VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems For Embedded and Real-Time Systems

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

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 2000. 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 will become 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 [33], DCOM [22], Jini [1] components, much less has 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, configuration, and reconfiguration mechanisms, and the associated dependency and non-functional analyses for real-time embedded systems. To start we are focusing on the construction of the OS-like portion of embedded systems.

Our approach enhances the process of building reliable software for the embedded systems revolution that is underway. We intend to apply our components, and configuration and analysis tools to several products such as set top boxes and multimedia mobile phones as proofs of concept. Our goal is to develop techniques that are general enough to construct 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 as a PC, or to run games[1]).

2 State of the Art

Basically, our work is focussed on component based solutions for embedded systems. Software engineering has worked on components for a long time. Systems such as CORBA [33], COM [21], DCOM [22], and Jini [1] exist. These systems have many advantages including reusability and higher reliability since the components are written by domain experts. However, CORBA and COM tend to produce heavyweight components that are not suited to embedded systems, and none of these systems have adequate analysis capabilities.

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 [41], VRTX32 [29], and QNX [15] 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.

For embedded systems we do find some work on component based OSs. These include: MMLITE [14], Pebble [12], Pure [3], eCOS [8], icWorkshop [16] and 2K [18]. 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.

[1]Our work concentrates on system-level adaptability rather than application level, but in many embedded systems the distinction is small. In other cases, like the set top box, changes at the application level might also require associated changes at the system-level.

Focussed domains have also produced component based solutions. 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 [23], and Ensemble/Nuprl [19]. 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. We believe that OSs for embedded systems is also a well defined focussed area and, consequently, a component based solution for this area is viable.

Perhaps the closest system to match our goals is the MetaH [40] 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.

3 System Architecture

To develop ASOSs for embedded systems we have developed a a toolset called VEST (Virginia Embedded Systems Toolset) that embodies a composition and analysis architecture. The architecture uses the concept of reflection as a unifying theme both for off-line analysis and, when appropriate, for on-line adaptability.

The architecture of VEST (see Figure 1) consists of the following:

- a component, subcomponent and microcomponent library
- a configuration tool that supports
  - infrastructure creation
  - embedded system composition
  - mapping of passive software components to active runtime structures (tasks/threads)
  - dependency checks including aspects and dependencies on hardware
- analysis tools
  - real-time analysis
  - reliability analysis

The library contains components, subcomponents, micro-components and previously created and saved infrastructures. The library also contains dependency information that includes aspects and specific dependencies on hardware and other physical system issues. All of this information is considered reflective information to be used by the configuration and

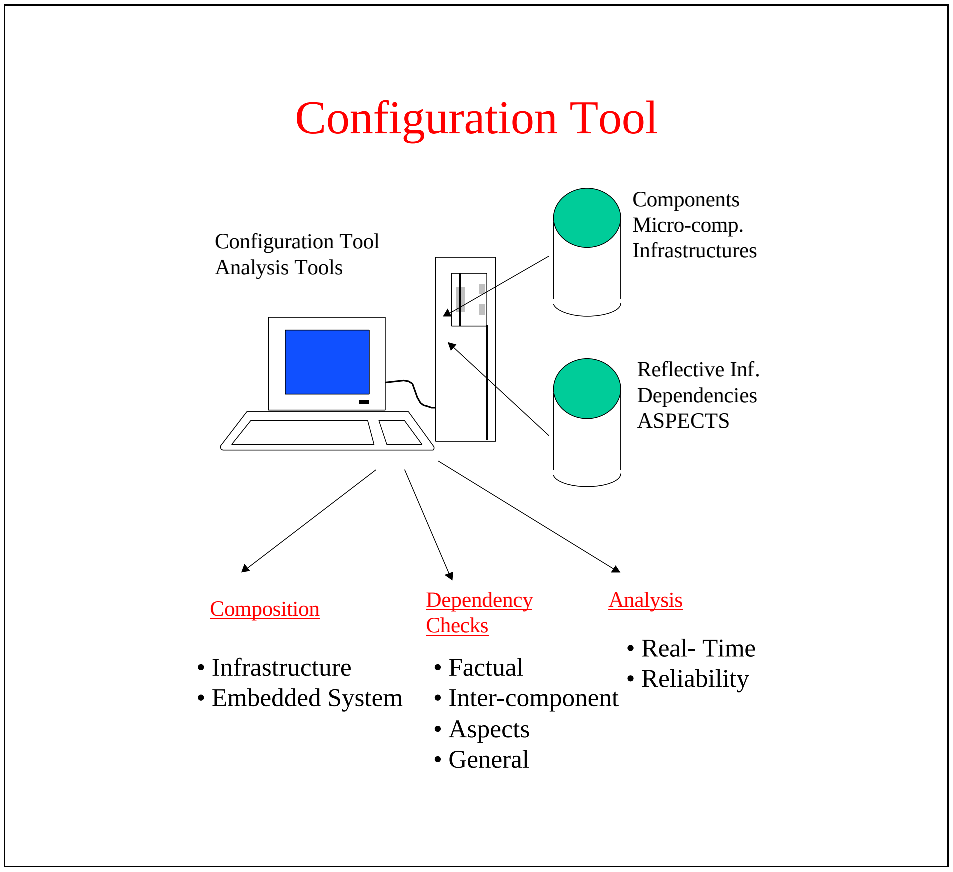

Figure 1: Configuration Tool.

analysis tools. The configuration tool enables adding or deleting components and reflective information, listing components, and browsing through the library. The infrastructure creation feature enables composing micro-components into an infrastructure. The user could also just choose a previously defined infrastructure if an appropriate one exists. The configuration tool then permits users to create the embedded system by composing OS-like components and application components into a system. This includes mapping passive components into runtime structures (tasks/threads).

Dependency checks are then invoked to establish certain properties of the composed system. This is a critical part of the system and more details are given below. For purposes of doing research, we have separated dependency checks into 4 types: factual, inter-component, aspects, and general. For each dependency check in each of the four categories the precise dependency relationship must be explicitly defined. Finally, a user can also invoke analysis tools such as real-time analysis and reliability analysis (similar to what is found in MetaH).

3.1 Definitions

Before we describe more details of our research we define a few key terms. In the literature, a software component is defined as a unit of composition with contractually specified dependencies and explicit context dependencies. A software component can be deployed independently and is subject to composition to third parties. Software components are mostly about large scale reuse of software assests across organizations. For embedded systems we have to look at components differently. Embedded system components must consider performance such as meeting deadlines and reliability requirements, cost, linkage to specific hardware, and provide global analysis. It is likely that components need to be domain specific and there may be little need for third party use, rather they are more specific for a given enterprise. The components must have attached to them significant reflective information that explicitly relates to real-time, reliability, power, size, and cost issues. This is in contrast, for example, to a JINI component which has a business card describing the component which contains information such as the vendor, version number, and the container file where the code resides. We expect that more reflective information will be added to Jini components as its use broadens. Components in VEST can be fragments of code, functions or objects.

Components may be created in a hierarchy; components can be made up of subcomponents. In an OS, a component such as task management may be constructed from subcomponents such as create task, delete task, and set task priority.

An infrastructure is the essential part of the system which supports the tailored embedded system. The infrastructure, once chosen, usually does not change. The infrastructure contains basic code, data structures, hardware, mechanisms for dynamic composability, and underlying assumptions such as whether this infrastructure supports preemption or not, and if security is ignored or not. Micro-components are those components that are used to compose the infrastructure. Typical micro-components are dispatch tables, indirection tables, interrupt handlers, plug and unplug methods, and proxies. It is important to note that systems like MMLite, Pebble, VxWorks, and MetaH provide an entire microkernel as the infrastructure. Our hypothesis is that these *infrastructures* are too large or not specialized enough for many embedded systems and hence we take the approach to construct the infrastructure itself first. Our belief is that there will be a relatively small number of infrastructures that will prove suitable to cover most applications.

4 The VEST Configuration Tool Set

In this section we describe the design of our tool set, called VEST (Virginia Embedded Systems Toolset) in more detail. This work is quite preliminary so we include key research questions that must be answered as well as our ideas for solutions. In providing the details we follow the structure of our tool architecture presented above.

4.1 Designing and Implementing Components, Subcomponents and Microcomponents

If we look at application specific OS work we can find many examples of components and subcomponents. Source code is available for some of these systems. We will populate our library with some of these components and subcomponents. Descriptions of hardware components also exist in the library.

We believe that embedded system components (both hardware and software) must be domain specific and exist in different levels of detail. Research questions include what are interface design guidelines for embedded systems, how can an open interface approach be used effectively, and how can issues such as security and preemption or non-preemption be addressed in the design and implementation of components? Our approach will use the notion of microcomponents to create (compose) the essential runtime infrastructure and then “other” components can be attached. We will compare our ability to develop tailored infrastructures with fixed infrastructures such as those found in microkernels. We will employ open interfaces and focus on cross cutting component issues for embedded systems. We will also use reflection to provide meta-level information regarding components and their intended uses. The reflective information will be used by the configuration tools and non-functional analysis. We have extensive experience with the design and implementation of reflective real-time systems [34, 36] and with specification languages for hardware platforms [24, 35].

4.2 Configuration Tools

Today’s configuration tools are very limited. Research questions include what are the essential infrastructure ingredients for a domain, what library structure is appropriate, what is doable by the tool and what needs to be left to the designer, and how do we incorporate rules and guidelines into the tools? VEST will enable the component construction of the infrastructure itself first by containing infrastructure micro-components as separate entities, and then construction of the embedded application. The infrastructure construction will likely include micro-components such as dispatchers or interrupt handlers, but also address some key underlying assumptions such as preemption versus non-preemption. The tools will have techniques to represent dependencies (of various types but focusing on embedded system issues and OS-like functionality) in an explicit manner, contain rules to aid in composition, and provide some semantic guarantees. It will use reflective information to set up the inputs for the non-functional analysis. The tools first support the creation of the functionality by composing code and then the mapping of that code to a runtime architecture (first to an active task/thread model and then to the hardware). We now consider each part of the toolset in more detail.

4.2.1 Infrastructure Creation

VEST allows for the construction of an infrastructure out of micro-components. Once an infrastructure is created it can be saved and used again in a different setting. A combination of experience and studying other systems will create the micro-components needed for infrastructure creation. We have developed some experience by constructing uOS [13] and have studied many of the current component based OSs to extract a set of microcomponents. This set includes:

- interrupt handlers
- indirection tables
- dispatchers
- plug and unplug primitives
- proxies for state mapping

Infrastructure differences fundamentally depend on issues such as preemption versus non-preemption, or will the system need security or not.

4.2.2 Embedded System Creation

Once an infrastructure has been composed the rest of the OS components and application code must be configured and composed into the system; first the passive collection of code into the modules of the system and then mapping the executables into runtime active structures (tasks/threads). Listing, browsing, and searching are features required. Rules and/or guidelines for what to do and not do can also be built into this phase.

4.2.3 Dependency Checks

Due the complexity of dependencies we break these checks into 4 types: factual, inter-component, aspects, and general. For each dependency check supported by the tool, explicit definition of that dependency is required together with the availability of the appropriate information. Note that it is our emphasis on dependencies among components that is one distinguishing difference of our approach from those such as MetaH.

Factual The simplest, component-by-component dependency check is called factual dependency checking. Each component has factual 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. If a particular component is classified as the most important one, it is trivial to check that this remains true via a priority assignment. The greater number of factual dependency requirements that are checked by VEST the more likely it is that simple mistakes are avoided. We are currently developing a large list of dependency checks for OS-like components.

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 [17] 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 application specific operating system issues 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 semaphores, counting sempahores, 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 asopects, 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 the next section.

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 below).

4.3 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 2. 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 [42], 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 3. In this figure 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.

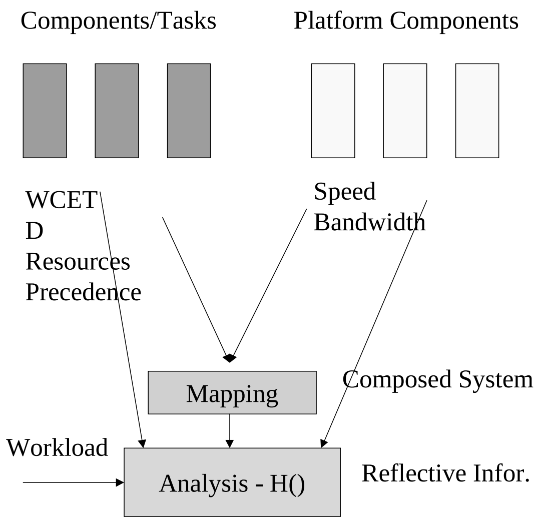

Figure 2: Real-Time and Concurrency Example.

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.

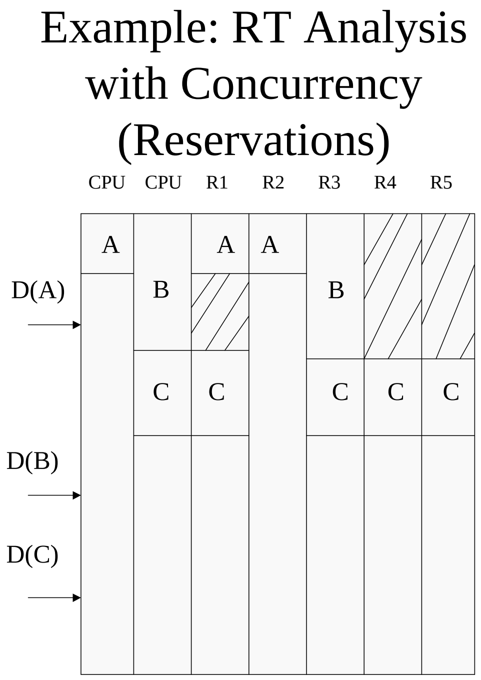

Figure 3: Feasible Schedule.

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.

As we said, many component based systems produce a collection of software without any analysis or minimal analysis. Research questions include how does one analyze the real-time, dependability, and security of the resultant system? We are developing system-wide analysis techniques for cross-cutting non-functional attributes such as real-time and dependability. While we can never be complete in the sense of building a tool that can do the analysis for any system whatsoever, we can provide analysis for particular situations of interest. In those cases we will be providing significant added value.

5 Our Future Work - Approach

For the solutions we develop to be useful they must be applicable to a wide variety of systems. Our approach will include performing the basic research on the questions posed above while investigating 4 types of systems (small fixed embedded systems, systems with hot swappable software, highly adaptable systems, and network-centric systems). These 4 types cover a wide range of embedded systems. For each of these 4 types of systems we will create the infrastructure, the components, perform the analysis, and evaluate the resultant product. We will begin by producing fixed and static embedded systems such as found in wristwatches or mobile multimedia phones. Second, we will try to determine the utility of our solutions for hot swappable OS-like software. This could include software in a factory process controller. Such solutions would enable software evolution, modifications due to faults, or due to performance mode changes. We will also investigate dynamically reconfigurable systems where the on-line system includes reflective information (taken from the off-line tool kit and analysis tools). This would support flexible embedded systems such as future set top boxes. Finally, we will extend our solutions into a network centric domain by looking at developing Internet appliances via our techniques.

6 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 function. A collection of components must interoperate, satisfy various dependencies, and meet non-functional requirements. These systems cannot all be built from scratch. Our components, configuration tool and analysis techniques and tools will substantially improve the development, implementation and evaluation of these systems. This could have beneficial impact on products from smart toasters, to set top boxes, Internet appliances, and factory controllers.

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] 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.

[13] Haskins J., Stankovic J., (2000), muOS, TR, University of Virginia, in preparation.

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

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

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

[17] Lopes, C., and Kiczales G. (1997), D: A Language Framework for Distributed Programming, TR SPL97-010, Xerox Parc.

[18] 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.

[19] Liu X., et. al. (1999), Building Reliable High-Performance Communication Systems from Components, *SOSP*, Vol. 33, No. 5.

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

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

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

[23] Morris R., Kohler E., Jannotti J., and Kaashoek M. (1999), The Click Modular Router, *SOSP*, Vol. 33, No. 5.

[24] Niehaus D., Stankovic J., and Ramamritham K. (1995), A Real-Time Systems Description Language, *IEEE Real-Time Technology and Applications Symposium*, pp. 104-115.

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

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

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

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

[29] Ready J. (1986) , VRTX: A Real-Time Operating System for Embedded Microprocessor Applications, *IEEE Micor*.

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

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

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

[33] Siegel J. (1998), OMG Overview: Corba abd OMA in Enterprise Computing, *CACM*, Vol. 41, No. 10.

[34] 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.

[35] 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.

[36] 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.

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

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

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

[40] Vestal, S., (1997), MetaH Support for Real-Time Multi-Processor Avionics, *Real-Time Systems Symposium*.

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

[42] 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.