Embedded System Design And Challenges

S. Sai Manimala, K. PrasanthKumarReddy.

ABSTRACT

Many embedded systems have different design constraints. Design culture dysfunction make design difficult to be successfully applying tools to embedded system. Integration of system design is increased. Due this there is a widening gap in size and complexity of chip-level design and design capabilities. To bridge this gap in design productivity a number of advances have been made in high-level modeling and validation. Specifically advances have been made in 'Abstraction and Reuse ` and 'Structured design methods `. Structured design methods are Component-Based Design and Platform Based Design. There are many trends for design embedded system. Out of that highly programmed platform and UML for embedded software development are recent one. In unified embedded system development methodology, these two can be combined. Though these two concepts are powerful in their own right, their combination magnifies the effective gain in productivity and implementation.

INTRODUCTION

1.1Overview:

Fabricating millions of transistors on chip has become easier, due to advances in microelectronics processing and devices. The microelectronic designers through advances in modeling and validation technique are exploring a number
of strategies. These are used to improve the design productivity and the quality of design. There is an impact of raising abstraction level at which designs are entered and validated on design quality and design time.

1.2 Components for embedded system design

For a system on chip (SOC) there are virtual components also. SOC represents implementation of a complete application on a single chip. SOC consist of a range of building blocks from processors, memory, to communication and networking elements. There may be top down or bottom up approach to building an application in SOC.

In the bottom up approach the application functionality can often be structured into various hardware and software components. Top-down approaches yield refinements that are then mapped to various hardware/software components. So, component may then be a piece of functionality implemented in software or as a dedicated piece of silicon hardware or combination of both. A component may be virtual in that it represents a well-defined functionality without an associated hardware hardware implementation.

The phrase "virtual component "is used to describe reusable IP (Intellectual Property) components. IP components are composed with other components, which are similar to real hardware components plugged into real sockets on the board.

1.3 Component Composition Framework

A composition framework provides reasoning capabilities and tools. The reasoning capabilities and tools enable a system designer to compose components into a specific application. These capabilities include selection of correct

interfaces, simulation of composed design, testing and validation for behavioral correctness and equivalence checks. A limited form of component composition is common in purely software system where environments often known as Integrated Development Environment (IDE).

IDE's are used to facilitate component selection and composition. Hardware component composition frameworks are more difficult to build as compare to software IDE's such as Microsoft Visual Studio. Part of complexity is due to various ways in which the integrated circuit blocks are represented, designed and composed. At higher abstraction levels, often a connection between components is create through limited set of ports and signals in. It is often known as, structured design for SOC's. Such a composition implies a structural representation for the components. Even if a component is not structural, but behavioral, it can often be composed using special components (e.g. protocol modules) interconnecting the components. To ensure systematic compatibility of models, it is important to address how the composition is resolved along each of the dimensions. The dimensions are temporal detail, Structural detail, Functional detail, Data value detail. This is often achieved by creating wrapper around the library components.

Wrapper are created for enabling communication values between different modules and co-ordination between them. Wrappers here refer to code that enables reuse of existing component models. Using programming languages, there are several ways in which in which such wrappers can be built. A common strategy for wrapper building is by using inheritance available in most object-oriented programming languages. In this approach wrappers are programmed manually by inserting code inside the inherited class. The wrapper and the component are same as the object.

An alternative is to use a wrapper that, if needed, delegates to the design component. In this case, component is not modified, the wrapper and the component are two distinct object. Modules from different libraries can be imported as is, and dynamically placed in wrappers at run time.

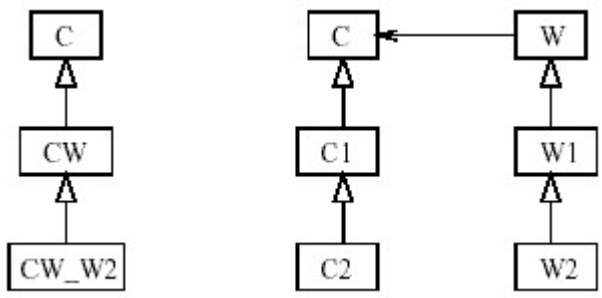


Fig. 1. Wrapper implementation strategies: (a) by inheritance (b) by composition

Figure 1(a) shows how wrapper is implemented by inheritance. In this case, if a designer wants to reuse a component of class C, the class can be specialized by inheritance to a subclass CW to implement the wrapper functionality. If the class CW is to be reused in a different context, then it can be also inherited into a class CW_W2 that implements more wrapper code to interoperate in the new integration context. The problem in this scenario is that all the three classes have a common self, and the original component has to be modified in every reuse context, via inheritance.

Figure 1(b) shows the UML diagram of how a wrapper hierarchy can be built for composition (the open arrow indicates an association). In this case, the wrappers are separate from the component object hierarchy, and the interoperability interface remains separated in the wrappers, and any call to functionality of the original

component is delegated from the wrapper to the component.

A good CCF provides a composition language and capabilities for dynamic composition, simulation and verification. The composition language is either visual or textual. Composition language should be able to ask for components from the component library. Framework must have automated support for selecting the correct type that makes the composition possible. The composition should be dynamic i.e. one does not have to go through recompile-test cycle when new components are added or replaced.

In order to be able to compose components at different levels of abstraction, and/or models of computation, met an information should be available. The meta information is about the components at a meta-level, such that it can allow users to understand implications of composing two arbitrary components.

Composition language: It is not used for specification of components. The role of composition language instantiates and connect the components. The component model describes the connection by dictating how and when things can be composed. A connection may be "relation" among components.

STRUCTURED DESIGN METHODS

2.1 Structured design methods types:

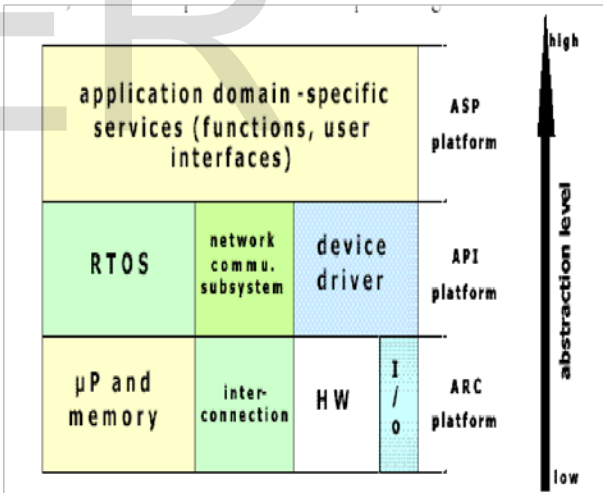
Platform Based Design and Component Based Design

Platform based design has emerged as one of the key development approaches for complex systems. The choice of platform is done after exploration of both the application and architecture design spaces. The choice of platform is driven by cost and time to market considerations. In component based design, components can be distributed or local. Distributed components can be thought of as objects that contain both data and operation. They are small service providers. The key is that they can be used as inputs (or arguments) to operations provided by other components and returned as the output from these operations.

2.2 Platform base design: Platforms are classified into three abstraction levels: architecture (ARC), application programming interface (APT), and specific programmable (ASP) platforms.

The ARC layer includes a specific family of micro architecture (physical network).

Hence, UML deployment diagram can be used to represent the ARC platform. The API layer is a switch abstraction layer wrapping ARC implementation details. The API should be presented by showing what kind of logical services are provided and how they are grouped together.



Platforms at Different Levels ASP is a platform, which makes a group of application domain-specific services directly available to users. For e.g., the function to set up a connection in the Intercom is such domain-specific service. In addition to calling these existing services, users sometimes need to modify or combine them, or even develop new services to meet certain requirements consequently, unlike API, here it becomes essential to show not only what functionality these services offer, but also how such services are supported by

their internal structures, and how they relate to each other. In UML, the class diagram best represents such information.

2.3 Component Based Design:

This approach promotes separation of interface and implementation. It provides a statistical environment built up from smaller components. And allows different algorithms and operations to be performed. This is done in different ways without minimal changes to overall environment. However, an implementation for small amounts of data may store it all in

memory. For very large quantities of data, it might be stored in compressed form either in memory or on disk. Alternately, the values may be produced in real-time from a device.

This approach brings up many different issues such as performance, security, reproduction, discovery etc. While the last of three of these have been dealt with in the context of the Internet, performance has not. A component-based architecture will more likely lead to increase in performance, with a little work. Suppose we have located and use a component that fits a tree model, but too slowly. Provided that, we can communicate with it only via its operations, we can locate a faster version of this.

Due to this substitutability of components it has got potential success. A little further thought leads us to see how components can give us what is termed high performance computing. These days' multiple processor machines and clusters involving multiple machines are becoming common. In simple terms, we can imagine each processor being associated with a component and one in charge of dispatching the subtasks that make up an overall computation. This task manager invokes operations in these distributed components and awaits the answer and pieces them together, potentially issuing new tasks to idle component.

2.4 Acknowledgments

The preferred spelling of the word "acknowledgment" in American English is without an "e" after the "g." Recognition of the importance or quality of something. acceptance of the truth or existence of something

Author
S. SAI MANIMALA,3RD YEAR, ECE, GITAM'S UNIVERSITY, VISAKHAPATNAM.

Author
K.PRASANTHKUMARREDDY3RDYEAR, MECHANICAL,LINGAYA'SUNIVERSITY FARIDABAD.

2.4 References

We would like to express our special thanks of gratitude to our Senior Professors who was always willing to help and give their best suggestions. Special thanks to the reviewers for pointing out ways to improve the presentation of this paper.