Developing Reusable Software Components for Embedded Systems

ECE750-T11 Component-Based Software Systems
Spring 2009

Babak Omidi
Tony Zhao
Group 6 - Project Presentation 1
Date: July 13, 2009
Agenda

- Definitions
- Background
- Motivation
- Literature Survey
- Problem Formulation
- Platform Independent Abstraction
  Architectural Style
Definitions – Embedded System

- A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system (IEEE, 1992) [1].
- 98% of all computer systems are embedded systems [1].
Definitions – Architecture and Architectural Style

- Architectural styles are recurring organizational patterns and idioms [2].
- An architecture is a conceptual model of how a system’s components are defined and developed, and how they interoperate [3].
- An infrastructure is the implementation of the conceptual model [3].
Definitions – Software Component

Software Component as defined by Szyperski:

- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [4].
Background – Software Component Elements

To be able to describe a component completely and to ensure its correct integration and updating, the component should consist of the following elements [5]:

- A set of interfaces provided to, or required from, the environment. These interfaces are particularly for interaction with other components.
- An executable code, which can be coupled to the code of other components via interfaces.
Background – Liberal View of Software Component Elements

- Some researchers believe that the strict view of Software Component Elements does not work well with the special requirements of embedded systems (e.g. reliability, safety and timing) [1].

- Liberal View: To be able to describe a component completely and to ensure its correct integration and updating, the component should consist of the following elements [1]:
  - A set of interfaces provided to, or required from, the environment. These interfaces are particularly for interaction with other components.
  - A source code, which can be coupled to the code of other components via interfaces.
Background – Liberal View of Software Component Elements

- Source code does not necessarily need to be modified to adapt the component.
- However, source code is needed to be compiled on the target platform.
- Some have chosen to keep a repository of component variants which are optimized on multiple target platforms [6].
Motivation

- Software **reuse** is good
  - Recent analysis of more than 13,000 problem reports collected by the mobile phone company Ericsson in Grimstad, Norway, has shown that software reuse does result in significantly **fewer problems** and **better stability** [7].
  - Reused components have significantly **lower defect-density** than non-reused ones [7].

- **Component Based Software Engineering (CBSE) enables software reuse** [1, 8, 9]
Motivation Cont’d

- The growing complexity of embedded systems requires methods that improve reusability [10].
- Software for embedded systems is typically monolithic and strongly platform-dependent [11].
Motivation Cont’d

- Other advantages of using CBSE for embedded systems [11] are:
  - Shorter development time
  - Better maintainability
Motivation Cont’d

- Most existing and well known component architectures are not suitable for embedded systems in their current form [12] (e.g. Pipe and Filters, N-Tiered, Layered, Blackboard and MVC).

- Component models such as .Net, J2EE and CORBA Component Model (CCM) cannot be used for embedded systems [12].
Literature Survey

[6] proposes a repository of components that contains variants and versions of the software optimized for each target platform.

- Waste of storage space
- The component developer needs to predict all future uses of the component. This is not feasible.
“Open Implementation” defines principles for system decomposition and interface design [13].

- Requires the programmer to tune the underlying implementation at composition time.
- Forces the programmer to learn the specifics of the component.
- The component cannot be treated as a blackbox
Literature Survey

- Component models developed are often targeted to specific application domains [12]:
  - PECOS [11]
  - Koala [15]
- They do not address the issue of platform specific code.
Literature Survey

- FIRSL (Flight Instruments Reuse & Standardization Library) provides a modified system architecture for Flight Software (FSW) [14]
  - FIRSL publishes standardized API's for reusable device functions
  - Components are open source code units
  - A step in the right direction which needs to be formalized and generalized!
Problem Formulation

- Need to separate *hardware independent* functionality from *hardware specific* functionality.

- This needs to be done at the highest level of abstraction:
  - A new *Architectural Style* needs to be developed.
Problem Formulation Cont’d

*From class notes*
Proposed Architectural Style – Platform Independent Abstraction

Hardware/Simulation Layer

Interface Mapping Layer

Operating System Layer

Hardware Independent Layer
Platform Independent Abstraction - Example

Hardware/Simulation Layer
- Physical Audio Control
- Physical Dialing Control

Operating System Layer
- Hardware Specific Audio Control
- Hardware Dialing Control

Interface Mapping Layer
- Audio Control Mapper
- Dialing Interface Mapper

Hardware Independent Layer
- Play MP3
- Play Video
- Send Fax
- Make Call

Platform Independent Abstraction - Example
Platform Independent Abstraction – Example Cont’d

- Detailed architectural model to be provided in the next presentation.

- Essentially a layered architecture
  - Each layer communicates only with the layer below it.
  - Layers below have no knowledge of the layers above.
  - Increasing abstraction towards the top layer
  - Maintainability, Reuse, Portability (replacing layers to work within a different context)
  - Difference is that each layer exports multiple interfaces
Platform Independent Abstraction – Example Cont’d

- Disadvantages of Layered Architecture are not applicable
  - Universally applicable – components by definition cannot spread around different layers
  - Performance is not affected since the levels of indirection get resolved at compile time
  - Determining the correct abstraction level is achieved by definition
  - Layer bridging is not allowed
Proposed Architectural Style - Details

- Hardware Independent Layer
  - Contains all components not dependent on the underlying hardware.
  - All hardware specific details are abstracted away.
  - Components could be composite components.

- Interface Mapping Layer
  - These are simple components.
  - Translate hardware functionality into hardware independent calls.

- Operating System Layer
  - Pipes calls to the Hardware/Simulation Layer.

- Hardware/Simulation Layer
  - Performs/simulates operations requested.
Advantages of Platform Independent Abstraction

- The majority of the software belongs to the Hardware Independent Layer. This includes the logical flow control which is most susceptible to errors.

- Development of the system is done on a desktop computer and the Hardware Independent and Interface Mapping Layer components are then transferred to the hardware platform. This approach has two advantages:
  - Since desktop computers are much faster than embedded systems, development and testing times are reduced significantly; ultimately improving time-to-market.
  - Development of such a system can be done by software engineers that are not necessarily familiar with the target hardware. This enables the utilization of stronger software developers as supposed to hardware engineers who might not be as proficient in developing software applications.
Proof of Concept

- In order to demonstrate the effectiveness of the “Platform Independent Abstraction” architectural style, we will be implementing a simple application that will be run on 2 embedded systems in addition to the hardware simulator. In order to do this, we will:
  - Develop and test the Hardware Independent and Interface Mapping components on a windows machine with the help of a hardware simulator.
  - Transfer the components to the target platforms without any modifications and demonstrate 100% component reuse
  - The two targeted platforms will be an ARM4I and an x86 processor.
References


References Cont’d


References Cont’d


References Cont’d
