EMPRESS: Component Based Evolution for Embedded Systems (Position Paper)

J. Gorinsek, S. Van Baelen, Y. Berbers, K. de Vlaminck
KULeuven
Department of Computer Science,
Celestijnenlaan 200A, 3001 Leuven, Belgium
E-mail: {jorisg | stefanv | yolande | kdv}@cs.kuleuven.ac.be

May 8, 2002

Abstract

In this paper we present our work in the field of component based evolution for embedded systems. Our focus is on the necessary properties of the underlying system supporting evolution. We discuss the requirements of such a system, which address both the dynamic updating of components and the embedded nature of our target platform. Based on these requirements we propose a new approach to design such a system. Our approach relies on flexible component rewiring in combination with delegation for the dynamic updating of components and on the run-time monitoring of timing and memory contracts to guarantee memory and timing constraints.

Keywords: *component based development, embedded systems, software evolution*

1 Introduction

This paper presents the current status in the EMPRESS [1] project. EMPRESS, which stands for "Evolution Management and Process for Real-Time Embedded Software Systems" aims to take component-based embedded software development to a next level. Component-based embedded software development was the main research topic in the DESS [2] project. In EMPRESS we define a methodology, notation and support for software evolution in embedded systems. Our research in this project – the component architecture to support run-time evolution of components for embedded systems – is the main focus in this paper.

Why component-based development for embedded systems? Embedded systems provide excellent opportunities for software reuse. For example: cell phones are often released in product families where hardware and functionality is very similar. A cell phone with a color display instead of a monochrome one can easily reuse significant parts of the software from another product in the same family. Ideally only the display driver should be adapted to control a color LCD. Unfortunately, the typical characteristics of embedded systems usually lead software developers to low level design where reuse and maintainability are sacrificed for speed and efficiency.

Why evolution for embedded systems? Embedded systems are characterized by a long life cycle. Usually, these systems are deployed to be continuously online for several years. During this lifetime we want to be able to update the software of these systems without bringing the whole system to a halt.

In the next section we present requirements for the EMPRESS component architecture, with focus both on general requirements and special requirements for embedded systems. In section three we present a short overview of existing systems which

support evolution and identify concepts that are interesting to incorporate in our architecture. In section four we describe how we want to extend the results from DESS to support run-time evolution. The current status and future work is addressed in section five. We conclude in section six.

2 Requirements

In this section we present our requirements for the EMPRESS component architecture[1]. These requirements are twofold: we distinguish general requirements which address the dynamic updating of components (presented in subsection 2.1) and special requirements imposed by the embedded nature of our target platform (discussed in subsection 2.2).

2.1 General requirements

This subsection discusses five key requirements for the EMPRESS component architecture. These requirements describe the essential features a component architecture supporting evolution should provide according to us. These are: support for the dynamic updating of components, the updating of the component system itself, the updating of subcomponents and component collections, the evolution of component interfaces and state transfer.

Providing support for the *dynamic updating of components* is a major requirement for our component architecture. Embedded systems can benefit greatly from the ability to dynamically update the software they are running. Bug fixing, optimization, adding functionality and maintenance are only a few of the possible benefits.

It should also be possible to *update the running component system* itself. This allows us to provide maximum flexibility towards the management of the components. For example: suppose our component system is designed to run only one version of each component at he same time. A possible change would be to update the version management subsystem of our component system in order to run two different versions of a component in parallel to increase reliability.

[1] All terminology regarding components used in this paper is based on the definitions from DESS as described in [3].

We don't want to confine ourselves to the dynamic updating of single components. We should also be able to update *collections of components* as well as *single subcomponents*. This way we free the application developer from the burden of a fixed granularity of updates. The developer can perform radical changes by updating whole collections of – related – components at once and at the same time he can do the fine-tuning that is necessary when developing embedded software by updating only subcomponents.

Our component architecture must provide support for *interface evolution* in such a way that old components can use a new component with a different interface in a transparent way. Interface evolution also includes interface extension. In order to extend the functionality of a certain system it might be necessary to extend the interfaces of certain components in the application.

We should be able to *transfer the state* of an old component to a newer version. We want our updated components to pick up where the old version was in its execution when it was updated. This means that we need to be able to save the current state of a running component, change it to match the possibly different internal structure of the new version and start the new version with the transferred state.

2.2 Special requirements for embedded systems

In this subsection we discuss the requirements imposed by the he embedded nature of our target platform. The scarcity in memory and processing time implies that the updating of a component should be well timed, its duration should be predictable and the amount of memory used to update a component should be minimized.

Updating a component at run-time requires a substantial *amount of memory*: at least two versions of the component to be updated are required. However, we are working with embedded systems, so certain memory constraints will have

to be obeyed, even during an update. Therefore we require that the memory overhead for updating a part of the application is as small as possible and predictable. This implies for instance that we will not be able to keep all versions of a certain component in memory.

The duration of an update must be predictable and well timed. This is necessary to be able to guarantee the timing constraints which an embedded real-time system imposes. Since we are dealing with real-time systems, these constraints have to be guaranteed at any time during execution and thus also during dynamic updates. As a consequence, we need good heuristics in order to determine the best time to update a given component.

3 Overview of existing systems

Research in the field of software evolution has been going on for more than 25 years now. Over these years various approaches to the problem of updating software at run-time have been proposed. In this section we present a subset of these approaches which introduce valuable concepts for component evolution or techniques targeted at embedded and real-time systems.

PODUS [4], a system allowing incremental updates to procedures at run-time, presents two very usable concepts: mapper functions to transfer states between versions and interprocedures that deal with argument changes. The concept of mapper functions can be adapted to the transfer of component states across versions. Interprocedures will be extended to wrapper components which convert changing component interfaces. However, we must ensure that the computing and memory overhead of these mapper functions and interprocedures doesn't cause the embedded application to fail. This is accomplished by delaying an update until certain constraints are met, another feature of PODUS.

The dynamic updating system developed by Hicks [5] which allows the updating of code, data and types at run-time, and JDRUM [6], which extends the Java VM to allow class updates during execution, provide tool support for the auto generation of update code. In Hicks' system the skeleton for the dynamic patches is automatically generated. JDRUM too provides support to the programmer in creating conversion routines. These concepts can be used to provide tool support for the auto generation of wrapper components and state transfer functions using information provided by the component system.

Another interesting suggestion of Hicks is that the program to be updated has to provide an interface to initiate an update. At certain points during execution the application checks a queue of outstanding updates. This allows the application programmer to decide when to update. In our system the update is delayed until certain resource constraints are met. This implies that the component system itself must decide when to execute a scheduled update. Although Gupta proved in [7] that the determination of appropriate constraints for the moment of an update is undecidable in general, it is however possible to determine specific resource constraints which must be met to allow a safe update. To determine these constraints one must first determine an upper bound on memory and CPU usage of a given update before it can be executed.

The dynamic updating system for the Chorus operating system as it is presented in [8] is one of the few systems supporting the dynamic updating of real-time applications. It provides great flexibility because of its ability to transfer ports between processes on the fly. However to accomplish this it relies heavily on the functionality provided by the underlying operating system. Even though an object-oriented abstraction layer is presented, porting these concepts is very hard.

SEESCOA [9], which provides runtime support for component evolution, uses *timing contracts* to model and ensure timing constraints at component instance level and between component instances. This notion can be extended to memory contracts [10] which detail the memory usage of a component. In SEESCOA particular attention is paid to more complex contracts which support the evolution of the components to which they are

attached.

Kniesel proposes the mechanism of *dynamic delegation* in Lava [11] (a variation of Java) as an extension to the classic wrapper-based approach. The main benefit of delegation is that it allows interface change without suffering from the *self* problem [12]. Also since a childs reference to its parent can be dynamically changed we can use delegation to ensure that a certain message arrives at the proper version of a given component. It allows interface change and doesn't require the code to be written with evolution in mind.

All of the above systems provide us with concepts that are either useful to design a system supporting component evolution or that apply to embedded systems. In the next section we explain our approach to the problem.

4 Our Approach

In this section we extend results from previous research – such as those from the DESS project – and show how we use our findings from the previous section to design our system. We discuss how DESS provides us with certain concepts that give it a benefit over standard UML to realize component evolution.

DESS defines a UML profile for component-based development of embedded systems. We present the specific extensions to UML provided herein which eased the realization of component evolution based on these concepts. These extensions are: the notion of provided and required interfaces, the extended modeling facilities for component interconnections and the notion of subcomponent plugs. The extensions cover both notation and architecture.

4.1 Notation

Provided and required interfaces: DESS [13] introduces the provided and required interface as a replacement for the UML <<uses>> and <<realizes>> concepts as shown in figure 1. Connections between the required interface of one component and the provided interface of another component are directed and can only be established if the interfaces of the two components are *compatible*.

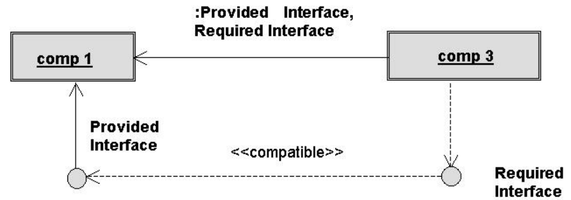

Figure 1: DESS component notation for required and provided interfaces.

Furthermore, a connection between a required and a provided interface can be either *frozen* or *dynamic*. Using the concept of dynamic connections and extending and formalizing the notion of compatibility between interfaces provides us with a powerful notation to model evolution.

4.2 Architecture

Connection decomposition: DESS allows the decomposition of connections into a middle-ware component and two simple connection components as illustrated in figure 2. This concept can be extended to include the decomposition of connections in a middleware and a *wrapper* component. This wrapper serves as a kind of interprocedure and maps the old interface to the new version using the delegation mechanism from [11]. These interprocedures are automatically generated.

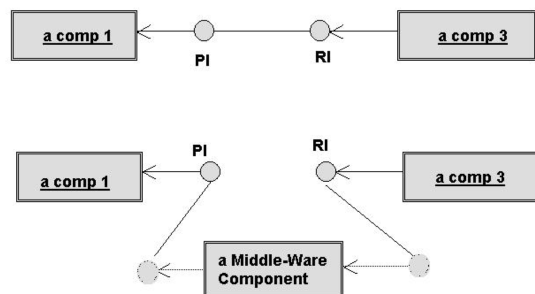

Figure 2: Connection decomposition in DESS

We must ensure that the computing and memory overhead of these mapper functions and interprocedures doesn't cause the embedded application to fail. A solution for this problem is the documentation of CPU and memory usage of these components in timing and memory contracts as in [9] which can be monitored at run-time.

All components must have performance attributes as described in DESS, available trough a special method in the interface. These attributes are used to specify contracts on the resource consumption of components, on the time a connection (connections are asynchronous) may take, on the resource consumption of wrapper components and state transfer. Combined with run-time monitoring these provide us with a valuable tool to realize real-time component interaction.

The re-wiring of component connections must be as easy as possible. Therefore all communication between components will be asynchronous and in a standardized format such as XML. The correct delivery of a message is the responsibility of the component system which has a directory of all component instances and their versions. This way components are unaware of the possible co-existence of different versions.

The actual changing of component versions relies on the dynamic component re-wiring capabilities of the component system and on *controlled* garbage collection. When a new version of a component is introduced, the old version continues its execution until it dies naturally. All messages to that component are from then on sent to the new version or, if necessary, they will be sent to a wrapper component for the new component. Since the duration of a single operation in embedded and real-time software is relatively short, the old component is expected to die very shortly after its update.

The execution of two component versions in parallel will not impose violations of the timing or memory constraints of the system since we delay the update using a technique inspired by [4] until the necessary resources are available to perform it safely.

Subcomponent plugs: DESS allows component composition of subcomponents. The composed component is modeled using subcomponent plugs which specify the interface a subcomponent must supply in order to be able to be plugged in there. As with component interfaces, we can extend the notion of *compatibility to the plug specification* to allow the insertion of multiple versions with changed interfaces.

However, at subcomponent level we cannot rely on the component system to deliver messages to the right version of a subcomponent. Therefore we have chosen to adopt the *delegation* mechanism from [11]. Using this technique all new messages from the component's interface will be delegated to the new version of the subcomponent.

The delegation mechanism is illustrated in figure 3. Each subcomponent with an interface compatible to interface b can be inserted in the *Text2PS plug*. Messages that arrive in a are *delegated* to b. This means that if a message arrives in a that expects an older version of the subcomponent with a different interface, the calls to methods in b that have changed will be automatically delegated back to a where they will be handled. In the handling of the delegated method call in a, we will adjust the arguments of the call to the changed version and call the changed method in b. If the interface of the subcomponent b has the correct version, the messages are just forwarded from a to b. Again, we expect the old subcomponent to die naturally.

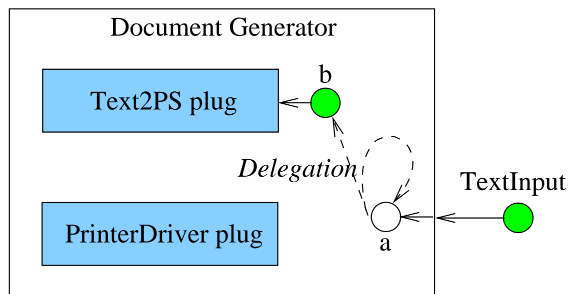

Figure 3: Delegation at interface level

5 Status and Future Work

A prototype supporting the concepts presented in this paper is currently being implemented. This prototype supports the dynamic updating of DESS components.

We will extend this prototype with run-time support for timing and memory contracts. This includes the implementation of a runtime mechanism for contract monitoring and tool support for the automatic generation of patches and monitoring code. As a proof of concept a test case will be developed using the EMPRESS methodology and component system.

6 Conclusions

In this paper we presented a new approach for software evolution in the field of component-based embedded systems. In this approach we had to deal with the scarcity of resources in our target embedded platforms. This resulted in thoroughly thought design choices for the supporting runtime system and explicit constructs to specify resource constraints in the form of timing and memory contracts.

References

[1] The EMPRESS consortium. The empress website, 2002. http://www.empress-itea.org/.

[2] The DESS consortium. The dess software development process for real-time embedded software systems, 2001. http://www.dess-itea.org/.

[3] The DESS consortium. Definition of components and notation for components, 2001. http://www.dess-itea.org/deliverables/ITEA-DESS-D144-V02P.pdf.

[4] M. Segal O Frieder. On dynamically updating a computer program: from concept to prototype. In *The Journal of Systems Software*, pages 111–128, February 1991.

[5] M. Hicks. *Dynamical Software Updating.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, June 2001.

[6] J. Andersson and T. Ritzau. Dynamic code update in jdrum. In *Workshop of Software Engineering for Wearable and Pervasive Computing*, June 2000.

[7] D. Gupta. *On-line Software Version Change.* PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[8] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the third international conference on configurable distributed Computing*, 1996.

[9] S. Van Baelen D. Urting and Y. Berbers. Embedded software using components and contracts. In *ECOOP 2001 SIVOES workshop, Budapest, Hungary*, June 2001.

[10] T. Holvoet Y. Barbaix, K. Hermans and Y. Berbers. Tuning parameters for component based design with memory constraints. In *ECOOP 2000 Workshop on Pervasive Component Systems*, June 2000.

[11] Günter Kniesel. Type-safe delegation for runtime component adaptation. In R. Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628, pages 351–366. Springer-Verlag, New York, NY, 1999.

[12] H. Lieberman. Using prototypical objects to implement shared behaviour in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 214–223, 1986.

[13] The DESS consortium. The dess methodology, 2001. http://www.dess-itea.org/deliverables/ITEA-DESS-D1-V01P.pdf.