**CAmkES**

**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 componentization of system services, component-based software engineering is a particularly attractive approach to developing microkernel-based systems.

**Introduction**

Traditional methods for developing embedded systems are resulting in increasingly unreliable embedded software. 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. Organizing 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.

CBSE also enables independent development of components which means that specialized expertise can be (independently) concentrated on different parts of the system as required.

Component Architecture

In this section, layered component architecture called, CAmkES, will be presented. The purpose of the architecture is to provide support for developing embedded systems on top of micro-kernels.

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.

The relationship between our component architecture and the underlying microkernel based operating system is tight integration with the operating system results in two requirements.

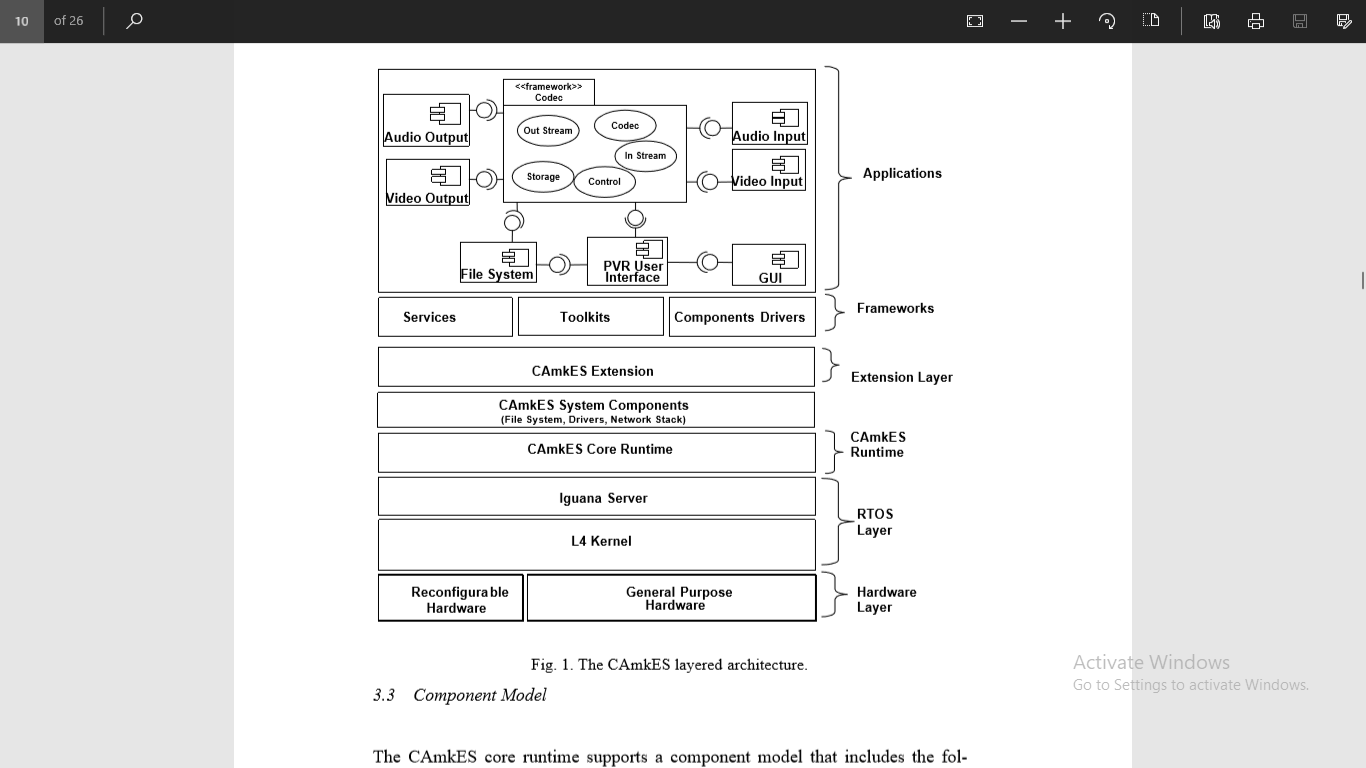
**First**, the architecture must directly make use of any mechanism provided by the OS (this includes inter-process communication, memory management and protection) and not re-implement similar mechanisms.

**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.

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 initialization (i.e., boot) time and that connections between components are established at design time and cannot be created or modified dynamically at runtime. This allows us to minimize 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.

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 organize operations and data into interfaces that have well defined 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 **data port** interfaces. An interface is defined by a CAmkES-specific interface definition language (IDL), which is based on the CORBA IDL.

RPC interface: An RPC interface defines synchronous communication between components by remote procedure calls.

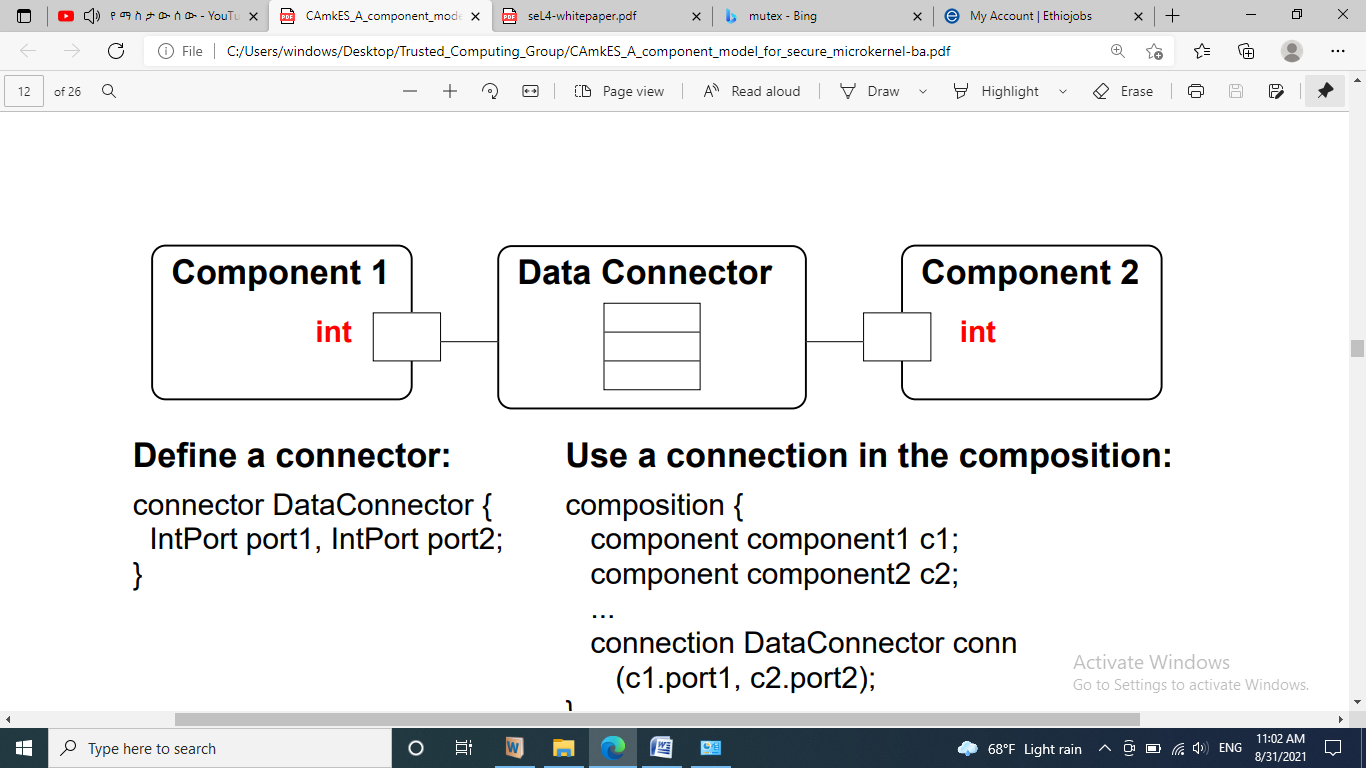
Event: CAmkES supports a publish/subscribe event model. Events are used for asynchronous notifications between components and they are emitted or consumed by components at event interfaces.

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.

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.

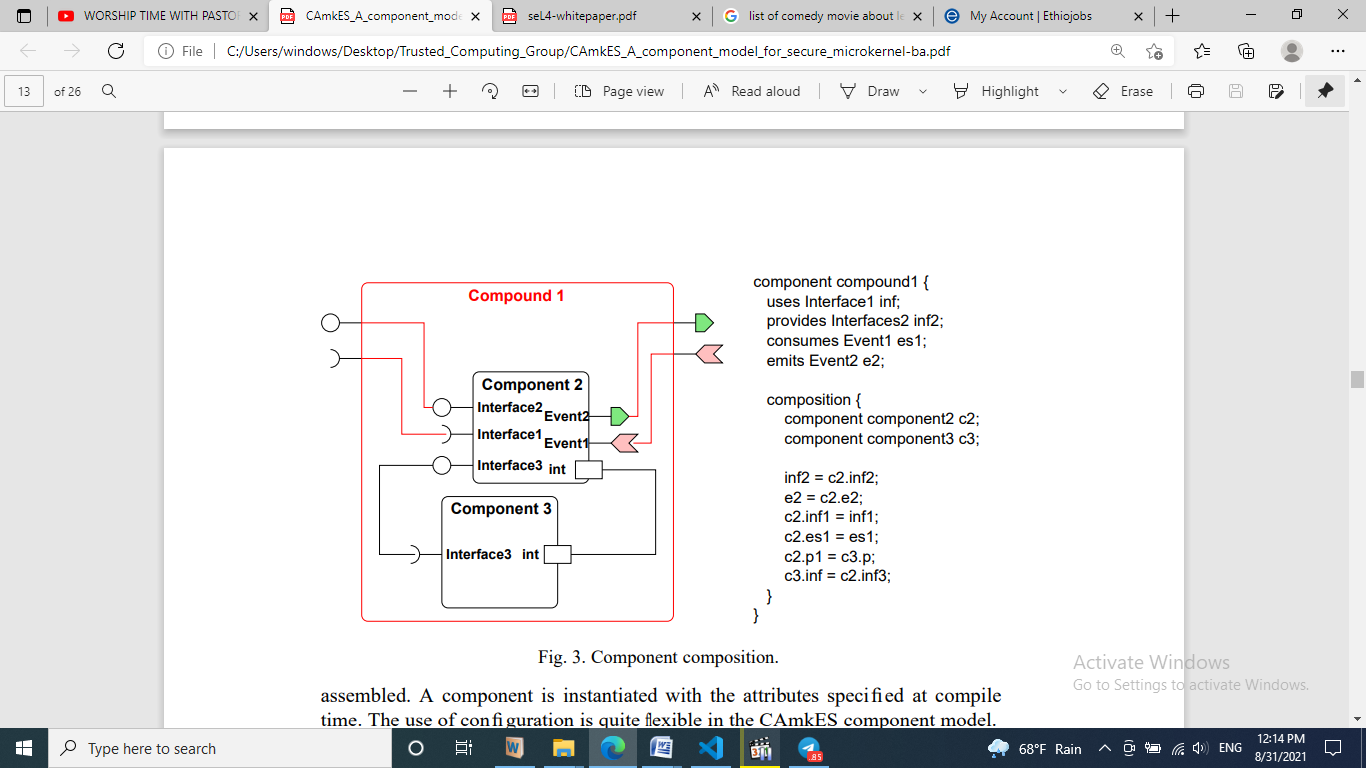
In our model, a connector has a name and a list of interface types that it connects. For example, a connector connecting a pair of dataports describes a data sharing relationship between them. A connector can describe 1- to-1, 1-to-many, many-to-1 and many-to-many relationships among interfaces. A connection is the instance of a connector.



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 assembled. A component is instantiated with the attributes specified at compile time.

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 embeddedapplication 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.

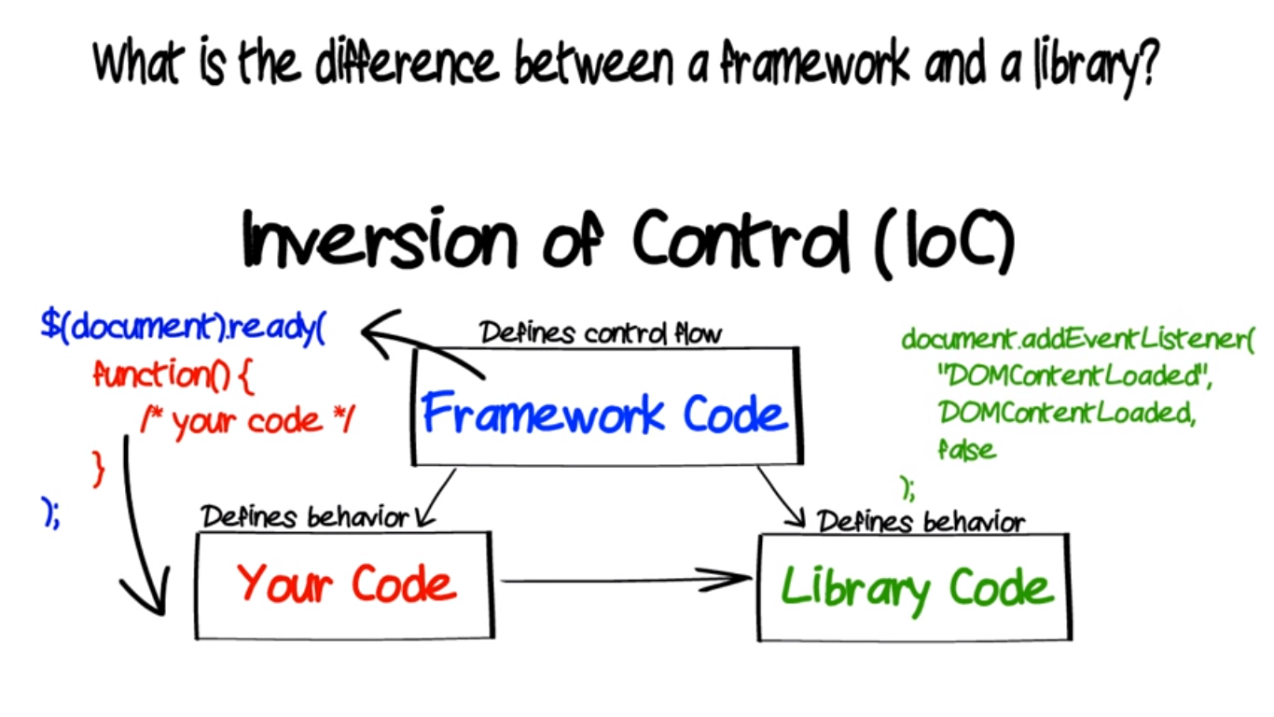
Note:

**Runtime Systems**: - A runtime system is a framework that typically monitors and orchestrates execution. There are many different types of runtime systems. Some runtime systems manage code to improve performance dynamically while the program is running; others monitor the code to understand what the code is doing or how it is executing on an underlying system. Of course, there are even language level runtimes which dynamically translate high-level program syntax on the fly into executable binary code, and examples of such systems include JS and Py runtimes.

A runtime is merely a platform for executing commands. Popular runtimes like JRE’s (Java Runtime Environment) and Node.js (a JavaScript runtime environment) process code and produce results.

**A framework** focuses more on best practices, i.e. the *way* you should write code. Frameworks are typically opinionated, designed to produce predictable outcomes. For example, the Express framework is an opinionated way to create a web server that runs on the Node.js runtime. Without this framework, it is still possible to create a web server from scratch upon Node.js….. But it can be time consuming and requires you to know all of the nuances of how a web server works. Frameworks are often used to fill knowledge gaps or simply make developers more efficient by providing reliable/predictable outcomes.

To use a construction analogy, a runtime is like the utilities built into the building, while a framework is more like the scaffolding. Frameworks are a construction idea, but runtimes are at work for the life of the building.



**IGUANA** is also an operating system (OS) personality that provides a set of services for memory management and process protection. Iguana is designed as a base for the provision of operating system services for embedded systems.

**Component-based software engineering**

CBSE is a process that focuses on the design and development of computer-based systems with the use of reusable software components.

In this type of development there is no concept of developing the software from scratch.

Components communicate with each other by their well defined interface.

**Abbreviation**

CAmkES (component architecture for microkernel-based embedded systems)

CBSE: - Component-based software engineering.

IDL: - An interface description language or **interface definition language** (IDL), is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language.

**Dictionary**

Asynchronous: - Pertaining to a transmission technique that does not require a common clock between the communicating devices.

Composite: - A conceptual whole made up of complicated and related parts. Consisting of separate interconnected parts

Constituent: - An artifact that is one of the individual parts of which a composite entity is made up; especially a part that can be separated from or attached to a system

Semantics: - The study of language meaning

Synchronous: - Occurring or existing at the same time or having the same period or phase.