware (if any) and the OS.

Tailorability has been the focus of OS microkernel research such as in Spin [2], Vino [9], and the exo-kernel [10]. However, these systems have not addressed embedded systems issues, are not component based and have no analysis of the type being proposed here. They also are applied to general purpose timesharing systems with totally unrelated applications executing on a general purpose OS.

Real-time kernels such as VxWorks [44], VRTX32 [31], and QNX [16] all have a degree of configurability. But the configuration choices are at a high level, e.g., you can include virtual memory or network protocols or not. There are no dependency checks or analysis tools to support composition.

The KNIT [32] system is an interesting composition system for general purpose operating systems. This system is addressing a number of cross cutting concerns in composing operating systems. For example, they consider linking, initialization, and a few other dependencies. To date, it has not focussed on real-time and embedded system concerns.

For embedded systems we do find some work on component based OSs. These include: MMLite [15], Pebble [13], Pure [3], eCOS [8], icWorkshop [18] and 2K [20]. However, these might be considered first generation systems and have focussed on building the components themselves with a substantial and fixed infrastructure. These systems offer very little or no analysis tools and, for the most part, minimal configuration support. For more details on these systems see [12].

Using components in focused domains has been successful. For example, there are tools and components for building avionics systems. There are also systems for creating network protocols such as Coyote [5], the click modular router [25], and Ensemble/Nuprl [21]. The success of these systems lies in the ability to focus on a specific area. This permits better control over problems such as understanding dependencies and avoiding the explosion of the numbers and variants of components.

To date, the closest system to match our goals is the MetaH [43] system. This system consists of a collection of tools for the layout of the architecture of an embedded system and its reliability and real-time analysis. The main differences from our work include MetaH begins with active tasks as the components, assumes an underlying real-time OS, and has limited dependency checking. In contrast we propose to first create passive components (collections of code fragments, functions and objects) and then map those onto runtime structures, and we also focus on adding key dependency checks to address cross cutting dependencies among composed code.

7 Summary

The number of embedded systems and processors are growing far faster than workstations. Many of these systems are composed of networks of processors, sensors, and actuators. Software for these systems must be produced quickly and reliably. The software must be tailored for a particular application/product. A collection of components must interoperate, satisfy various dependencies, and meet non-functional requirements. These systems cannot all be built from scratch. A goal of our toolkit is to substantially improve the development, implementation and evaluation of these systems. The toolkit focuses on using language independent notions of *aspects* to deal with non-functional properties, and is geared to embedded system issues that include application domain specific code, middleware (if any), the OS, and the hardware platform. Our toolkit could have beneficial impact on products from smart toasters, to set top boxes, Internet appliances, defense systems, and avionics.

References

1. Arnold K., O'Sullivan B., Scheifler R.W., Waldo J., and Wollrath A.(1999) The Jini Specification. Addison-Wesley.
2. 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.
3. 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.
4. Beugnard A., Jezequel J., Plouzeau N. and, Watkins D. (1999) Making Components Contract Aware. Computer, 32(7), 38-45.
5. 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.
6. Booch G. (1987) Software Components with Ada: Structures, Tools and Subsystems. Benjamin-Cummings, Redwood City, CA.
7. 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.
8. Cygnus (1999) eCos - Embedded Cygnus Operating System. Technical White Paper (http://www.cygnus.com/ecos).
9. Endo Y., et. al. (1994) VINO: The 1994 Fall Harvest, TR-34-84, Harvard University.
10. 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.
11. 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.
12. Friedrich, L., J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. (2001). A Survey of Configurable, Component-Based Operating Systems for Embedded Applications, *IEEE Micro*, Vol. 21, No. 3, May-June, pp. 54-68.
13. 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.
14. Haskins J., Stankovic J., (2000), muOS, TR, University of Virginia, in preparation.
15. Helander J. and Forin A.(1998) MMLite: A Highly Componentized System Architecture. Proceedings of the Eighth ACM SIGOPS European Workshop. Sintra, Portugal.
16. Hildebrand D. (1992) An Architectural Overview of QNX. Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures. Seattle, WA.
17. Honeywell DOME User Guide, http://www.htc.honeywell.com/dome/support.htm.documentation.
18. Integrated Chipware IcWorkShop (http://www.chipware.com/).
19. Lopes, C., and Kiczales G. (1997), D: A Language Framework for Distributed Programming, TR SPL97-010, Xerox Parc.

20. 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.
21. Liu X., et. al. (1999), Building Reliable High-Performance Communication Systems from Components, *SOSP*, Vol. 33, No. 5.
22. Meyer B. and Mingins C. (1999) Component-Based Development: From Buzz to Spark. Computer, 32(7), 35-37.
23. Microsoft Corporation and Digital Equipment Corporation (1995) The Component Object Model Specification. Redmond, Washington.
24. Microsoft Corporation (1998) Distributed Component Object Model Protocol, version 1.0. Redmond, Washington.
25. Morris R., Kohler E., Jannotti J., and Kaashoek M. (1999), The Click Modular Router, *SOSP*, Vol. 33, No. 5.
26. Niehaus D., Stankovic J., and Ramamritham K. (1995), A Real-Time Systems Description Language, *IEEE Real-Time Technology and Applications Symposium*, pp. 104-115.
27. Nierstrasz O., Gibbs S., and Tsichritzis D. (1992) Component-oriented software development. Communications of the ACM, 35(9), 160-165.
28. Oberon Microsystems (1998) Jbed Whitepaper: Component Software and Real-time Computing. Technical Report. Zurich, Switzerland (http://www.oberon.ch).
29. Object Management Group (1997) The Common Object Request Broker: Architecture and Specification, Revision 2.0, formal document 97-02-25 (http://www.omg.org).
30. Orfali R., Harkey D., and Edwards J. (1996) The Essential Distributed Objects Survival Guide. John Wiley and Sons, New York.
31. Ready J. (1986), VRTX: A Real-Time Operating System for Embedded Microprocessor Applications, *IEEE Micro*, Vol. 6, No. 4, pp. 8-17.
32. Reid, A., M. Flatt, L. Stoller, J. Lepreau, and E. Eide. (2000) Knit: Component Composition for Systems Software. *OSDI 2000*, San Diego, Calif., pp. 347-360.
33. Samentiger J. (1997) Software Engineering with Reusable Components. Springer-Verlag, Town.
34. Saulpaugh T. and Mirho C. (1999) Inside the JavaOS Operating System. Addison Wesley, Reading, Massachusetts.
35. Short K. (1997) Component Based Development and Object Modeling. Sterling Software (http://www.cool.sterling.com).
36. Siegel J. (1998), OMG Overview: Corba abd OMA in Enterprise Computing, *CACM*, Vol. 41, No. 10.
37. Stankovic J., and Ramamritham K. (1991), The Spring Kernel: A New Paradigm for Real-Time Systems, *IEEE Software*, Vol. 8, No. 3, pp. 62-72; also in text on *Readings in Real-Time Systems*, IEEE Press, editor Mani Krishna.
38. Stankovic J. , Ramamritham K., Niehaus D., Humphrey M., and Wallace G. (1999), The Spring System: Integrated Support for Complex Real-Time Systems, special issue of *Real-Time Systems Journal*, Vol. 16, No. 2/3.
39. Stankovic J. and Ramamritham K. (1995), A Reflective Architecture for Real-Time Operating Systems, chapter in *Advances in Real-Time Systems*, Prentice Hall, pp. 23-38.
40. Sun Microsystems (1996) JavaBeans, version 1.0. (http://java.sun.com/beans).
41. Szyperski C. (1998) Component Software Beyond Object-Oriented Programming. Addison-Wesley, ACM Press, New York.

42. Takada H. (1997) ITRON: A Standard Real-Time Kernel Specification for Small-Scale Embedded Systems. Real-Time Magazine, issue 97q3.
43. Vestal, S., (1997), MetaH Support for Real-Time Multi-Processor Avionics, *Real-Time Systems Symposium.*
44. Wind River Systems, Inc. (1995) VxWorks Programmer's Guide.
45. Zhao, W., Ramamritham, K., and Stankovic, J., (1987) Preemptive Scheduling Under Time and Resource Constraints, **Special Issue** of *IEEE Transactions on Computers* on Real-Time Systems, Vol. C-36, No. 8, pp. 949-960.