Component Architecture and Modeling for Microkernel-based Embedded System Development

Yan Liu and Ted Wong
National ICT Australia Ltd, Australia
{jenny.liu; ted.wong}@nicta.com.au

Abstract

Microkernel-based approach provides operating system support for developing embedded systems with high performance and safety through memory protection. This allows us to introduce architectural mechanisms that enable good separation of concerns, while still satisfy the performance, security and reliability needs of embedded systems. Organizing embedded software as interacting components with well-defined interfaces is compatible with modeling methods. The challenge issue is to seamlessly transform models and integrate tools at different levels of design, implementation and deployment. In this paper we present our solution to this issue. Our contribution is twofold: first, we device a unified model driven and component based development approach. We adopt a layered architecture to construct a tool chain, which allows flexible extension at different layers. Second, we develop a software tool suite in order to support and demonstrate our solution, which includes a UML-based modeling environment and a set of component development tools on top of a microkernel operating system.

1. Introduction

Embedded systems are key components for ensuring safety and efficiency in critical infrastructures such as defense, health care, transportation, and water. Devices become increasingly smaller in physical terms but larger in software terms. As such, software is often a fundamental part of a device’s functionality. This increase in software complexity has had a significant business and cost effect on the development process of embedded systems. Economically, embedded systems need to have flexible development methods with tool supports to reduce time-to-market and to improve maintenance.

Embedded systems have a number of unique properties and requirements that have a direct impact on the methods and tools used for their software development. Embedded systems are very domain specific and as such are highly heterogeneous in terms of the hardware and software components used. For example, the embedded software for automotive systems has different features and requirements to that for mobile applications. There is no simple notation of a platform as it is in the enterprise system development such as J2EE or .Net. In addition, embedded systems are often real-time systems, which mean they have stricter requirements for many properties than enterprise systems do, including timeliness, liveness, safety, reliability and low resource consumption, such as power and memory.

These properties and requirements have greatly increased the difficulties in developing embedded systems, and in particular of those activities related to system design, verification and maintenance. There is an increasing demand on unified software engineering methods, tools and techniques for embedded systems development [1][5].

A combination of the model driven approach and the component architecture can deliver a promising solution towards rapid assembly of high performance and safety critical embedded systems [1]. However, new challenges are raised by the attempt to integrate MDD and CBSD for embedded systems development, including concepts and platforms modeling, model transformation and integration with existing software [9].

In this paper, we focus on the issue of automating the transformation from design models to programmable and deployable component modules of embedded software, especially on integrating the model driven development process with the existing software tools and platform.

In order to achieve this goal, we adopt a layered architecture to construct a complete tool chain that seamlessly transforms models and integrating tools at different levels of design, implementation and deployment. This allows flexible extensions to models and tools at each level. Our contribution is twofold: first we device a unified model driven and component based development approach. Second, we develop a software tool suite associated with our approach, which includes a UML-based modeling environment and a set of component development tools on top of a microkernel operating system.

The structure of the paper is as follows: section 2 introduces the related technologies upon which our solution is built. Section 3 discusses the related work. Section 4 discusses the component architecture and tools, which is the base for developing the modeling constructs. Section 5 presents the model transformation from design models to other software artefacts, including model driven generation techniques and tool implementation. Section 6
describes the case study of developing a PCI bus driver. Section 7 concludes the paper with future work.

2. Background
We explore several enabling software engineering technologies and investigate their synergies in this section. We identify their contributions towards the rapid assembly of embedded systems. This motivates our research to combine these technologies and develop a unified solution.

2.1. Microkernel OS Components
Microkernel OS has been proposed as a software platform to support the computing of embedded systems with limited resource and strict constraints for safety and reliability [6]. In NICTA, a technology stack has been developed with several tools for a microkernel-based embedded systems development. A detailed review of the research and tools are described in [8]. We provide a brief overview of the software most relevant to the work presented in this paper, including L4 microkernel and Iguana operating system.

L4 Microkernel. L4 is a microkernel that provides a minimal set of most important operating system functionalities, including memory protection and inter-process communication. The L4 kernel supports the construction of a small trusted computing base (TCB) [6]. In security terms, a TCB is part of a system that can bypass security policies and therefore must be fully trusted. In another word, the microkernel is the only code that runs in a processor’s privileged mode. Other code runs in the processor’s unprivileged user-mode, and is protected by the microkernel from direct interference by unrelated code. In this way, L4 provides good partitioning between different applications and OS services. This means buggy or malicious code can be spotted and isolated to prevent the whole system being affected.

The minimality of a microkernel means that it does not use excessive resources. Furthermore the small size of kernel code makes it possible to fully formally verify the microkernel. This capability is delivered by two other NICTA research projects, sel4 and L4.verified [8].

Iguana OS. A minimal operating system, called Iguana, is developed on top of L4 specifically for the use in embedded systems. Iguana provides essential services, such as memory management, naming, and supports for device drivers. These cover basic services required by many embedded applications.

Microkernel-based approach provides the operating system support for developing embedded systems with high performance and safety through memory protection. This allows us to introduce architectural mechanisms that enable good separation of concerns, while still satisfy the performance, security and reliability needs of embedded systems.

2.2. Enabling Software Engineering Technologies
Component-based software development (CBSD) addresses the complexity and flexibility requirements of embedded systems by composing systems from independent, well-defined building blocks [5]. It brings obvious benefits for embedded systems such as reusability, maintainability, and a reduction of software complexity, which lead to improved productivity.

There is a strong synergy between the approach of the microkernel-based system design and the CBSD: on one hand, the CBSD approach to assembling 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 those features such as protection and communication needed to build secure and reliable systems consisting of interacting components.

Component-based embedded software development necessitates the abstraction of the system requirements, feature properties, functionalities and qualities. It demands on mapping the abstraction as the component definition to the implementation in programmable platforms. This is the essence of model driven software development (MDD).

MDD is a software engineering approach that aims to reduce the complexity and improve the quality of software systems by constructing software systems through models and associated modelling technologies. Models provide abstractions of physical systems that allow engineers to reason about their systems by ignoring extraneous details while focusing on relevant ones.

A synergistic engineering approach that combines the CBSD with the MDD can help to deal with heterogeneity of embedded systems at different levels. Platform independent design models can be transformed into platform specific implementation with tool supports. This helps to reduce programming errors and bugs, and even optimize the implementation directly in the generated code.

Based on above discussion, a combination of the model driven approach and the component architecture on the microkernel operating systems can deliver a promising solution towards rapid assembly of high performance and safety critical embedded systems. One of the challenges is to preserve the semantic meaning of the component architecture in the design models. This means design models must conform to the component architecture to demonstrate sufficient accuracy in capturing the system’s behaviour.

3. Related Work
The adoption of platform-independent design models has been widely investigated in different application domains [3][15][16][17]. For example, Model-Integrated Computing (MIC) [17] targets CORBA middleware-based real-time and distributed systems. It provides a domain-specific modeling environment to facilitate
system analysis of the models, and to automatically synthesize applications from the models. This development approach has heavy weight at the level of component model and the operating system, which make it less suitable for developing resource constraint embedded systems. Our approach, instead, is more related to high performance and microkernel-based embedded systems. Our modeling environment is tailored to our dedicated lightweight component architecture for embedded systems, called CAmkES [10]. However, our approach is currently lack of the support of non-functional properties modeling and analysis. This remains our ongoing research work. Our previous work in [11] discusses how to introduce new features into the component model. We believe that our approach is extensible to address this issue.

Conmy et al. [4] discuss the issues needed to be overcome when MDA approach is used to certify a safety critical design. The paper concludes that architectural components and properties essential to a given set of safety requirements must be described in the meta-models. Design models must conform to meta-models to demonstrate sufficient accuracy in capturing the system’s functionality and behaviour. Finally transformation programs run on the models should enforce and ensure that safety requirements are met when they manipulate or check models. Our research approach is inline with these findings that the structure of our tools supports extensions to address safety properties as described in [4].

There are many research efforts on model-driven development for embedded systems [1][2][12][17]. One reason for there are so many model driven environments and tools is that embedded systems are very domain specific and diverse in both hardware and software. Each tool and method can only address issues of particular application domains [1]. It remains a grand challenge to consolidate methods and tools, and to establish reusable knowledge bases across different types of embedded systems.

France et al present a thorough review of research challenges in model-driven development of complex software systems [6]. Hovsepyan et al further discuss that new challenges are raised by the attempt to integrate MDD and CBSD for embedded systems development, including concepts and platforms modeling, model transformation and integration with existing software [9]. They all confirm that despite the research challenges, model-driven development, if realized, can conceivably result in significant improvement of productivity and quality in software (including embedded software) development. In this paper, we focus on the integration of MDD concepts and processes with the existing software tools.

4. CAmkES Architecture and Tools

In NICTA, we have developed a component architecture, called CAmkES on top of microkernel-based operating systems (L4 and Iguana introduced in section 2.1). This component architecture defines a platform independent component model. This component model can be further realized by specific programming languages (such as C and C++) and can be deployed onto a specific platform such as L4/Iguana. Our previous paper in [6] only describes the design and rationale of CAmkES architecture and the component model. The paper published in [7] reviews the synergies of CAmkES and other NICTA Embedded System research work. In this section, we briefly present the CAmkES architecture, followed by the descriptions of the CAmkES development tools. This architecture setup the base for our solution to integrate the model driven development process with the existing CAmkES tools.

4.1. Components and Connectors

A CAmkES component is the basic unit of organizing behaviour, encapsulating operations and data into interfaces. CAmkES supports three types of interfaces, namely remote procedure call (RPC), event and dataport interfaces. An interface is defined by the CAmkES-specific interface definition language (IDL), which is based on the CORBA IDL. Details of these interface type definition can be found in [11].

CAmkES component model encapsulates communication between components in explicit architectural elements called connectors and connections. In CAmkES, there are three kinds of connectors (connections), namely RPC connector (RPC connection), Event connector (Event connection) and Dataport connector (Dataport connection). Connectors are explicitly defined and implemented by system developers, rather than being implicit in the architecture as they are in most component systems. Thus, when designing a system, the developer specifies the components, the connections between components, and the specific connectors used for these connections. A simple HelloWorld example is shown in Figure 1. This example illustrates the usage of the connector and the connection. As it is shown, IguanaRPC is a default RPC connector provided by the CAmkES library to realize the connection hello between the compound component helloworld and the component helloclient.

The CAmkES library provides a set of default connectors for L4/Iguana. However, this does not prevent other types of connectors being used. New user defined or customized connectors can be added without modifying the core CAmkES architecture. This is the key feature of the CAmkES component model that helps to achieve the separation of concern: component functionalities and interactions remain cleanly separated. Moreover, the developer has full control over the mechanisms used for
interaction between components. This connector architecture not only provides the design flexibility, but also provides flexibility in making tradeoffs between performance and other desired properties from different connectors. For example, during the design and implementation of a network access router based on CAmkES and L4/Iguana, we have taken the advantage of connectors to experiment with safety vs. performance tradeoffs [17]. We have implemented several connectors ranging from those that provided complete memory protection between components to no protection between components. These connectors were used to produce versions of the router that represented the different tradeoffs with regards to protection and performance.

4.2. Code Generation Tools

CAmkES includes a set of software tools to facilitate the CBSD. We discuss the tool support in the context of the development process shown in Figure 3.

At the design level, interfaces are defined by IDL while components and connectors are specified using CAmkES Architecture Description Language (ADL). CAmkES component model is platform independent. It is the implementation of components, connectors and connections that are specific to the programming language. This is in contrast to component models such as EJB or COM+, in which the component model is the same as the programming model.

At the development level connectors and connections encapsulate the interactions between components that reply on the mechanisms of the underlying OS. In L4/Iguana, all CAmkES components are run as L4/Iguana servers. Each component is associated with a thread, which controls the communication protocol. This kind of communication and control code is generated and encapsulated in separate stub files for both components. CAmkES code generation follows traditional compiler-based approach: an ADL compiler, called Magpie, takes inputs of the component specification in IDL/ADL files, parses each component into a token stream, and builds a custom abstract syntax tree (AST). For each type of the connector, there is a template for the code generation. Templates are tailored to the AST to produce the target code for individual connectors. Figure 2 shows the sample outputs for a RPC connector and an Event connector. Customized connectors can be used to generate stub code the same way as CAmkES default connectors.

At the deployment level, a build tool takes inputs of the generated code as well as the generated stubs. This build tool then produces outputs as loadable images for L4/Iguana. At the runtime, images are loaded and components are initialized. Connections between components are created using the stub code generated.

A performance evaluation has been conducted by a set of benchmark operations. Details of this evaluation are described in [11]. The results compare CAmkES component architecture with L4/Iguana based solutions. The results demonstrate that the CAmkES component-based implementation has comparable performance to the implementation using L4/Iguana libraries. This means that the CAmkES architecture is flexible, has low overhead and allows software to take full advantage of the benefits that a finely-tuned high-performance microkernel provides.
5. **Modeling in CAmkES**

The CAmkES architecture has a clear separation of infrastructure and application modules. CAmkES gains the ability to easily compose applications of modules (through composition of components). This modularity naturally suggests the adoption of MDD to automate the development process and leverage existing CAmkES code generation tools. The aim is to automatically transform design models to reusable and composable CAmkES artefacts (such as software code).

Developing the modeling capability of CAmkES requires the following aspects:

- meta models that captures the semantics of the CAmkES component model;
- graphic presentation of the models and visual operations to manipulate the models;
- model transformation from one format to another while preserving the same semantic meanings;
- flexible and extensible architecture to incorporate new features of the component model.

All of these requirements should be captured in a single and unified design and development environment in the context illustrated in Figure 3. We leverage the powerful Eclipse platform to develop such an environment.

5.1. **CAmkES Modeling Constructs**

The modeling starts with the definition of modeling constructs, which is required to capture the same semantic meaning of the component architecture itself. The CAmkES meta model defines the basic modeling constructs to capture the semantics of CAmkES component model. Based on the description in section 4.1, the CAmkES constructs include `Component`, `Port`, `Association`, `Connector` and `Interface`. The CAmkES Interfaces have two types, namely `Required Interface` and `Provided Interface`. For each construct element, the meta data defines the properties and the constraints of this element such as name, visibility, operations allowed, appearance and so on. The CAmkES modeling constructs are UML™ 2.0 compliant, and they use UML notations.

The meta model also defines the relations between modeling constructs. In the CAmkES architecture, components communicate using interfaces. A component exposes interfaces through ports. The type of a port is defined by the interface that the port is associated with. A port is linked to either a CAmkES required interface or a CAmkES provided interface. The link is defined by a CAmkES association. By this means, a port type is either a provided or required interface.

A pair of provided and required interfaces can only be connected if they are of the same type. This constraint is imposed on the connection between components by checking their port type. The connection type is defined by a connector. Instead of introducing new modeling elements for each type of the connectors, we define a UML profile for CAmkES connectors. A connection can be annotated by a stereotype, such as `<<RPC>>`, `<<Event>>` or `<<Dataport>>` to indicate its connector type. For a new user defined connector, it should have its own UML profile defined.

Composition of components is realized by connecting components through their provided or required interfaces. Components can be composited into a compound component, which can further require or provide interfaces to other components. For example, in Figure 1...
a compound component helloworld is composed of the helloworld component and the helloworld component. A provided interface from the helloworld component is now exported to the helloworld component as the provided interface of the compound component. This exportation of interfaces is also modelled as a connection of the two interfaces (the original interface and the exported interface) in the CAkMES meta model. In order to differentiate this exportation from other connection, a set of stereotypes are introduced to export interfaces, including <<DataportExport>>, <<RPCExport>> and <<EventExport>>. These stereotypes are encapsulated in the CAkMES connector.

5.2. CAkMES Modeling Environment

With the CAkMES meta model defined, our next step is to realize the constructs in a modeling environment. This is based on the Eclipse modeling framework (EMF) and the open source modeling tool Papyrus [13]. Papyrus implements UML 2 specification and OMG standards as UML profiles. Papyrus itself is implemented as an Eclipse plugin, which is Eclipse UML2 compatible. Eclipse UML2 includes a set of editors for viewing and editing UML models. It also provides the APIs to manipulate the UML models.

We extend the component diagram of Papyrus with CAkMES modeling constructs. The implementation utilizes the Eclipse UML2 components and APIs. Figure 4 shows an example of realizing the CAkMES connector constructs. The visual presentation of the construct in Eclipse modeling workbench is managed by the ComponentDiagramEditor component. ComponentDiagramEditor invokes ConnectorEditPart to map the operations performed on the visual notation to the actual instance of the connector. ConnectorFigure deals with the drawing of the construct. Similarly other CAkMES constructs can be realized in Eclipse.

6. Model Driven Generation

CAkMES follows the Model Driven Architecture (MDA) paradigm to automate the development process (shown in Figure 3), transforming platform independent design models to platform specific programmable and deployable software artefacts. MDA is a variation of MDD. Models in MDA are defined by UML diagrams. It is the responsibility of the model transformation that actually generates platform specific software code. A transformation can be either model-to-model or model-to-text. The model-to-model transformation means that the design models are transformed into other type of models through intermediate data. For example, UML design models for systems engineering can be transformed into CAD simulation models automatically [14]. A new OMG standard called Query/View/Transformation (QVT) is specially initialized for this purpose. At the current stage, CAkMES relies on the model-to-text transformation to generate code. More precisely, CAkMES utilizes template-based code generation techniques.

6.1. Template-based Code Generation

Template-based code generation (TCG) is a software technique that typically uses a template engine to transform a model into compilable code, given the format specified in a template. TCG has already been used by CAkMES tools at the compiler level. As discussed in section 4.2, the CAkMES ADL compiler generates stub code for each connector using templates defined in Python.

The task of code generation at the design level is to extract model data from the UML diagrams, manipulating the data and generating ADL files. Without MDA supports, components are manually defined and compiled in ADLs. We use Eclipse Java Emitter Templates (JET) engine to develop this code generation function. At the heard of the JET code generation resides a template. A
template is a mixture of static sections, which JET will reproduce unmodified in the generated output, and tags which controls model processing flow, formatting, and Java specific code generation. The sample code below shows how to use JET to produce the lines of importing IDL files in the head file.

```% for (int i = 0; i < model.getIDLInterfaces().size(); i++) {%>
/* Definition of IDL interface */
import "%model.getIDLInterfaces().get(i).getName()">idl*;
{%>
```

What the JET engine produces from the template is not the ADL code itself. Instead, it produces a CAMkES ADL generator, which is a Java class. Its methods have parameters which need to be populated by information extracted from the design models. Therefore, this ADL generator is invoked within the Eclipse workbench to generate ADL files. Figure 5 shows the model processing flow for ADL code generation. In the Eclipse environment, a code generation wizard is added to its tool bar, shown as label 6 in Figure 9. An action event is trigged when a user click this wizard. This event is captured by Eclipse ActionSet, which invokes ADLGenerationRunner to load the UML diagram files, parses each element and invokes ADLGenerator to produce ADL files for CAMkES components. At the end, there is a directory for each component containing its generated code. Similarly, IDL files can also be generated.

6.2. Complete Tool Chain
The generated IDL/ADL files capture the same semantic meanings as the design models in the UML diagrams. Now they can be seamlessly integrated with platform specific code generation tools as described in section 4.2 to produce OS specific component implementation. A completed CAMkES MDD tool chain is shown in Figure 6. This approach benefits the embedded system development in twofold:

**Extensibility.** First, new features of the component model can be introduced and the ADL/IDL code generation can be updated by adding new ADL/IDL templates. Second, the CAMkES component model is generic and not platform specific. It can be integrated with other real-time operating systems (RTOS) by replacing the OS specific templates for the component and connector code generation. In addition, the builder can be replaced by other OS specific deployment tools.

**Separation of concern.** MDD hides the complexity of OS specific programming models from the developers at the design level. The stub code generated is optimized already to improve the performance of inter-component communication. For example, if two components are in the same memory protection domain, direct calls rather than RPC calls are generated. This enables the developers to concentrate on design with reduced development effort.

![Complete Tool Chain Diagram](image)

Figure 6 CAMkES model driven generation: layered architecture

7. Case Study: PCI Bus and Device Driver
In order to illustrate the usage of our approach in the development of an embedded system, we discuss a relevant and simple case study, the development of a PCI bus driver as shown in Figure 7. Most systems that incorporate a PCI I/O bus use a driver that scans the bus at start up in order to determine what devices are attached. The resulting information about which devices are attached is used to find, instantiate, and initialise appropriate drivers.
This PCI bus scenario includes components for the bus driver as well as the required device drivers (e.g., an Ethernet driver), their clients (e.g., a network stack component) and connections between them. The bus driver component includes functionality to scan the bus and to invoke the connected device driver’s configuration interface in order to initialise it correctly.

At the design time, the system developer does not know what specific device will be connected to the PCI bus and so cannot include this in the system specification. It is up to the PCI bus driver to determine which device driver components to create and to ensure that the clients are connected to those drivers. In order to implement these scenarios, the extra components are added as shown in Table 1. These components will implement the interfaces shown in Table 2.

### Table 1 Extension Components

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory</td>
<td>does the actual work to create a new component instance</td>
</tr>
<tr>
<td>Registry</td>
<td>Maps a keyword string to an opaque piece of binary data</td>
</tr>
<tr>
<td>Binder</td>
<td>does the actual work to connect two component instances at given interfaces</td>
</tr>
</tbody>
</table>

### Table 2 Extension Interfaces

```
// IFactory interface:
component_instance_id create (component_id)

// IRegistry interface:
register (component_instance_id, keyword[])
lookup (component_instance_id)

// IBinder interface:
component_instance_id find_keyword (keyword[])
bind (component_instance_id, interface_id, component_instance_id, interface_id)
```

The conceptual component model is shown in Figure 8. These components are used to implement the scenarios as follows:

1. Factory finds loaded component code. The factory component is responsible for locating loaded component code given a component identifier. The result of such a lookup is platform specific and will be implemented internally by the factory component.

2. Creator invokes Factory to create a component instance. The factory component locates the appropriate loaded component code and creates a new instance (the details of this are platform dependent). It returns a component instance identifier, which is an opaque platform-specific data type.

3. Connector invokes Binder to connect two components.

4. Creator invokes Registry to register component with key words. A component invokes the register function of the registry component, providing the identifier of the component it wishes to register along with relevant keywords.

5. Client invokes Registry to find component

6. Search the registry with keywords. A component invokes the `find_keyword` method of the registry component, passing a keyword and receiving the identifiers of matching component instances.

### Design-time

Figure 9 demonstrates the resulting design models in UML using the CAmkES modeling extensions in Papyrus. Components and connectors of this architecture can be generated for L4/Iguana implementation by the tool chain described in Figure 6.

With regards to the performance of generated code from design models, we believe the generated code should achieve the same performance level as the code manually programmed. This is because they both rely on the same CAmkES compiler to optimize the communication code in a stub file. Previous results from benchmarking measurements have demonstrated that the performance of CA.mkES implementation is comparable to the
performance of the implementation using L4/Iguana libraries.

This example demonstrates that the CAmkES modeling capability can help with functional design of embedded systems. It remains our future work to extend the CAmkES component model to support specification, modeling and validation of non-functional properties such as timeliness and power consumption.

8. Conclusion

This paper describes our research on devising the model driven architecture based on the component model and the microkernel-based operating system. We focus on the rationale of constructing tools that help to transform design models into software code. The resulting solution includes a UML-based modeling environment in Eclipse and a set of microkernel-based component development tools. Applying component-based and model driven development in the embedded system domain also introduces several new concepts and challenges. It remains our future work to address the following issues:

- Must fulfil the real-time requirements present in many embedded systems;
- Should provide flexibility for component design and implementation. For example, it should allow components to be parameterised and configured, it should allow various ways of connecting components together, and it should allow components to make as few assumptions as possible about their environment;
- Should support a variety of computational models (or architectural styles), including event-based models, dataflow models, etc.
- Require testing and validation support in the design phase of embedded systems.

It is also intriguing to further validate if our approach is scalable and practical for more complex embedded systems development.

9. Acknowledgement

National ICT Australia is funded through the Australia Government’s Backing Australia’s Ability initiative, in part through the Australian Research Council.

10. References

Using SysML—Part 2: Celebrating Diversity by Example. INCOSE Intl. Symposium, San Diego


1. CAmkES constructs  2. Exposing interfaces through ports  3. Connections between interfaces
4. Adding and applying CAmkES profile  5. CAmkES design model hierarchy  6. Code generation plug-in

Figure 9 CAmkES modeling environment