CAmkES: A Component Model for Secure Microkernel-based Emebedded Systems

Ihor Kuz, Yan Liu, Ian Gorton, Gernot Heiser

National ICT Australia [1]
University of New South Wales, Sydney, Australia

Abstract

Component-based software engineering promises to provide structure and reusability to embedded-systems software. At the same time, microkernel-based operating systems are being used to increase the reliability and trustworthiness of embedded systems. Since the microkernel approach to designing systems is partially based on the componentisation of system services, component-based software engineering is a particularly attractive approach to developing microkernel-based systems. While a number of widely used component architectures already exist, they are generally targeted at enterprise computing rather than embedded systems. Due to the unique characteristics of embedded systems, a component architecture for embedded systems must have low overhead, be able to address relevant non-functional issues, and be flexible to accommodate application specific requirements. In this paper we introduce a component architecture aimed at the development of microkernel-based embedded systems. The key characteristics of the architecture are that it is has a minimal, low-overhead, core but is highly modular and therefore flexible and extensible. We have implemented a prototype of this architecture and confirm that it has very low overhead and is suitable for implementing both system-level and application level services.

Key words: component architecture, microkernel, embedded system,

Email addresses: ihor.kuz@nicta.com.au (Ihor Kuz), jenny.liu@nicta.com.au (Yan Liu), ian.gorton@nicta.com.au (Ian Gorton), gernot.heiser@nicta.com.au (Gernot Heiser).

[1] National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

1 Introduction

Traditional methods for developing embedded systems are resulting in increasingly unreliable embedded software. As embedded hardware capabilities increase, the ability to add more functionality to embedded systems also increases, leading to a growing complexity of embedded system software. However, while the complexity of the software increases, the methods and technologies used to develop have not changed significantly. While arguably sufficient for small systems, these methods and technologies are insufficient for building the larger and more complex systems being developed today.

Overcoming this problem requires the application of more advanced software engineering techniques to help ensure improved quality and more efficient development of embedded software. Component-based software engineering (CBSE) is a technique that is particularly well suited to this problem. CBSE provides a way to compose systems from independent, well-defined building blocks. Organising software in this way helps to provide structure and improves the reusability of code. It also improves flexibility by allowing components to be added and removed from a system (possibly at run-time), as well as allowing components developed in different languages to interact with each other. This means that developing and maintaining software becomes overseeable and more efficient. Furthermore, CBSE also enables independent development of components which means that specialised expertise can be (independently) concentrated on different parts of the system as required.

While CBSE has seen a wide adoption in the domain of enterprise computing, there are major differences between software for enterprise systems and software for embedded systems that prevent us from simply taking enterprise CBSE technologies and applying them in the embedded systems domain. The differences between embedded and enterprise systems fall into two categories: resource restrictions and non-functional requirements. Unlike enterprise systems, embedded systems have considerable resource restrictions. Deployment, cost and size concerns lead to significant restrictions in processing power, memory size and energy resources. Developers of software for embedded systems must ensure that their software can perform sufficiently on slower processors, can fit into reduced memory, and can run efficiently in order to conserve energy.

Embedded systems also have non-functional requirements such as timeliness, safety and dependability, that are less relevant to enterprise systems. Embedded systems are often real-time systems, which means that they have a temporal aspect to their behaviour. Software developed for such systems must be predictable so that its temporal properties can be analysed and reasoned about. The software must also be written in such a way that strict temporal deadlines can be dependably met. Besides being real-time systems, many embedded systems are also deployed in safety (or mission) critical applications. The software developed for such systems must

not fail. Software failure in such applications could lead to mission failure, damage to material and even loss of lives. Finally, many embedded systems are deployed in environments where they cannot easily be maintained or replaced. This means that the software must be reliable and counted on not to fail. Alternatively, if the software does fail, the system must provide mechanisms to notice the failure and rectify itself so that it can continue functioning.

In this paper we propose a component model and associated architecture targeted specifically at the development of embedded systems.[2]

The main contribution of this paper is a component model and architecture designed to run on a small microkernel-based operating system. It is meant to be used to develop components that provide OS services (such as drivers, file systems, network stacks, etc.) as well as application components that make use of the underlying OS services. The key feature of the model is that it is highly flexible and extensible, and has an extremely low overhead.

In the next section we discuss the role of operating systems in embedded systems and review existing component models, both those meant for enterprise computing and those meant for embedded systems. Afterward, in Section 3 we present the design of our architecture. In Section 4 we discuss how non-functional properties and requirements are dealt with in our architecture. We have built a prototype of the system which is discussed in Section 5. Using this prototype we have performed experiments to measure the overhead imposed by our component architecture, we provide an overview of these experiments and discuss their results in Section 6. Finally, we conclude with observations about the overhead costs of our component architecture and discuss future work.

2 Background

Due to their unique resource and non-functional constraints, embedded systems have different operating system (OS) requirements than traditional and enterprise computing systems. Likewise they impose different constraints on the software engineering approaches used to build them. In this section we discuss the use of operating systems in embedded systems and look at existing component models that have been designed for the development of embedded systems.

[2] Note that while we are initially targeting non-distributed systems, we have taken care that the model remains applicable to distributed embedded systems as well.

Often the resource restrictions and the highly specific nature of an embedded system lead developers to choose not to use an operating system at all. Instead, all resource management functionality is implemented directly in the application software. More likely, however, an embedded system will be based on a real-time operating system (RTOS). An RTOS provides OS functionality tailored toward the needs of real-time applications. Currently, as the capabilities of embedded hardware increase, we find that many general-purpose operating systems (such as Windows and Unix) are being deployed in embedded systems. Usually these operating systems are modified to decrease their size, and have some real-time functionality (such as real-time schedulers) added to them.

While there has been much focus on the real-time aspects of embedded operating systems, few existing embedded operating systems sufficiently address issues of protection, reliability and trustworthiness. With safety critical embedded systems becoming more complex and security threats becoming more prevalent in the embedded systems domain, security, reliability and trustworthiness requirements are becoming more acute. Microkernel-based operating systems are stepping in to fill this void (as evidenced by the microkernel-based embedded OSes from vendors such as Green Hills, QNX, Sysgo, etc.).

In a microkernel-based system, a minimal set of operating system functionality (most importantly memory protection and inter-process communication) is implemented in the kernel, while all other functionality is implemented as services that run outside the kernel. Since the microkernel is the only code that runs in a processor's privileged mode, there is only a small amount of code that can directly interfere with the proper function of the whole system (either by causing irregular system behaviour, or by violating system security). All other code runs in a processor's unprivileged user-mode and is protected by the microkernel from direct interference by unrelated code. All of this has a number of important implications for embedded systems. The minimality of a microkernel means that it does not use excessive resources. Furthermore, it also means that it is easier to analyse the code to verify that it is bug-free. The fact that most functionality runs in user-mode and is protected from other user-mode code means that microkernel-based systems can provide good partitioning between different applications and even between different OS services. This means that buggy or malicious code is not as likely to take the whole system down as it would be if there was no protection. Furthermore it means that misbehaving code can be spotted, stopped and replaced without bringing the system down.

Since microkernel-based systems promote the separation of functionality into separate services, there is a strong synergy between the approach of microkernel-based system design and CBSE. On the one hand, the CBSE approach of modeling a

system as interacting components fits the model of an operating system as a set of interacting services. On the other hand, a microkernel provides exactly those features (protection and communication) needed to build secure and reliable systems consisting of interacting components.

2.2 CBSE for Embedded Systems

Many of the component models based on enterprise component technologies such as .Net, J2EE and CORBA Component Model (CCM), fail to address the critical issues of using components in embedded systems, including resource constraints, real-time performance, fault tolerance, energy use, etc. Recent research and engineering efforts have focused on establishing component-based software engineering disciplines targeted specifically at embedded systems. We roughly divide this related work into three categories. (1) Component models that target specific application domains such as field devices, consumer electronics, vehicular systems, etc. Examples of such models include PECOS (Genßler et al., 2002), Koala (van Ommering et al., 2000) and Save (Hansson et al., 2004). A detailed survey of other domain-specific component models is available (Möller et al., 2004); (2) Component-based operating systems such as TinyOS (Hill et al., 2000), Pebble (Gabber et al., 1999) and Think (Fassino et al., 2002); (3) Middleware-based component models tailored for embedded and real-time systems and focusing on non-functional attributes. Examples of these include CIAO (Wang et al., 2001), COMQUAD (Göbel et al., 2004) and PECT (Wallnau, 2003).

PECOS was originally designed in the domain of field devices. It has a data-flow-oriented model where components communicate by sending or receiving data. Components can be either active or passive, with active components having their own thread of control. PECOS does not focus on non-functional properties other than timeliness and performance optimisiations.

Koala is designed by Philips and is focused on software product-line development of consumer electronic devices. Koala focuses mainly on restricted resource constraints and provides a lightweight component model. Koala components communicate through remote procedure call style interfaces. Only static binding of components is supported and all invocations are hard-coded into components so that the runtime overhead is minimised. Koala does not take into account non-functional properties such as timing, safety and security.

The SAVE (SAfety critical components for VEhicular systems) project has developed a component model, SaveCCM, targeted at vehicular systems (Hansson et al., 2004). SaveCCM is part of a component-based development framework called SAVEComp (Tivoli et al., 2005). SaveCCM supports static configuration of components and component bindings. It focuses on quality attributes such as timeliness

and predictability of component behaviors. Analysis tools are actually provided by the SAVEComp environment. An example is using SaveCCM components to composite control loops.

TinyOS is an open-source operating system designed for wireless embedded sensor networks. It features an event driven component model, which is complementary to typical hardware models. This makes it possible to implement components in hardware, or to implement hardware devices as software components. TinyOS supports only static component bindings. There are no facilities for dynamic component creation or destruction, dynamic binding, or dynamic allocation. TinyOS provides very primitive services and does not provide any protection.

CIAO (Component-Integrated ACE ORB) is a CCM implementation built on top of TAO (The Ace ORB). It is optimised for distributed real-time embedded systems (DRE) by modeling DRE-critical systemic aspects, such as QoS requirements and real time policies, as installable and configurable units. Following the CCM specification, components interact using interfaces and events. CCM components run in a container, which provides them with an execution environment. The overhead of common container-management operations must be minimised by a CCM implementation to meet the resource constraints of an embedded system. Evaluation of CIAO performance based on a benchmark measurement indicates that by optimising the component communication, CIAO's CORBA 3.x CCM capabilities do not add significant overhead above and beyond its underlying TAO CORBA 2.x implementation (Krishna et al., 2005). However the ORB (Object Request Broker)-based communication in TAO can still impose overhead that is not affordable for strict resource-bound embedded systems.

Research effort has also been devoted to component models and architectures designed to fulfill non-functional requirements. COMQUAD (COMponents with QUantitative Properties and ADaptivity) has devised a component container architecture that splits the architecture into a real-time capable (RT) part and a non-real-time capable part (NRT). The requirement of a specific non-functional quality attribute is specified in a contract. The COMQUAD container uses JBoss, a J2EE component container, to coordinate the interaction between the NRT and RT parts and guarantee the quality attributes as specified in the contract. JNI (Java Native Interface) is used to invoke the native code of RT parts from the COMQUAD Java-based container. The use of Java makes this architecture less suited to resource-constrained embedded systems

PECT (Prediction-enabled Component Technology) provides a general framework for reasoning about quality attributes of component-based systems. The focus is on how to apply an analytical theory to predict a specific quality attribute for a component-based system given the component specification and the properties attached to each component. PECT works at the conceptual level and has to be separately instantiated for individual cases.

Schmidt (Schmidt, 2003) has proposed a dynamic model that defines configuration and behavioral contracts and associates these to components and architectural assemblies of components. This enables the prediction of extra-functional properties during architectural design. However, this work is mainly focused on distributed real-time systems and does not address all the critical embedded-systems issues.

3 Component Architecture

In this section, we present our layered component architecture called, *CAmkES* (Component Architecture for microkernel-based Embedded Systems). The purpose of the architecture is to provide support for developing embedded systems on top of microkernels. The architecture provides a component model, standard interfaces and component definitions, component implementations, standard services, and support for various architectural patterns suited to embedded systems.

Before presenting the details, it is important to emphasize the relationship between our component architecture and the underlying microkernel-based operating system. Since the architecture is meant to be used to develop both application and operating system components, one of the driving motivations of the design is tight integration with the operating system. This results in two requirements. First, the architecture must directly make use of any mechanism provided by the OS (this includes inter-process communication, memory management and protection) and not reimplement similar mechanisms. Second, all mechanisms provided by the architecture must be efficient enough that they can be used by operating system components without creating significant performance penalties for the rest of the system.

3.1 Overview

The CAmkES layered architecture is shown in Figure 1. At the bottom is the **hardware layer**, which includes the CPU, memory, bus and any other devices. On top of the hardware layer is the RTOS (Real-Time Operating System) layer, which consists of a microkernel and a supervisory OS (in our case the microkernel is L4 and the supervisory OS is Iguana). Further support such as device drivers, file systems and network stacks can be included in this layer, however, these services can also be implemented as CAmkES components and can, therefore, reside at a higher layer instead.

The **CAmkES core runtime** forms the foundation of the component architecture, providing an execution environment and the basic services required to deploy CAmkES components. The core runtime supports static components and component compositions. This means that component instances are only created at system

initialisation (i.e., boot) time and that connections between components are established at design time and cannot be created or modified dynamically at run-time. This allows us to minimise overhead (for example by inserting direct procedure calls into components, thus avoiding inter-process communication (IPC) and marshaling overheads) for the most basic component-based applications.

More advanced component features are provided by extensions that run on top of the core runtime in the **extension layer**. The extensions are themselves components that make use of the core runtime features. The extension layer is designed to address the various aspects of supporting dynamic components, including dynamic creation and destruction, dynamic binding, dynamic configuration, etc.

Frameworks further extend the functionality of the component architecture by providing components and services specifically geared to particular application domains. Finally, user-defined components combine with the underlying layers to form complete **applications**.

This layered architecture provides a good separation of concerns. Putting advanced component features in the extension layers separate from the core runtime allows the core runtime overhead to be minised. For example, support for dynamic binding implies increased overhead with regards to code size and processing. By placing this support outside the core we can limit this overhead cost to those systems that actually require dynamic binding. The extension layer will clearly have fewer resource constraints than the core runtime layer. In the rest of this paper, we focus on the design and implementation of core runtime part of our component architecture. The extension layer and frameworks are part of our ongoing research effort.

3.2 Feature Summary of the Core Runtime

The core runtime forms a key part of the CAmkES layered architecture. Its main features are listed below:

Modular: The core runtime only includes features that are really needed for any particular target application. Other features can be added in the form of extensions. This modularity allows users to extend the runtime with special features on an as-needed basis depending on their specific applications.

Simple: The core runtime is lightweight and only focuses on static components and their composition. This minimises the overhead introduced by the component architecture.

Predictable: For static components, stubs and glue code are inserted into the components themselves at compile time. This allows the temporal behaviour of the resulting application to be analysed. Predictability can be achieved for static components with static composition.

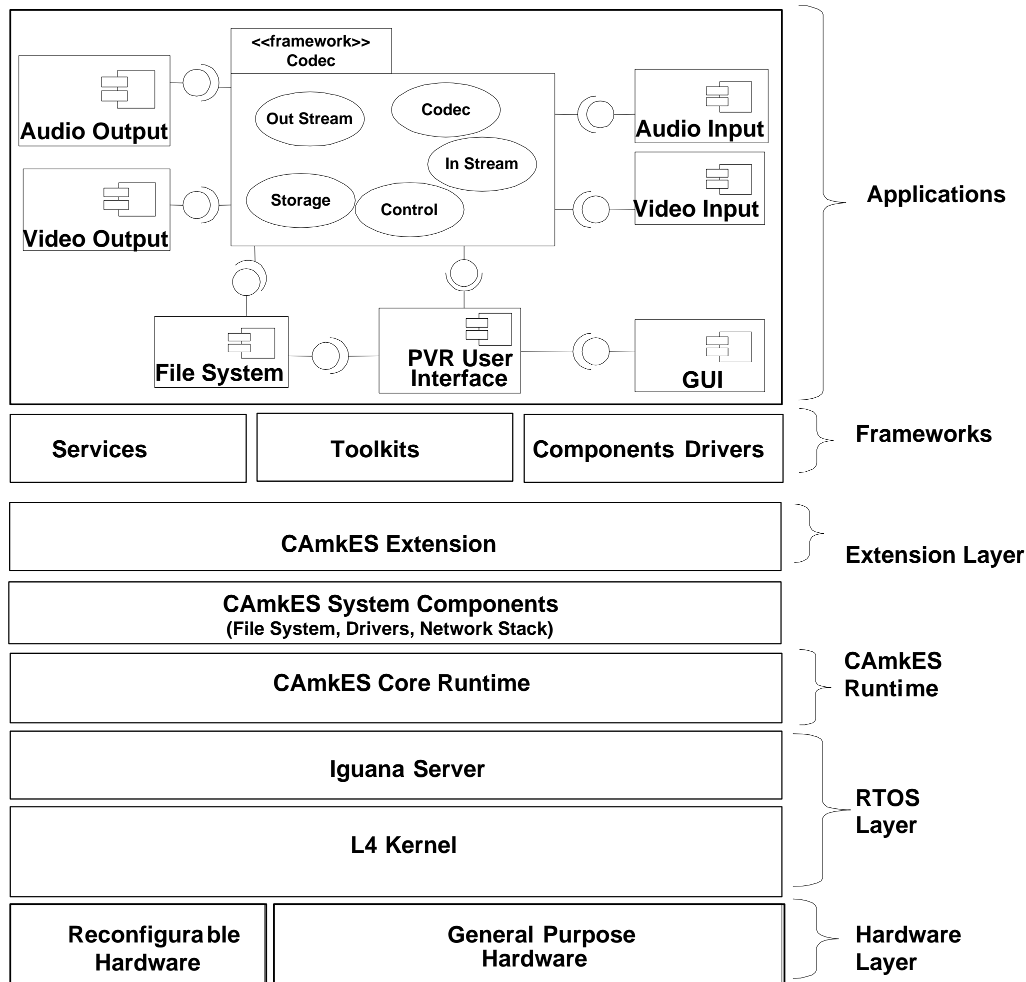


Fig. 1. The CAmkES layered architecture.

3.3 Component Model

The CAmkES core runtime supports a component model that includes the following architectural elements, namely *components*, *interfaces*, *connectors*, *connections*, *compositions* and *configurations*.

3.3.1 Component

A *component* is the basic unit of encapsulated behavior, which is used to organise operations and data into interfaces that have well defi ned semantics and behaviors. Components expose interfaces that allow applications and other components to access their features.

A component can be either *passive* or *active*. A passive component is similar to a language level object. It provides access to methods but does not have a thread of

control. An active component, on the other hand, does contain its own thread of control.

3.3.2 Interfaces

CAmkES supports three types of interfaces, namely *remote procedure call* (RPC), *event* and *dataport* interfaces. An interface is defined by a CAmkES-specific interface definition language (IDL), which is based on the CORBA IDL (Object Management Group, 2004).

RPC interface: An RPC interface defines synchronous communication between components by remote procedure calls. A component must explicitly state whether it *provides* or *uses* an RPC interface.

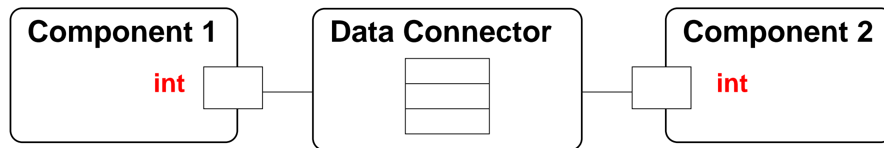
Event: CAmkES supports a publish/subscribe event model. Events are used for asynchronous notifications between components and they are *emitted* or *consumed* by components at event interfaces.

Dataport: The dataport interface represents shared variables that allow components to transfer data between each other. A pair of connected dataports represents the same variable or the same range of memory. This is unlike the data-only interfaces (or ports) defined in other component models, where they are used to transfer or copy data between components, but do not have sharing semantics. True data-sharing allows us to reduce performance overhead as compared to copying.

3.3.3 Connectors and Connections

Our component model encapsulates communication between components in explicit architectural elements called *connectors* and *connections*. A connector is a runtime pathway of interaction between two or more components (Clements et al., 2002, Part I. Chapter 3). In our model, a connector has a name and a list of interface types that it connects. For example, a connector connecting a pair of dataports describes a data sharing relationship between them. A connector can describe 1-to-1, 1-to-many, many-to-1 and many-to-many relationships among interfaces. A connection is the instance of a connector. It is associated with two or more components. An example of the use of connectors and connections to define a composite component is shown in Figure 2. Details of composition are discussed in Section 3.3.4.

The use of connectors and connections in our model leads to a unique feature of CAmkES: being able to encapsulate data sharing between components as an architectural connector. For example, two components may require synchronised dataport connections. This can be done using different synchronisation mechanisms such as mutexes, semaphores, spin locks, etc. In our model each mechanism is defined and implemented as a separate connector. Communicating components sim-



Define a connector:

```
connector DataConnector {
  IntPort port1, IntPort port2;
}
```

Use a connection in the composition:

```
composition {
  component component1 c1;
  component component2 c2;
  ...
  connection DataConnector conn
    (c1.port1, c2.port2);
}
```

Fig. 2. Using connectors to connect components.

ply use a dataport interface, with the synchronisation being taken care of by the appropriate connector. Which connector is used depends on the connection specification. This approach is in line with the general vision of software architecture (Shaw, 2005).

The flexibility provided by connectors is, of course, not limited to dataport mechanisms, and can equally be applied to RPC and event mechanisms. Architectural connectors provide a means for separating concerns, that is separating a component's functional behavior from its interactions with others. This improves the extensibility of the system since a communication protocol can be replaced without affecting the component implementation. Also the use of connectors and connections facilitate the representation, analysis and enforcement of requirements at runtime (Hansson et al., 2004; Genßler et al., 2002).

3.3.4 Composition

In CAmkES, an entire application is modeled as a *composite component*, i.e., one that contains instances of other components. Component composition makes use of connectors and connections. A composite component, like a non-composite component, generally exports interfaces, however it does not directly implement these interfaces. Instead, the interfaces are connected directly to constituent components, which provide the implementations. An example of this is shown in Figure 3 where Component 2 implements all of Compound 1's interfaces. For the core runtime all instances of composed components are created at compile time.

3.3.5 Configuration

A component can also have one or more attributes, whose values represent the component's status or settings. These values are specified, not inside the component definition, but in a separate *configuration specification* when components are

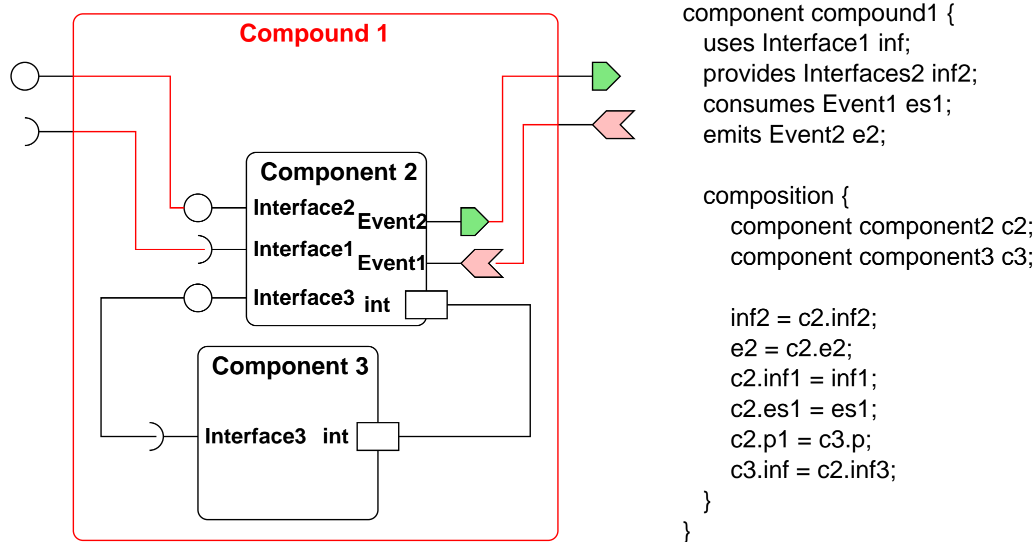


Fig. 3. Component composition.

assembled. A component is instantiated with the attributes specified at compile time. The use of configuration is quite flexible in the CAmkES component model. Both attributes and configuration specifications can also be applied to compound components and connections. This configuration model provides a way to address both functional and non-functional requirements for embedded systems built on CAmkES. An example of using configuration to specify secure access control is presented in Section 4. Further investigation into addressing non-functional properties and requirements using this configuration model is part of our ongoing work.

3.4 Computational Model

The CAmkES component model is general and not targeted at any specific embedded-application domain. As a result it does not prescribe any specific execution (or computational) model. For example, systems built based on the CAmkES model can be control-flow oriented, where executions are triggered by invocations on RPC interfaces or events through event interfaces. It can also be data oriented, with access to shared data between components being established through dataports. The CAmkES core runtime provides a library of default connectors for RPC, event and dataport interfaces.

4 Non-Functional Properties

The CAmkES component model, together with its core runtime support, addresses the restricted resource aspect of embedded systems, both in terms of memory and

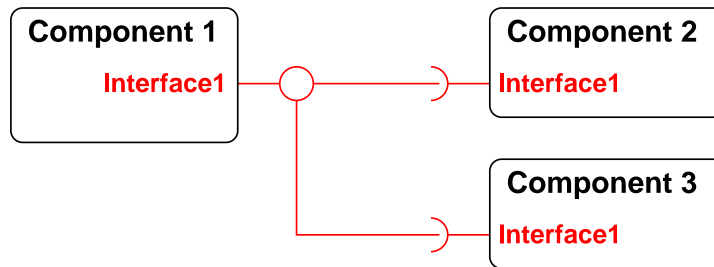
processing constraints. Since CAmkES also targets the development of operating system components, this imposes stricter constraints on its overhead. Because of this, rather than reimplementing various mechanisms in the architecture, our support for non-functional properties largely relies on mechanisms already provided by the OS. Furthermore, since the core model is static and components, connections, and configurations are all known ahead of time, glue and stub code are aggressively optimised to reduce the overhead introduced by the component model.

Besides resource restrictions, the safety and security properties of embedded system are of utmost importance. These are addressed in our model through a tight integration with an underlying secure microkernel-based operating system (L4/Iguana). L4/Iguana, which has been developed specifically for safe and secure embedded systems (Heiser, 2005), provides protection mechanisms such as capability-based access control to encapsulate complex software into protected components. [3]

The basic security model provided by CAmkES is based on Iguana's capability model and involves controlling and restricting access to components. In particular we use configuration specifications, as discussed in Section 3.3.5, to specify a capability list in the configuration of a connection. Figure 4 shows a scenario where Component 1 provides interfaces to Component 2 and Component 3. In this example, *c2.interface1.method2=x* means that Component 2 can access method2 of interface1 provided by Component 1. Similarly, *c3.intface1.method3=-* means Component 3 cannot access method3 defined in interface1 provided by Component 1. We can see from the configuration of the capability list that Component 2 is given permission to invoke method2 defined in interface1, while Component 3 does not have the right to invoke method3, but can invoke method4. These access restrictions are enforced by Iguana at runtime. Note that by relying on mechanisms already provided by Iguana, we can support this model without adding significant overhead to the CAmkES runtime. Furthermore, since the access control is part of the functionality of a connector, it is possible to use different access control mechanisms by using different connectors.

Given our static model, it would also be possible to analyse a composition at build time to ensure that no access restrictions are violated. In such a situation, the runtime access controls would not be required, which would save much runtime overhead. This approach to safety and security has not yet been followed up, but is something that we wish to look into in the future.

[3] Iguana's access control is partially based on the model of Mungi (Heiser et al., 1998), a single-address-space operating system developed by the same group as Iguana.




```
configuration {
  ...
  connection SimpleInfConnector conn {
    capabilitylist {
      c1.interface1.* = x;
      c2.interface1.method2 = x;
      c3.interfaces1.method3 = –;
      c3.interface1.method4 = x;
      ...
}}}
```

Fig. 4. Configuration of access control.

5 Implementation

To validate our design we have implemented a prototype of the CAmkES core runtime. The main goal of this protoype is to show that the design can lead to a low overhead, minimal and efficient implementation. Besides the fact that we have taken effort to make the design of the core runtime minimal and modular, we have also made sure that the implementation maps onto the underlying operating system mechanisms with as few mismatches as possible.

5.1 L4 and Iguana

Since the CAmkES architecture is designed to run on the L4/Iguana embedded operating system we provide a short overview of the L4 microkernel and Iguana supervisory OS before continuing with a discussion of the prototype and its performance.

L4 is a second-generation operating-system microkernel. It provides a small set of fundamental mechanisms and abstractions that run privileged in kernel-mode, leaving typical operating systems tasks (such as process management, device drivers, interrupt handlers, file system, etc.) to be implemented and run as unprivileged user-mode servers. L4's main features include memory protection, memory mapping between address spaces, low inter-process communication (IPC) overhead (very close to the host platform's hardware-dictated context-switch costs), and a small

footprint. More information about microkernels and L4 can be found in (Liedtke, 1996; L4 Community, 2005)

Due to its minimal nature, L4 does not provide much of the functionality that one would normally expect from an operating system. In particular, since L4 also avoids implementing policy, it does not provide any specific model of operating system services such as process management, memory and address space management, access control, etc. This task is left up to a supervisory OS running in user-mode on top of the microkernel. In our case this OS is Iguana.

Iguana is specifically designed for use in embedded systems. It has low memory and cache footprints and provides basic services such as memory management, protection management, a remote procedure call (RPC) based IPC mechanism, low-overhead data-sharing and a basic device driver framework. Iguana provides a single non-overlapping address space that is shared by all threads. In Iguana the concerns of memory protection and memory translation (i.e., providing address spaces) are separated. This means that despite all threads sharing the same address space, Iguana also provides memory protection. The separation of memory protection and translation also means that Iguana-based systems can be readily deployed on processors without virtual memory. Moreover, this allows increased performance to be gained on processors (such as ARM7 and ARM9) with virtually-addressed caches where an overlapping address space layout would require a cache flush on every context switch.

Iguana provides a client-server model of interaction. Applications and operating system services run as Iguana servers and interact with each other using IPC. An Iguana server consists of a thread with an associated memory section running in a protection domain. Threads are Iguana's basic units of execution and scheduling, and memory sections are the basic units of virtual memory allocation and protection. Protection domains provide memory protection between threads executing different programs (or servers). A protection domain roughly corresponds to the concept of a task or process in other systems, except that a protection domain does not define a separate virtual address space. Threads in the same protection domain have full access to each others memory, while threads in different protection domains are protected from each other and can access each others memory only if permitted to by the access control system. This is implemented using capabilities, which are security tokens that define access rights to memory sections and threads. Thus, in order to access a memory section in another protection domain a thread must hold an appropriate read or write capability for that memory section.

Each Iguana server implements a server-specific interface that consists of a set of methods that can be invoked on that server. Iguana provides a remote-procedure-call style of IPC. A client invokes a server's method by calling a local stub function. The stub marshals parameters and sends a message to the server using underlying L4 IPC mechanisms. At the server side, the message parameters are unmarshaled

by a similar stub and the appropriate function is invoked. Before invoking another server's methods, a session must be established between the client and server. Besides setting up a communication channel, establishing a session also involves ensuring that the communicating parties hold the right capabilities. In order to invoke a method on a server in another protection domain, the invoker must hold an appropriate execute capability for that server.

5.2 Mapping CAmkES to Iguana

In order to run CAmkES components on top of Iguana we provide a mapping of CAmkES concepts onto Iguana concepts. CAmkES components are generally placed in separate Iguana protection domains and are implemented as separate Iguana servers. This provides proper encapsulation and prevents other components (or processes) from purposefully or inadvertently accessing a component's internals. Furthermore, it allows the architecture to restrict access to a component's interfaces to authorised parties only, as shown in the example in Section 4. Note that the underlying OS makes use of hardware-based memory protection to enforce this.

CAmkES RPC interfaces map indirectly to Iguana interfaces. Unlike CAmkES interfaces, Iguana interfaces act as units of protection rather than encapsulation. In order to provide method-level access control this means that, when mapping to Iguana, the individual methods of a CAmkES RPC interface are translated to separate Iguana interfaces. We call these the *Iguana equivalent interfaces*.

In our prototype implementation, the components are active and contain dispatch threads that allow them to service Iguana RPC requests. CAmkES dataports map to shared Iguana memory sections so the sharing of memory sections in Iguana is managed by the memory management (or protection) unit and does not require any copying of data. CAmkes events map to Iguana asynchronus notifi cations, however, since the Iguana implementation of these is currently in a state of flux, they have not been included in the prototype.

Connections are mapped according to the interfaces that they connect. RPC connections naturally result in Iguana IPC communication. This is managed by stubs generated from Iguana IDL descriptions of the connected interfaces' Iguana equivalents. Dataport connections are implemented as shared Iguana memory sections. Dataport initialisation code takes care of setting up the memory sections and mapping these onto appropriate local variables in the relevant components.

Compound components do not map directly onto any Iguana entities. Since a compound component contains other components, but does not implement any functionality itself, it is not necessary to have a separate entity representing it. Instead, any access to a compound component's interface is routed directly to the compo-

nent actually implementing that interface.

Loading and initialising a CAmkES-based system proceeds roughly as follows:

- A boot image containing L4, Iguana, and CAmkES components is loaded into the system's memory.
- L4 starts and loads the Iguana user-mode server.
- Once loaded, the Iguana server proceeds to load and initialise its services.
- After basic services such as chipset drivers, naming, etc. have been loaded, a CAmkES loader routine is run. The loader routine is responsible for loading all components, initialising them and establishing connections between them.
- Connection establishment involves the creation of Iguana sessions, allocation of shared memory sections and the distribution of capabilities according to the component configuration specifications.
- Finally, once all components and connections have been initialised, component dispatch and control threads are started.

6 Evaluation

The main goal of our prototype was to show that the CAmkES architecture design can lead to a low overhead implementation. In this section we provide empirical evidence that this is the case. We focus both on the memory and performance overhead and show that it is minimal compared to standard Iguana overhead. We also provide a rough idea of the overhead that Iguana imposes compared to bare L4. Note that we do not include a discussion of the suitability of L4/Iguana as a base OS for embedded systems. A discussion on this can be found elsewhere (Elphinstone et al., 2005; Heiser, 2005). For this evaluation it is assumed that the overhead of L4/Iguana itself is acceptable. [4]

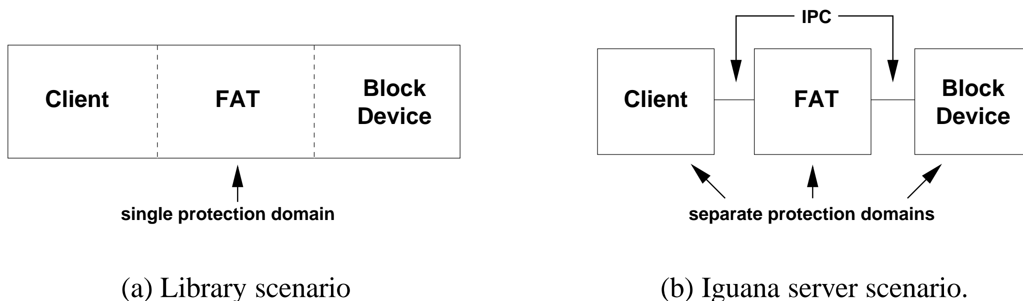
For this evaluation we implemented a filesystem service as a CAmkES component. We examined the size of this implementation and compared it to an equivalent implementation based on plain Iguana (one using IPC and one using no IPC). We also implemented a simple benchmark program to exercise the filesystem service's interfaces and measured the performance overhead introduced by the IPCs and additional CAmkES infrastructure code.

[4] There are, however, possibilities for optimising L4/Iguana and improving its overhead. A CAmkES based system will clearly benefit from any such optimisations.

6.1 Filesystem Service

For the filesystem service we implemented a simple FAT32 filesystem library, which was utilised in three different scenarios. In the first scenario the library was linked directly into a client program, which allowed the client to directly access the filesystem without having to use any IPC. In the second scenario we wrapped the library in Iguana code so that the functionality could be accessed from other protection domains using Iguana IPC. In the third scenario we wrapped the library in CAmkES component code, creating a FAT filesystem component. These three designs are illustrated in Figures 5 and 6.

In Figure 5(a) the FAT library is linked into the test program and all the code runs in a single protection domain. In Figure 5(b) the FAT code is linked into an Iguana server. The client runs in a separate protection domain and must use Iguana IPC to invoke the file system functions. The underlying block device (which simply provides a block interface to a region of memory) runs as a separate server in a different protection domain. In Figure 6 the FAT functionality is implemented as a component. We added an additional utility component so that we could test the overhead of compound components as well. The FAT and Util components are combined in a single compound FATFS component that provides separate file system and utility interfaces. This scenario also includes a separate Block Device component. Note that the components also provide some dataports. These are used to transfer data during the read and write operations.



(a) Library scenario

(b) Iguana server scenario.

Fig. 5. Library and Iguana server scenarios

6.2 Performance Overhead

A simple analysis of the design reveals basic information about the IPC overhead of the different scenarios. Whereas the library design has no IPC overhead, the Iguana design introduces at least two IPCs per invocation (one to invoke an operation and one to return the results). Most of the calls also require access to the block device which involves at least two more IPCs. The calls that transfer data also require shared data to be read and written. The IPC and data sharing overheads in

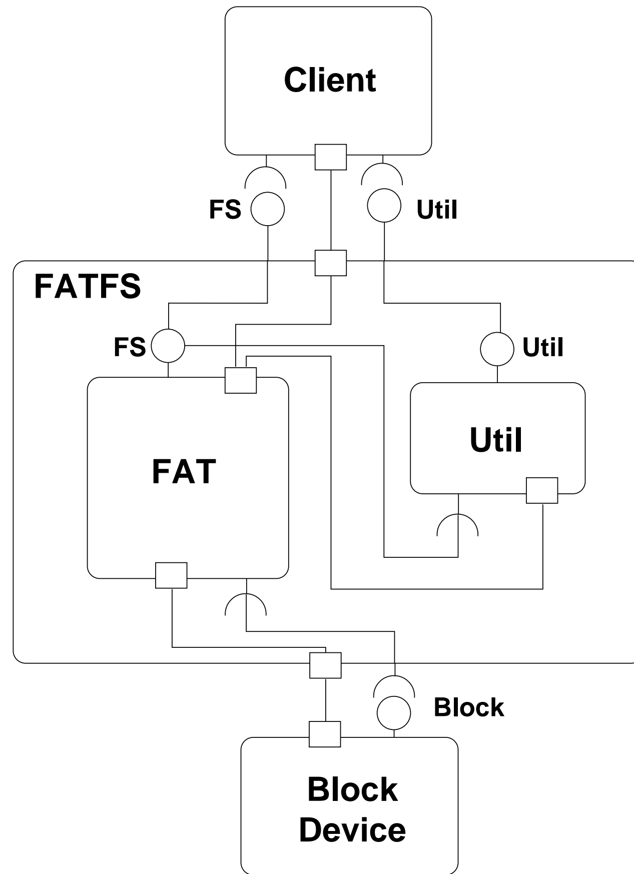


Fig. 6. CAmkES component scenario.

the component scenario are similar to the Iguana scenario overheads. Note that in the component design, there are extra connections between the compound component's interfaces and the actual components implementing those interfaces. In our implementation these connections disappear and the invocations are performed directly on the inside components. The case where the Util component is called adds at least two extra IPCs since the Util component must invoke operations on the FAT component.

We performed tests to determine the exact performance overhead imposed by both Iguana IPC and CAmkES as compared to the simple library scenario. The experiments were performed on a PLEB2, a small, locally developed embedded systems board featuring an XScale PXA255 processor with 32MB SDRAM and 8MB FLASH memory.

In order to calculate the overhead of Iguana and CAmkES on the performance of components we ran a series of tests exercising each method provided by our filesystem implementation. Figure 7 shows the results of these tests. The detailed results of our measurements are shown in Table 1. [5] The `null` operation measures

[5] The `mkdir` operation in the library version was broken and we therefore do not provide

the pure round trip time of an invocation going to the file system server and back. It shows that the extra overhead introduced by CAmkES is approximately $1\mu s$ or 2.65% overhead over the Iguana scenario. Other operations involve more than a single round-trip IPC (e.g., calling the block device component) and therefore have higher overheads. However, in all cases the overhead remains below 7%. For `read` and `write` operations, we also measured the execution times for different buffer sizes ranging from 1 byte to 4 KB. Figure 8 shows the average execution time of the read and write operations. We can see from Figure 8 that CAmkES performs very close to Iguana, the average overhead being within 5% of Iguana.

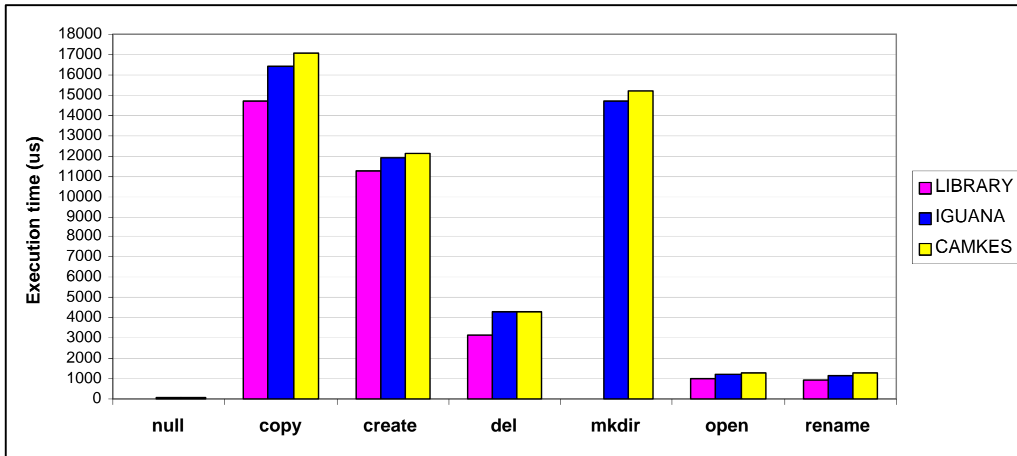


Fig. 7. Execution time of FAT file system operations.

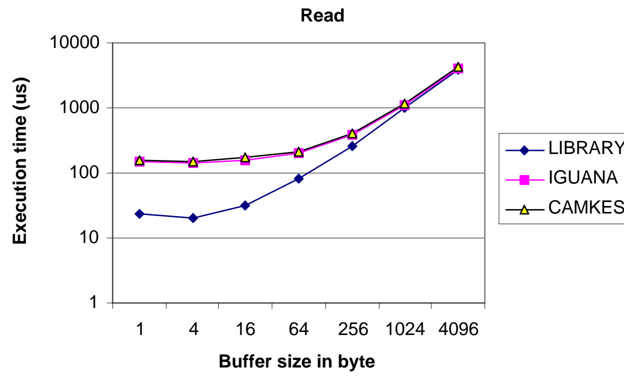
The benchmark results demonstrate that the CAmkES component model only introduces a very low level overhead and the overhead of most operations are within 5%.

6.3 Footprint Overhead

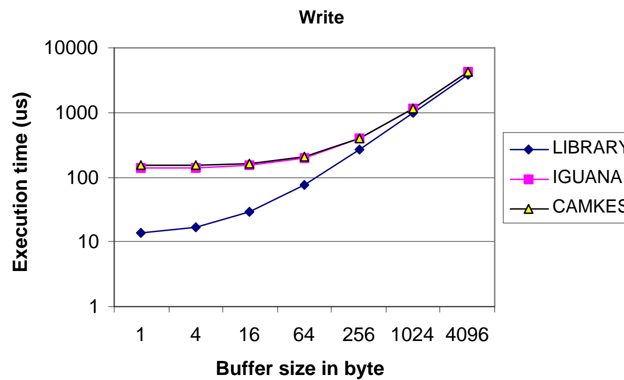
Besides performance overhead, the footprint or size overhead is also an important metric for embedded systems. There are several ways to measure the footprint of a system. Counting the lines of code gives an idea of the size and complexity of the source code that makes up a system. However, this does not always reflect on the final size of the running system. Code may be conditionally compiled based on configuration parameters, which means that not all the source code is used to create an image. Likewise platform specific code is often reimplemented for each platform that the code runs on and greatly inflates a system's code base. The static size of the loadable binary gives a more realistic measure of the system size. In the following analysis we will look at both of these measurements of CAmkES overhead.

With regards to lines of code (loc), the FAT file system library itself is 728 loc. The

a number for it.



(a) `read` operations



(b) `write` operations

Fig. 8. Execution time for `read` and `write` plotted on a logarithmic axis.

test client is 508 loc. The library scenario does not add any signifi cant code overhead to this. The Iguana scenario adds 361 loc in the form of generated stubs and extra code to make accessing the stubs more programmer friendly. The CAmkES scenario adds 786 lines of code compared to the library scenario. Compared to the Iguana scenario, CAmkES adds 425 lines of code. This extra code is mainly due to the fact that the CAmkES to Iguana mapping implements each method as an Iguana interface, which adds extra generated structural code for processing and initialising those interfaces. There is also some duplication of generated code due to similar connections (for example between the client and the FS component and the Util component and the FAT component) and the fact that the Util component is in a separate protection domain in the component scenario.

We also analysed the static image sizes of the Iguana and component scenarios. The overall sizes are presented in Table 2. The table shows a signifi cant overhead

Operation		Library (μs)	Iguana (μs)	CAmkES (μs)	CAmkES/Iguana
null	Average	0	44.21	45.38	2.646%
	Std dev	NA	1.578	2.977	
copy	Average	14677.54	16398.11	17078.02	4.146%
	Std dev	784.05	543.90	528.15	
create	Average	11269.69	11884.96	12103.49	1.839%
	Std dev	76.84	431.35	433.17	
del	Average	3161.44	4284.01	4305.82	0.509%
	Std dev	24.23	49.91	56.70	
mkdir	Average	NA	14688.44	15233.34	3.710%
	Std dev	NA	543.49	528.56	
open	Average	1006.03	1216.11	1283.22	5.518%
	Std dev	784.05	543.90	528.15	
rename	Average	954.3	1181.46	1258.34	6.507%
	Std dev	4.262	4.713	5.351	

Table 1
Benchmark results for the FAT file system operations.

(54.5%) of the component scenario compared to the Iguana scenario. This is due to the fact that the component scenario has an extra Iguana server running (the Util component). Since each server runs in a separate protection domain it must link in all the (standard) libraries that it uses, which makes up for a substantial part of the overhead. [6] If we look at the sizes without the Util component, we see that the overhead in the component scenario is much smaller (1.3%).

7 Conclusion

We have presented CAmkES, a component architecture for the development of embedded systems. It is designed to run on top of microkernel-based operating systems, which provides the features and mechanisms necessary to develop protected components with very low overhead. In this paper we focused on the component model and the core runtime support for components developed according to the CAmkES specification. While many of the individual features presented can be found in other models, we claim that the particular combination of features makes our model ideal for development of embedded systems. In this light, the most im-

[6] This is an issue with Iguana that will be resolved in the future.

Scenario	Server	text	data	total
Iguana	FAT	39008	3094	42102
	block	33264	2162	35426
	total	72272	5256	77528
Component	FAT	40040	3030	43070
	Util	39100	2158	41258
	block	33352	2138	35490
	total	112492	7326	119818

Table 2
Static image sizes (in bytes) for the Iguana and component scenarios.

portant features of our component model are that it is extensible, flexible and not restricted to any specific architecture style. Furthermore, in contrast to component models introduced in the related work, the data sharing architecture style enabled by the dataport interface and connectors reduces the overhead of communication compared to copying data between components.

We have implemented a prototype of the core runtime and devised a case study using it. We developed a FAT file system OS component using the CAmkES component model. Using this component we measured overhead introduced by the component model both in terms of the memory footprint and the round trip time of the invocation of each function defined in the interface. The results verify that the CAmkES component model has very low overhead (within 7% invocation overhead compared to the underlying OS).

In our future work, we will incorporate the management of other non-functional requirements such as timeliness into the core-run time. We will also implement the extension layer adding dynamic behavior, such as hotswapping of components, dynamic binding of interfaces and providing protection of access to newly created or added components.

References

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J., 2002. Documenting Software Architectures: Views and Beyond. Addison Wesley Professional.

Elphinstone, K., Heiser, G., Huuck, R., Petters, S. M., Ruocco, S., Nov. 2005. L4cars. In: Embedded Security in Cars (escar 2005) Workshop. Cologne, Germany.

Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G., Jun. 2002. Think: A software

framework for component-based operating system kernels. In: Proceedings of the USENIX Annual Technical Conference. Monterey, CA, USA.

Gabber, E., Small, C., Bruno, J. L., Brustoloni, J. C., Silberschatz, A., Jun. 1999. The Pebble component-based operating system. In: Proceedings of the USENIX Annual Technical Conference, General Track. Monterey, CA, USA.

Genßler, T., Christoph, A., Winter, M., Nierstrasz, O., Ducasse, S., Wuyts, R., Arévalo, G., Schönhage, B., Müller, P., Stich, C., Oct. 2002. Components for embedded software: the PECOS approach. In: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02). Grenoble, France.

Göbel, S., Pohl, C., Röttger, S., Zschaler, S., Mar. 2004. The COMQUAD component model: enabling dynamic selection of implementations by weaving non-functional aspects. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04). ACM Press, Lancaster, UK.

Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M., Sep. 2004. SaveCCM - a component model for safety-critical real-time systems. In: Proceedings of the 30th EUROMICRO Conference (EUROMICRO '04). Rennes, France.

Heiser, G., Dec. 2005. Secure embedded systems need microkernels. USENIX ;login: 30 (6), 9–13.

Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., Liedtke, J., Jul. 1998. The Mungi single-address-space operating system. Software: Practice and Experience 28 (9), 901–928.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K., Nov. 2000. System architecture directions for network sensors. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000). Cambridge, UK.

Krishna, A. S., Wang, N., Natarajan, B., Gokhale, A., Schmidt, D. C., Thaker, G., Mar. 2005. CCMPerf: A benchmarking tool for CORBA component model implementations. Real-Time Systems 29 (2–3), 281–308.

L4 Community, 2005. The L4 headquarters. http://l4hq.org.

Liedtke, J., Sep. 1996. Towards real microkernels. Communications of the ACM 39 (9), 70–77.

Möller, A., Fröberg, J., Nolin, M., May 2004. Industrial requirements on component technologies for embedded systems. In: Proceedings of the International Symposium on Component-based Software Engineering (CBSE7). Edinburgh, Scotland.

Object Management Group, Mar. 2004. Common object request broker architecture (CORBA/IIOP). OMG Specifi cation.

Schmidt, H., Mar. 2003. Trustworthy components: compositionality and prediction. Journal of Systems and Software 65 (3), 215–225.

Shaw, M., Nov 2005. Sparking research ideas from the friction between doctrine and reality. Stevens Award Lecture.

Tivoli, M., Fredriksson, J., Crnkovic, I., July 2005. A component-based approach for supporting functional and non-functional analysis in control loop design. In:

Tenth International Workshop on Component-Oriented Programming. Glasgow, Scotland.
URL http://www.mrtc.mdh.se/index.phtml?choice=publications&id=0927
van Ommering, R., van der Linden, F., Kramer, J., Magee, J., Mar. 2000. The Koala component model for consumer electronics software. Computer 33 (3), 78–85.
Wallnau, K. C., 2003. Volume III: A technology for predictable assembly from certifi able components. Tech. Rep. CMU/SEI-2003-TR-009, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.
Wang, N., Schmidt, D. C., Kircher, M., , Parameswaran, K., 2001. Adaptive and reflective middleware for QoS-enabled CCM applications. IEEE Distributed Systems Online 2 (5).