Extending the Capabilities of Component Models for Embedded Systems

Ihor Kuz and Yan Liu

National ICT Australia
ihor.kuz@nicta.com.au; jenny.liu@nicta.com.au

Abstract. Component-based development helps to improve the modularity and reusability of embedded systems. Component models devised for embedded systems are typically restricted due to the limited computing, storage and power resources of the target systems. Most existing component models for embedded systems therefore only support a static component architecture and provide a simple and lightweight core. With the increasing demand for more feature-rich embedded systems these component architectures must be extended. In order to remain useful for the development of resource-restricted embedded systems, however, the extensions must be optional. Creating such extensions requires a cost-effective development process that can produce reusable, rather than application-specific, extensions. This necessitates a systematic approach to seamlessly integrate application specific requirements of the extension, the existing component model and the constraints of the computing environment. In this paper we propose a scenario-based architectural approach to extending the capabilities of the CAmkES component model. This approach is used to distil application specific requirements and computing constraints, summarise generic scenarios, drive the extension to the core CAmkES architecture. We illustrate our approach with a case study involving the addition of dynamic capabilities to CAmkES.

Keywords: embedded system, component, extension, scenario, architecture design

1 Introduction

Component-based development helps to improve the modularity and reusability of software and is increasingly being applied to embedded and real-time systems. Component architectures for embedded systems have major differences from those for enterprise systems mainly due to the resource restrictions of embedded systems. Deployment, cost, and size concerns lead to significant restrictions in processing power, memory size, and energy resources. Developers of software for embedded systems must, therefore, ensure that their software can perform sufficiently on slower processors, can fit into reduced memory, and can run efficiently in order to conserve energy. Since component-based implementation of applications often demands extra computing resources, component architectures and models for embedded systems are

devised to be simple and lightweight. They typically allow only static architectures that do no change at runtime, limit the ways that components can be connected and do not provide for memory protection between components. This means that, for example, components cannot be created or destroyed at runtime, nor can new connections be created or existing connections be broken.

With increasing demand for more feature-rich embedded systems (including mobile phones, cars, multi-media systems, etc.) that require dynamic features such as downloading and updating of system software, dynamically changing configurations, etc. component models for embedded systems need to be extensible and flexible in order to support such features when they are required. The domain-specific nature of embedded systems means that features required by different types of devices (for example, mobile phones and cars) will be very different. This makes it impractical to devise a comprehensive component model that provides all the possible features.

One solution is to develop a component model that supports core features and also embeds services for developing extra features in a monolithic design. This is similar to the component models implemented for enterprise applications, such as J2EE and CORBA Component Model (CCM). In these approaches a container hosts the application components and also provides services such as transaction and security management. Such a solution leads to a heavy component model and has the disadvantage that restricted resources can prevent its practical use.

Another solution is to extend the static component models with only required features and develop a flexible architecture for incorporating newly added features. This solution imposes challenges for the design of the component architecture at two levels: first, the core model of the component architecture needs to be extended to incorporate new application specific requirements; second, in order to achieve cost-effective development, extensions to the core component model should be reusable by other applications with similar requirements. Furthermore, any resource restrictions that apply to the design and implementation of the component architecture will also apply to extensions.

This necessitates a systematic approach to the development of feature extensions for embedded systems that seamlessly integrates application specific requirements that originally led to the need for extensions, the existing component model, and the constraints of the computing environment. In this paper we propose a scenario-based approach to extending the capabilities of our CAmkES component model. Our approach distils the requirements of the target embedded application and integrates them with CAmkES components and relevant architectural patterns. We demonstrate our approach with an illustrative case study that involves adding dynamic capabilities to CAmkES.

2 Overview of CAmkES

CAmkES (Component Architecture for microkernel-based Embedded Systems) is an architecture that we have developed to enable the component-based development of embedded systems. Specifically, CAmkES targets embedded systems based on the L4 microkernel [10]. Since microkernels are light on resource requirements, provide

good protection between applications and OS components, and make for a suitably small trusted computing base, they are a highly suitable base upon which to build reliable and trustworthy embedded systems [6]. In this paper we provide only a very brief introduction to the CAmkES architecture and development process, more details about the architecture can be found in [8].

Similar to standard component architectures, the CAmkES architecture provides components, interfaces, and connectors. However, CAmkES is targeted at potentially resource restricted embedded systems, which leads to three key features of the architecture (and its associated component model).

First, connectors are first class concepts and can be defined by users whenever specific communication functionality is required. This means that CAmkES allows developers to place components in the same address space, place components in separate address spaces, or even place the components on separate machines. CAmkES also provides the flexibility to tailor communication between components to application-specific needs.

Second, the core CAmkES component model makes it possible to reduce the overhead introduced by component-based computing by limiting built-in features. In particular, it only allows for static architectures that do not change at runtime. This means that components cannot be created or destroyed at runtime, nor can new connections be created or existing connections be broken. Since not all applications will require this kind of dynamic capability they should not have to pay for it.

Finally, the core CAmkES component model and architecture can be extended by adding components that act as *extensions*. These components implement functionality that extends the architecture's capabilities. For example, in order to allow the creation and destruction of components at runtime appropriate extension components must be added.

With regards to the development process, at design time a system architect defines components and their interface in an architecture description language (ADL). The full system application is also specified in ADL, including specification of all the components and connectors involved, and all the connections between components. The component functionality is implemented separately in a regular programming language such as C.

The ADL files are compiled to produce loading and initialisation code, communication stubs, and any other required runtime support code, all of which is compiled together with the component code, and combined with the L4 kernel to produce a loadable system image. At system boot time the system image is loaded into memory, the kernel starts up and invokes the CAmkES loader to load all the components. The loader creates and initialises all components and their connections, and after everything is ready starts the system running.

3 The Approach

Initially the need for an extension is driven by application specific requirements, such as being able to dynamically create a device driver instance at runtime, or being able to update components on-the-fly without bringing the system down. If the component

model being used does not provide features required to do these things, the developer needs to extend the model with the appropriate capability. While a developer could work out application-specific solutions, such solutions are rarely reusable. As such, when later development requests for similar features arise, new implementations must be redeveloped. Moreover, the maintenance overhead also increases since the solutions are tangled within the different applications. Obviously this is time consuming and not a cost effective development process.

Rather than developing ad-hoc and non-reusable solutions, we envision a systematic approach that enables the development of generic extensions to a core component architecture. The extensions are implemented as components and services that support generic scenarios and are not specific to a particular application.

There is currently a gap in this process between the application-specific requirements and the generic extension components. We consider adopting a scenario-based software architecture analysis approach [13] to solve this problem. In scenario-based analysis of software architecture, scenarios (i.e., detailed requirement descriptions in specific contexts) are used as a tool to analyse quality attributes and the primary utilisation of these quality attributes. Kazman et al. use scenarios to express the particular instances of each quality attribute that are important to the customer of a system. In this paper the scenarios we refer to are consistent with the meaning as described in [13], which is *a brief description of an anticipated or desired use of a system*. The scenarios are usually described by a sentence, for example, "a PCI bus driver creates a device driver component instance at runtime."

Key application-specific scenarios can be derived from the application-specific requirements. By identifying these key application scenarios, we can better understand what is required of the component model and how components can be applied in the design and development of the application.

Our scenario-based approach to designing generalised and reusable extensions is summarised in Figure 1. On the right-hand side we show the desired application and its requirements, while the left-hand side shows the four main steps of our approach. These steps are outlined below.

The first step is to derive key scenarios that cover the application-specific requirements. Many techniques for developing scenarios have been devised in scenario-based software architecture evaluation methods [2][4]. These techniques can be applied here to derive these key scenarios. For example, scenario brainstorming can be exercised with the goal of identifying the type of activities that the system must support.

The second step involves generalising the key scenarios. The challenge here is to capture all the major components and connections that are involved in the key scenarios and redesign them to be generic. An example of a general scenario is that CAmkES is able to identify a component type, find loaded code for the given component type and create a component instance.

Normally the scenario-based software architecture evaluation method starts with general and preliminary scenarios that have been gathered, classified and prioritised. During walk-through meetings, the architecture is evaluated according to these scenarios. Detail is added to the general scenarios and they evolve into application and context-specific concrete key scenarios.

In our approach we start with application specific requirements and come up with concrete key scenarios. Since our aim is to develop reusable extensions these key scenarios are further generalised into scenarios covering the common requirements of extensions that cover a broad range of applications. This demands expertise from the software architects and engineers who are responsible for designing the extension scenario and making decisions with regards to the selected scenarios. A practical approach is to start with the key scenarios and gather feedback from possible stakeholders and other software architects, then to revise the key scenarios toward more generic scenarios.

The third step requires extension of the component model to support these general scenarios. An important requirement is that the extension be generic and reusable. One approach to extension is to change the core component model to incorporate the new requirements. This has potential maintenance and integration issues since the changes may cause compatibility issues with legacy code developed for the original core component model. It is also possible that the core component model implementation is not available, or it is impractical to make changes to it. A better approach then is that the implementation of these extensions should utilise existing components to the maximum extent through a flexible architecture. In the case of CAmkES, an extension to allow the dynamic creation of a component instance can be implemented as a factory component. This factory could be implemented as a static CAmkES component and loaded into the system memory from the boot image at boot time. A concrete example is illustrated later in Section 4.

Finally, the resulting generic solutions are used in the design and development of the specific application. At this stage, patterns as best practices can be applied as part of the application architecture. The context in which a pattern is applicable must match the context of the application scenarios and the component model must support an appropriate programming paradigm for implementing the pattern.

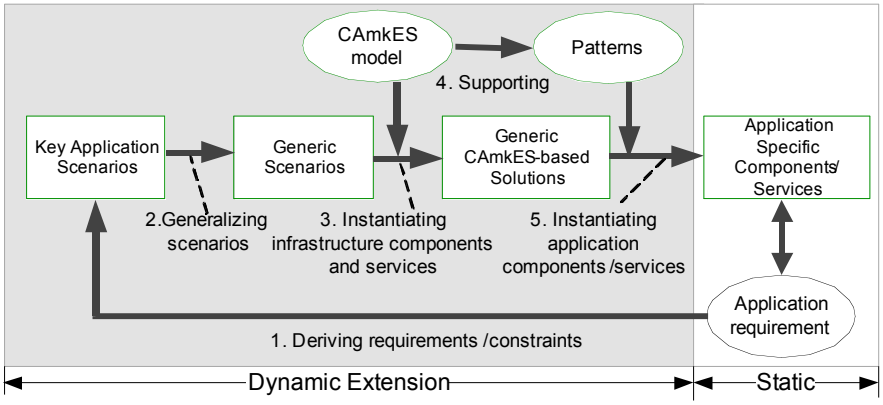

Figure 1 Scenario-driven component-based development approach

4 Case Study: Developing a PCI Bus Driver

In order to illustrate how our approach is used in the development of a real system we discuss a simple (but relevant) case study: the development of a PCI bus driver. 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. There are two ways to build and use such a PCI driver.

The first is a static approach in which the developer knows which devices will be attached to the bus and the PCI driver is simply used to provide initialisation information for the drivers of the attached devices. The second approach is a dynamic approach where the PCI driver scans the bus and the resulting information about which devices are attached is used to find, instantiate, and initialise appropriate drivers.

Implementing the static approach using CAmkES is straightforward and does not require functionality beyond that provided by the core model. The components implementing 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 are all specified at design time and are created at system initialisation time. 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. It does not need to create any components at run time since the driver component is already there and connected to it.

Implementing the dynamic approach, on the other hand, is not possible using only the core CAmkES functionality. In particular, at 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. As such, the device driver component cannot be created at system initialisation time nor is it possible to specify the connections to the device driver component at design time. 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 do this at runtime we must add appropriate extensions (in the form of extension components) to CAmkES.

In the following we use the approach described in this paper to design and implement generic CAmkES extensions that allow us to build a system taking the dynamic approach to PCI bus driver design.

4.1 Deriving Key Scenarios

To keep the case study feasible we specify further details about the actual scenario that we wish to implement. We assume that the possible devices connected to the bus are limited and known ahead of time so that driver code can be made available in the loaded image (this simply prevents us from having to include discussion of functionality to download the code or search for it on external storage). We also assume that the family of devices (e.g., Ethernet card, audio device, etc.) that we discover on the bus is fixed. This means that the client knows the family of device that it will connect to, and therefore the interfaces that it wants to connect to, and prevents us from having to introduce interface discovery into the example. Another

simplifying assumption is that there will be only one device attached to the bus (i.e., only one instance of the device driver component will need to be created). Finally, while the client does need to find the driver to connect to, there will be a predetermined way for the client to find the driver component.

Note that many of these further details comprise constraints on the system. Other constraints are provided by the application environment, and may relate to resource restrictions, temporal requirements, security requirements, etc.

These constraints together with the initial application requirements allow us to perform the first step and distil key application scenarios, which results in the following:

- Bus driver determines component required for a specific device.
- Bus driver finds loaded device driver component code.
- Bus driver creates device driver component instance.
- Bus driver connects to device driver component's configuration, interrupt, and IO interfaces.
- Bus driver initialises device driver component.
- Bus driver registers device driver component instance in a registry with a predetermined keyword.
- Client searches registry for desired device driver component by keyword.
- Client connects to device driver component interfaces.

The second step is to generalise these scenarios so that they are no longer application specific. We start by noting that we have at least two different types of components involved:

- The bus driver and client components are included in the system specification at design time and are created at system initialisation time.
- The device driver component, on the other hand, is created at runtime.

Likewise there are different types of connections involved, either created at system initialisation time or at runtime. There are more distinctions possible, and Table 1 provides an overview of the various aspects to consider when distinguishing between the types of components and connections.

Based on the different combinations of these aspects we can identify all the different types of components and connections. For further clarity we have given names to some of the more common combinations.

We call components that have property A1 and B1 *static components*. Components with A2 and B1 are called *dynamic components*, while those with A2 and B2 are called *dynamically loaded components*. We call connections with property C1 and D1 *static connections*, those with C2 and D2 are *dynamic connections* and those with C2 and D1 are *partially dynamic connections*.

Note that some combinations are not feasible, including: A1 and B2, C1 and D2, and A2 and C1 as they violate the logic of the component model.

Table 1 Component and Connection Category

A. How component instances are created
1. Statically created components are components that are created at system initialisation time. *2. Dynamically created components are components that are created at runtime (after all static components have been created and the system has been started).*
B. How component code is loaded into the system
1. Statically loaded components are loaded into the system memory at system boot time (from the boot image). *2. Dynamically loaded components are loaded into the system memory at runtime, typically from secondary storage (e.g., disk, RAM disk, flash disk, etc.) or network.*
C. When the connections are created
1. Statically created connections are created at system initialisation time. *2. Dynamically created connections are connections that are created at runtime (after all static components and connections have been created and the system has been started).*
D. When the connection type is defined
1. Statically defined connections have the connector types defined at design time in the ADL specification. Practically this means that communication stubs are compiled directly into the component. *2. Dynamically defined connections have the connector types defined at bind time. Practically this means that the stubs are not compiled into the component code but must be dynamically linked in some way (e.g., using dispatch tables).*

4.2 Generalising Scenarios

In this case study the bus driver and client components are static components, while the device driver is a dynamic component. While the connections between the static components and the dynamic component can be dynamic connections, for simplicity sake we will assume that they are all partially dynamic connections.

Generalising from the application-specific scenarios we arrive at the following. The components involved can be generalised based on their role:

- **Creator** (*static component*): requests that a new component be created. In the application-specific scenario this is the bus driver component.
- **Created** (*dynamic component*): a component created at runtime. In the application-specific scenario this is the device driver component.
- **Connecter** (*static component*): requests a connection to be created between the interfaces of two components. Both the bus driver and the client components play this role in the application-specific scenario.
- **Client** (*static component*): a component that is connected to the created component. In the application-scenario both the bus driver and the client components take on the client role.

Based on this, the generic scenarios we arrive at are:

- Find loaded code for a given component type.
- Create a component instance given a component type.
- Connect the corresponding interfaces of two component instances; this requires being able to identify component instances and interfaces.
- Register component instance at a registry with keywords.
- Search the registry with keywords.

4.3 Extending CAmkES

Next we design CAmkES extensions that allow us to implement these generic scenarios. Our main goal is to design and implement extensions in such a way that they are generic and reusable. It is also important to consider how the extensions can utilise the existing tools associated with the component architecture (e.g., the compiler and loader of CAmkES) in order to reduce the engineering effort and cost of development. Our solution to address these requirements is that the extension components will all be static components, and all connections to the extension components will be static as well. We end up with three extension components as shown in Table 2. These components will implement the interfaces shown in Table 3.

Table 2 Extension Components

Factory	does the actual work to create a new component instance
Registry	Maps a keyword string to an opaque piece of binary data
Binder	does the actual work to connect two component instances at given interfaces

Table 3 Extension Interfaces

```
// IFactory interface:
component_instance_id create (component_id)
// IRegistry interface:
register (component_instance_id, keyword[])
{component_id, keyword[]} lookup (component_instance_id)
component_instance_id[] find_keyword(keyword[])
// IBinder interface:
bind(component_instance_id, interface_id, component_instance_id, interface_id)
```

These extensions are used to implement the general scenarios as follows.

- 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 used internally by the factory component in the following scenario.

- The creator component invokes the factory component's create method. 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.
- The connecter invokes the bind method on the binder component providing component instance identifiers of the two target components and their corresponding interfaces.
- A component invokes the register function of the registry component, providing the identifier of the component it wishes to register along with relevant keywords.
- A component invokes the find_keyword method of the registry component, passing a keyword and receiving the identifiers of matching component instances.

We apply knowledge of design patterns in this step to come up with extensions that can be used as parts of appropriate patterns when possible. For example, the factory design pattern is applied to create new instances of components while the registry component adopts the service locator design pattern.

4.4 Developing Application Components

Finally, given the extensions and their use in the generic scenarios we can implement the desired application by defining the appropriate application components and having them use the extensions as required.

For the PCI scenario, the three application components are: the bus driver component (PCI bus driver), the client component (TCP/IP network stack), and the device driver component such as an Ethernet driver, which is to be created by the factory. The first two are static while the device driver component is dynamic. The design-time component architecture of the application is shown in Figure 2. It contains the two static application components and the three extension components with appropriate connections between them. The interfaces of the application components are shown for completeness, however, we do not discuss them further.

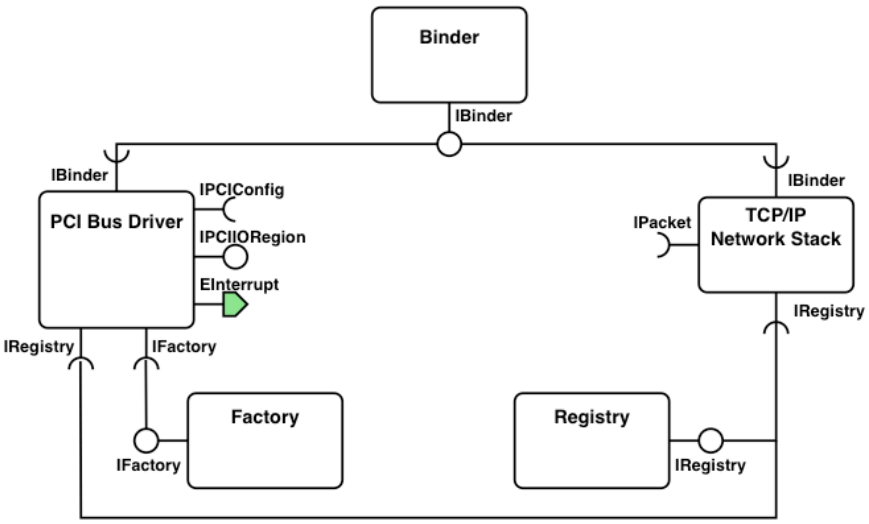

Figure 2 Design-time component architecture

At runtime the following steps are taken:

1. The bus driver scans the bus and maps the found device to an appropriate component identifier using internal (and driver-specific) data, such as PCI vendor ID and device ID fields in the PCI device configuration header.
2. The bus driver creates a device driver component by calling the factory with the appropriate component identifier.
3. The bus driver connects to the device driver component by invoking the binder component's bind method.
4. The bus driver registers the device driver component with the registry component by invoking the register function and using a preset keyword (e.g., "ethernet").
5. The client queries the registry component for the preset keyword until a positive result is returned (an alternative approach would involve the client registering with the registry to be notified when a match for the keyword is available).
6. Given the results from the registry the client connects to the device driver by invoking the binder component's bind method.

The component architecture after these steps is shown in Figure 3. Note that we now have one new component and several new connections.

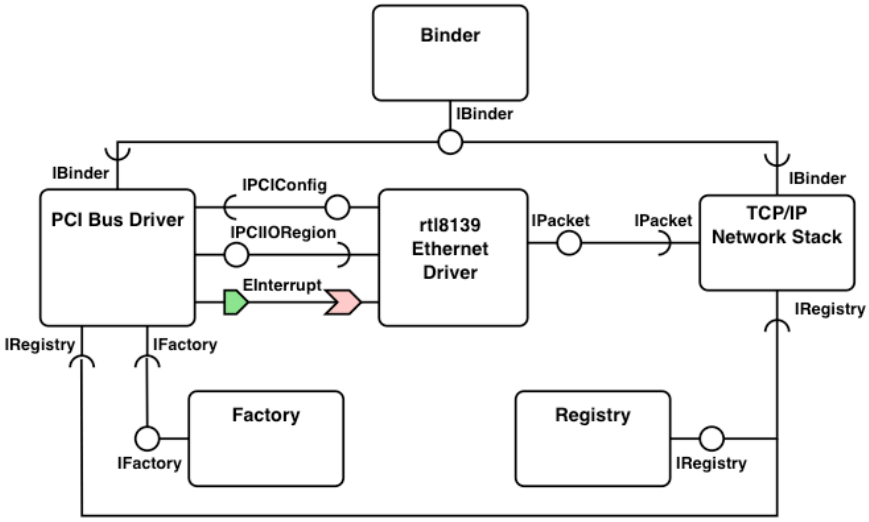

Figure 3 Run-time component architecture

To further develop the extensions, we can remove each of the scenario assumptions made earlier, thus broadening the scenario and changing the requirements. Doing so may affect some of the decisions made so far (in terms of what extensions to provide and the details of the interfaces) but will make the extensions more broadly applicable and therefore more general. In this way the design process can feed back into itself, allowing iterative feedback-based development of applications and generic CAmkES extensions.

4.5 Evaluation

We have followed this approach and implemented prototypes of the extension components as presented above. In our prototype, the extended dynamic functionality was fully contained in the static extension components. Future work will include the relaxation of several assumptions regarding the application scenario, and will lead to the development of further extensions such as those required for dynamically loading code and more flexible dynamic binding.

In terms of the overhead introduced by this implementation, it is mainly incurred by the static core CAmkES components. This is because the extension components for the dynamic features in this case are implemented as static components.

5 Applying the Approach to Non-functional Requirements of Embedded Systems

The previous case study illustrated the use of our scenario-based approach based on the analysis and generalisation of application-specific functional requirements. The approach is also suitable when the application-specific requirements are non-functional. For embedded systems important classes of non-functional requirements include temporal requirements relating to timeliness of execution and worst-case execution times, power management, reliability, etc. Following the approach as outlined earlier, we would start with application-specific scenarios that illustrate the non-functional requirements and remove the application-specific aspects to arrive at generalised scenarios relating to the requirements. We would then use these scenarios to design extensions to the component model that enable the implementation of such scenarios. The specific extension mechanism used may differ for these requirements than for functional ones, that is, we may need to make more changes than simply adding extension components. The details of extending the CAmkES model for non-functional requirements are future work.

6 Related Work

The basic characteristics of embedded systems, their requirements and constraints, and the implications to component models are summarised and presented in [9]. The challenging issues of devising an appropriate component model for embedded systems are recognised when component-based software engineering is adopted in developing embedded systems through experience. That is existing technologies for enterprise systems such as CCM, J2EE and .Net/COM+ cannot be used, or at least used directly for embedded systems [9]. The constraints of embedded systems are further articulated in [5]. This essence is also acknowledged in this paper. The approach we proposed in this paper is based on our existing CAmkES component model, which takes into account the constraints and requirements for embedded systems. We focus on the reusable extension of this core CAmkES component model. Our contribution is to demonstrate that this approach is feasible for developing the extension by reusing existing component models and making them general for different applications.

Andrews et al summarised the impact of embedded system evolution on real-time operating system use and design [3], in which how to map components onto application specific requirements remains a challenging problem. We consider dynamic updates as typical extension requirements for embedded system evolution. Research and engineering effort has been devoted to the field of dynamic updates. For a good overview, we refer to [12]. This paper focuses on a method to extend the capability of component-based design and implementation for embedded systems. We used dynamic updates as an example to illustrate our scenario-based approach to extending capabilities of CAmkES component model for embedded systems. The approach integrates application specific requirements to the development of generic and reusable extensions.

Another potential use of our approach is realising non-functional requirements through extensions to existing component models. Most component models do not address or at least have limitation in providing support for non-functional properties, such as timeliness, security, safety, reliability and fault tolerance. Ibrahim et al presented the ongoing research at Philips Semiconductors on improving productivity and reliability. It provides a literature survey of some techniques that address the issues of productivity and reliability [1]. In [7][16] solutions to address the timeliness and safety of embedded systems are proposed within their component model. In this paper we only briefly suggest that non-functional requirements can be implemented following our approach. It remains our future work on how to extend the component model to support non-functional requirements on quality attributes.

Scenario-based approaches are widely applied in software architecture evaluation. Methods and mechanisms are well established. For a good overview, we refer to [2][4]. In this paper we adopt a scenario-based approach to drive the design and implementation of extensions. The general scenarios constructed in this paper are consistent with the concept of a general scenario used in [13], which describes what achieving a quality attribute goal means. In [13], general scenarios describe how the architecture should respond to a certain stimulus. In this paper, a general scenario is abstracted and derived from key scenarios for quality attributes of interest. It is described to be generic and less application specific than the key scenarios. It remains our future work to apply this approach to more case studies and further validate its practical use.

7 Conclusion

Given the resource restrictions of embedded systems, component models specifically targeted for such systems are typically minimal, providing only limited functionality with which to build systems. In order to be widely useable and to fulfil demands imposed by modern embedded systems and their applications it is often necessary to extend the capabilities of such component models. In CAmkES, our component model for embedded systems, extending functionality is achieved by including extension components in a system's design. The need for extensions typically comes up when designing specific applications, however, the functionality provided by such extensions should be generalised such that the extensions can be reused to provide similar functionality for other applications.

We introduce a scenario-based approach to developing such generalised component model extensions. In this approach we start with an application's requirements and iterate through the steps of distilling key scenarios and then generalising these scenarios to make them independent of any specific application. The general scenarios provide requirements and context, which we use to guide development of the extensions. Finally, with the help of architectural patterns we use the extensions during the development of applications.

This approach has been applied during the development of CAmkES extensions that provide dynamic capabilities (e.g., creating and binding components at runtime). We describe a part of this experience in our case study section. Future work involves

investigating the application of this approach to create extensions that address non-functional application requirements, as well as other constraints imposed by an embedded system's environment.

8 Acknowledgement

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

9 References

[1] Ibrahim, A. E., Zhao, L., Kinghorn, J.: Embedded Systems Development: Quest for Productivity and Reliability. Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'06), (2006) 23 – 32.

[2] Ali Babar, M., Gorton, I.: Comparison of Scenario-Based Software Architecture Evaluation Methods. 11th Asia-Pacific Software Engineering Conference, (2004) 600 – 607.

[3] Andrews, D., Bate, I., Nolte, T., Otero Perez, C.M., Petters, S.M.: Impact of Embedded Systems Evolution on RTOS Use and Design. 1st International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'05), (2005).

[4] Dobrica, L., Niemela, E.: A Survey on Software Architecture Analysis Methods, IEEE Transactions on Software Engineering, 28(7), (2002).

[5] Hammer, D.K., Chaudron, M.R.V.: Component-based software engineering for resource-constraint systems: what are the needs? 6th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2001), Rome, Italy (2001) 91 – 94.

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

[7] Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M.: SaveCCM – a component model for safety-critical real-time systems. 30th EUROMICRO Conference (EUROMI-CRO'04). Rennes, France (2004).

[8] Kuz, I., Liu, Y., Gorton, I., Heiser, G.: CAmkES: a component model for secure microkernel-based embedded systems**,** Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems, 80(5), (2007) 687–699.

[9] Crnkovic, I.: Component-based approach for Embedded Systems, Ninth International Workshop on Component-Oriented Programming, (2004).

[10] Liedtke, J.: On μ-kernel construction. In 15th SOSP, Copper Mountain, CO, USA, (1995) 237–250.

[11] Bass, L., Klein, M., Moreno, G.: Applicability of General Scenarios to the Architecture Tradeoff Analysis Method, Technical Report, CMU/SEI-2001-TR-014, October 2001.

[12] Hicks, M.: Dynamic Software Updating. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, June, 2001.

[13] Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-Based Analysis of Software Architecture, IEEE Software November 1996, 47-55.

[14] Gheorghita, S.V., Basten, T., Corporaal, H.: An Overview of Application Scenario Usage in Streaming-Oriented Embedded System Design,, Eindhoven University of Technology, Report number: esr-2006-03, ISSN 1574-9517, 20 May 2006.
[15] Vandewoude, Y., Berbers, Y.: Run-time Evolution for Embedded Component-Oriented Systems. 18th IEEE International Conference on Software Maintenance (ICSM'02), (2002).
[16] van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer 33 (3), (2000) 78–85.