SOFTWARE DEVELOPMENT SOLUTIONS FOR MULTIPROCESSOR AND SOC SYSTEMS USED IN SDR

Toby McClean (Zeligsoft, Gatineau, QC, Canada, toby@zeligsoft.com)
Mark Hermeling (Zeligsoft, Gatineau, QC, Canada, mark@zeligsoft.com)

ABSTRACT

The advances in system-on-chip (SoC) and multiprocessor platforms have made software development for these environments much more involved. Today's platforms contain multiple processors, often from different classes (DSPs, FPGAs, GPPs and so forth), and each of these processors can contain multiple processing cores. The traditional approach used to develop software cannot address this complexity due to two major shortcomings. Firstly, it views the world as a single layer in a homogeneous environment or at most as an application layer deployed to some form of an execution environment layer. Secondly, the traditional approach uses a code-based development environment to develop the application layer, which provides the developer with no information about the overall system.

This paper looks into why the traditional approach no longer suffices when developing software for complex platforms and what other technologies are available to bridge the gap.

1. INTRODUCTION

The modern software engineer needs to develop software that executes on multiple different platforms with varying types of processors, high-speed buses, peripherals and accelerators. The engineer requires information to understand the impact of the distribution of software and the usage of the platform. This information includes the available MIPS, MMACS or number of gates, but also the latency and throughput for the physical connections on a platform. Change is constant in the software development arena, and that is true for distribution as well: It will change during the life-cycle of a project. This puts more pressure on non-functional attributes like reusability and portability, which were never easy to deal with in the first place.

Besides information, the software engineer needs assistance and guidance. The engineer needs help to make sure that the application can adjust to change while maintaining important properties like performance and code size. The software engineer also needs help to enable communication across the processors in these complex platforms.

The complexity of current-day platforms makes it more difficult for the engineer to obtain the information he needs and to develop the software that satisfies requirements. This fact, combined with tightening product development cycles, has reached a breaking point. This breaking point cannot be solved by growing the development team; a new solution to developing software for complex platforms is required. Technical articles appearing in trade magazines, web sites and journals are increasingly clamoring for a solution to this problem.

In this paper we will briefly look into the shortcomings of traditional software development approaches. From there we will explore best-practices such as graphical modeling, component-based development, layering and code-generation to see how they can alleviate the problem. We will show how layering can be used to include the physical hardware layer and how this applies to the system-on-chip and multiprocessor world. We will also show how layering can benefit code-generation to result in smaller, faster code.

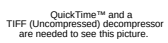

Figure 1 -- The Mercury Computer Systems Ensemble2 is an example of a complex platform

The resulting solution is a best practice based approach to provide the software engineer with the tools to deliver the next generation of complex systems on time, within budget and with the highest possible software quality. It bridges the gap between the current methodologies and the new, complex platforms that are being introduced.

2. TRADITIONAL APPROACHES TO SOFTWARE DEVELOPMENT

2.1 Architecture-Centric Development

Most medium to large, modern day software development projects use an architecture-centric approach to development [1]. That is, the development team sits down early on in the project and maps how they plan to divide responsibilities in the system. An architecture typically involves horizontal partitioning of the system into layers and/or vertical partitioning into subsystems. The

architecture also describes the allocation of functionality to processors and cores and the communication medium used in the interactions between them.

There are typically three main goals to this partitioning: 1) Divide and conquer complexity to facilitate implementation; 2) Decouple the different parts of the architecture to increase reuse; and 3) Increase the ability to relocate functionality to other processors in the system.

Architectures come in many forms. Many development teams use graphical modeling tools utilizing diagrams like class diagrams and collaboration diagrams from the Unified Modeling Language (UML) [2].

2.2 Current Development Approaches

Once the architecture has been laid down, the development team can start to translate that architecture diagram into a more detailed design and ultimately code. This is typically a manual process, modeling tools such as the IBM Rational [3] and Telelogic Rhapsody [4] solutions are popular, but some teams prefer to do this with white boards.

The problem with all of these approaches is that they do not give the developer the ability to address the complexity of the platform. These approaches focus on developing application functionality; they do not express how that functionality is mapped to the platform or how the pieces communicate together.

The development team has to make a-priori decisions on where to allocate functionality and what communication facility to use. They must then embed these decisions into their source — people often refer to this as the runtime architecture code. The run-time architecture code can occupy as much as 50% or more of the entire system.

Runtime architecture code is notoriously difficult to write and debug, even more so for complex platforms. Developers often hard-code communication, resulting in the increase of coupling between different parts of the application as well as between the application and the platform. Any change to the platform, such as the communication medium or the hardware impacts the runtime architecture and can result in multi-man-months worth of effort.

The development team will also have to work on configuring the platform. Complex platforms have flexible communication busses with routers or switches that need configuring. They also have processors with accelerators that can be turned on and off. The result of this configuration activity could be low-level C code, linker control files, XML files or other artifacts. This activity requires highly specialized knowledge, which is often difficult to find, train and retain.

As part of the mad dash to a working system, the original architecture often gets sidelined. The original design intent gets forgotten and developers make shortcuts where they need them — like pointers into entities that were meant to be decoupled, system calls that were not intended and the like. The result is that the original architecture deteriorates, this is not immediately noticeable, but it does impact non-functional attributes of the system like re-usability and portability.

The result of this is a working but rigid system. This rigidness prevents the team from making changes to the allocation of functionality to the platform, and it prevents them from finding the optimal allocation. It also impacts reusability of the code base, due to the coupling between code and platforms. This in turn makes it more difficult to build a family of products using different variants of hardware boards.

Middleware like CORBA [5] can alleviate some of these problems, however, it also causes a level of overhead to be incurred that, while typically acceptable between GPP processors, is often unacceptable for low-level signal processing functionality on DSPs and FPGAs.

2.3 Summary

The impact of what we have described is far more severe then it might seem at first glance. It impacts embedded software development projects as follows:

- The runtime architecture is time consuming to establish;
- Exploring deployment possibilities for the optimal deployment is time consuming;
- Reuse of application software and platform software is tedious;
- Changing deployments is expensive;
- Deterioration of architecture

As was previously stated, these problems cannot be solved by adding more people. The problems are real, have been reported by many projects and will not go away by themselves. A new way of development software for complex platforms is needed.

3. SOLVING THE MULTIPROCESSOR AND SOC DILEMMA

The previous section described some of the challenges that teams face when developing software for complex platforms. From this a number of requirements can be distilled that must be satisfied by a development environment for these systems. Development teams need the ability to:

R1 Change the allocation of functionality to hardware;

R2 Change the choice of communication media;

R3 Make effective use of resources and peripherals on a complex platform;

R4 Configure a piece of software, platform or SoC;

R5 Utilize multiple platforms in a product line.

R6 Communicate software and system architecture and design.

The next section will look at several well-known best practices and how they can be used together to provide the flexibility that software developers need.

3.1 Component-based development

Component-based software development concerns development of software as independently deployable, encapsulated and reusable elements. The entire system is divided into components with strongly typed interfaces. The interface of a component expresses how the component interacts with the environment and not how the component is implemented.

A component can describe control behavior like routing, or it can describe signal processing behavior like Forward Error Correction (FEC). A component can contain other components (hierarchical composition) or can have one or more implementations. An FEC component can, for example, have an implementation for a particular RTOS on a GPP or DSP, or for a particular flavor of an FPGA.

A set of source code files typically implement a component. This source code has two completely different responsibilities. Firstly the source code implements the behavior of the component, but secondly, the source code also communicates with other components in the system. These responsibilities should be kept separated, the functional behavior of a component will always be the same, but the communication behavior depends on the communication mechanism chosen.

Several component-based standards exist. The OMG's *Deployment and Configuration of Component-based Distributed Applications Specification (D&C)*'[6] the *Software Communications Architecture (SCA)* [7] and the *Automotive Open System Architecture (AUTOSAR)* [8] are some examples.

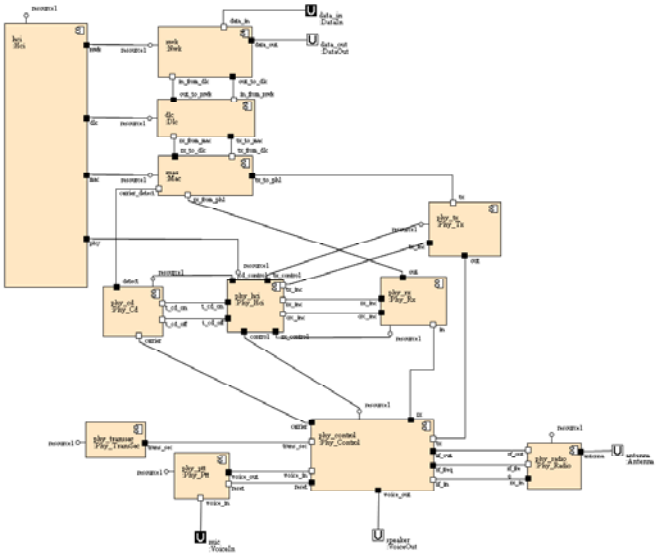

Figure 2 -- Example of a component-based application

The use of components enforces encapsulation of the component's internal functional logic from the environment. This results in a component being independently reusable. A component can be stored in a component library and reused over multiple projects.

Composing applications of multiple components makes that application deployable over a distributed system. The amount of distribution is only limited by the level of granularity of the components. A single application can have a number of deployments, each deployment representing a different allocation of components in the application to processors in the platform.

Some component-based standards, like the SCA, dictate a particular layer of middleware for management and communication (in the case of the SCA the Core Framework and CORBA and MHAL). However, a management layer is not mandatory for a component-based system. Likewise, a middleware layer is not required. Components can communicate over RTOS messaging and platform or customer proprietary messaging frameworks.

Components place requirements on their execution environment. These can be simple requirements such as "200 MIPS", or "500 MMACS", or more complex requirements such as a multi-channel serial port with a 16-bit word length and a 2 word frame.

The use of components for software satisfies requirement **R1 (change the allocation of functionality to hardware).**

3.2 Graphical modeling

The use of components, as highlighted in the previous section, requires that a graphical representation be present to help developers properly understand, communicate and evaluate the systems under construction. Modeling allows teams to represent the system visually, from a high level, and down through all the detail concerning the components and their requirements. Software teams are often geographically dispersed and the use of software models allows them to work together easily.

The model of the system describes the set of communicating components, or, in other words, the runtime architecture mentioned before.

The graphical model contains the components (described in the previous section) but also contains other elements described by additional concepts introduced in the next sections below.

The use of graphical modeling for capturing the architecture satisfies requirement **R6 (communicate software and system architecture and design)**.

3.3 Layering

Complex software uses the concept of layering to further improve encapsulation. A layer groups together components

at a certain level of abstraction. Typical layers are an application layer specific for a particular piece of software, a business layer and a platform layer.
Layering is an important technique to satisfy **R5 (utilize multiple platforms in a product line).** The lower layers change when the platform changes, but the higher layers only change when the application changes. Layers are able to contain the churn when a certain aspect of a system changes.
Layers require services from lower layers and provide services to higher layers through service access points and service provisioning points respectively.

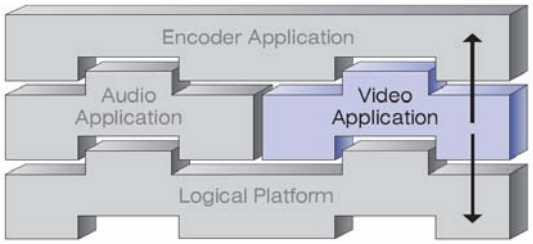

Figure 3 -- Layers in a software system

Layering can be used to include the physical aspects of the system. This represents the actual chips and physical connections in the platform, the elements that consume power to perform functions.
Components are the main building blocks of these layers, even for the logical execution environment and the physical layer. Components can represent operating systems, DSPs, FPGAs, partitions on FPGAs, logical communication busses (for example CORBA), or RTOS messaging. Components can likewise represent chips, hardware connections, memory, multiplexers and bridges.
Logical and physical components have properties that describe resources of interest, for example latency, MIPS and MMACS.
As a rule of thumb, a system consists of 3 to 5 layers, including the physical layer.

3.4 System-centric development

The layers mentioned in the previous section describe the entire system, from the hardware layer up. However, the layers are completely independent by design and need to stay that way so that they can be reused in other products or with other hardware. A system-centric model includes all the layers and also has the capability to capture and configure relationships between the layers; this is done in a new modeling concept called a 'deployment'. A deployment contains a number of model elements from different layers and it configures the elements and stages the components in the higher layer to the components in the lower layer.

Configuration is the act of providing settings for the properties on the layer. For example, the TCP/IP stack on an RTOS can be provided with an IP address. A CORBA bus can be provided with the location of the naming service in use. This satisfies requirement **R4 (configure a piece of software, platform or SoC)**.
Deployment includes configuration and staging:

1. Configuring of properties on the components;
2. Staging of the components in one layer to the layer below;
3. Staging of connections between components to logical and physical communication busses;
4. Connecting services between layers.

The configuration and staging steps combine the layers in the system and aggregate them into a complete system representation. The deployment describes how the components work together, where they execute and how the layers are connected.
The developer can easily change a staging by dragging a component to a different processor, a connection to a different communication bus and so forth. This satisfies requirement **R2 (change the choice of communication media)**.
The deployment is a final check-point for the user to verify whether the system will actually work. That is, the user can verify whether the resources, provided by the physical layer upwards, meet the requirements expressed by the components.
The system can be queried and analyzed as well in order to calculate all possible stagings, or to find, for example, the staging with the lowest latency. This satisfies requirement **R3 (Make effective use of resources and peripherals on a complex platform)**.

3.5 Generation

Modeling is required to provide developers with necessary understanding and means to communicate. However, modeling reaches the peak of its benefit if it is also used to generate implementation code.
Components and layers in the model are kept completely encapsulated and independent of each other and the platform. They come together in the deployment; hence this is where code is generated from and is known as Deployment-Aware Generation™ (DAG).
The code make-up of a component, containing functional logic and communication and control code, was described in 3.1. The functional logic of a component is fixed. It is written by the developer and contains the actual behavior of the component.
The communication and control code depends on how the component is used in the application, which other components the component is connected to, and the communication media selected for its outgoing and

incoming connections. DAG code can generate this code automatically from the deployment in the model.
This is called design time location transparency. It is important to note that the functional logic of the component is completely independent of the communication framework. Communication in this case could be CORBA, TI DSP/BIOS MSGQ, VxWorks messaging or a proprietary transport. The user makes this choice during design time.
DAG allows for very tight and efficient code to be generated as there is no code, memory or run-time overhead, it simply contains the code that is needed for the communication. DAG generates code for the entire run-time architecture that was mentioned earlier.

3.6 Summary

The best practices described in the previous sections combined together provide the developer with a powerful set of tools to tackle his projects. Component-based development, graphical modeling and layering have been used for a long time with great success. System-centric development, which includes a model of the physical layer as well as deployment models, is new. Zeligsoft has pioneered deployment modeling in its flagship product Zeligsoft CE. Deployment-Aware Generation, that is, generation from the deployments in the system-centric models is brand-new and this facilitates developers writing code for complex systems, as they no longer have to write the run-time architecture code, while retaining the high performance that they need.

4. CONCLUSION

This paper began with a discussion on the challenges software developers faced when dealing with today's complex embedded systems. The problems they were experiencing were then investigated and articulated, and then distilled into a number of requirements.
A system-centric software development approach was then presented that builds on object-oriented programming and standards like the SCA, but combines it with a truly system-centric view, that is, a view that includes the distribution of software over hardware. A system-centric approach uses graphical models, component-based technologies, layering, deployments and generation. The system-centric approach provides the necessary tools for the software developer — including high-level architects, software implementers, and testers. These tools allow the developers to see their work in the light of the larger system.
Generation with the system-centric approach also provides for the tightest code generation possible. It combines complete knowledge about the system with advanced generation techniques. It provides design time location transparency, but it can also support the runtime location transparency offered by middleware layers like CORBA.
The system-centric approach is the best of all worlds. It gives the designer the power to make informed decisions by querying the graphical models he builds with his, potentially geographically dispersed, team.
The system-centric approach is supported by the Zeligsoft CE 3.x software development environment, currently targeting advanced TI DSPs (TCI6482, TCI6487, C6455 and other members of the c64x+ DSP core family). Other processors will be available to meet market and customer demand.

5. REFERENCES

[1] Bass, L., Clements, P. and Kazman, R. Software Architecture In Practice 2nd Edition. Addison-Wesley Professional, 2003
[2] OMG. Unified Modeling Language (UML). http://www.omg.org/cgi-bin/doc?formal/07-02-03.
[3] IBM Rational. Rational Systems Developer. http://www.ibm.com/software/awdtools/developer/systemsdeveloper/index.html.
[4] Telelogic. Telelogic Rhapsody. http://www.telelogic.com/products/rhapsody/index.cfm
[5] OMG. CORBA/IIOP Specification. http://www.omg.org/cgi-bin/doc?formal/04-03-01
[6] OMG. Deployment and Configuration of Component-Based Distributed Applications. http://www.omg.org/docs/ptc/03-07-02.pdf.
[7] Software Communications Architecture (SCA). http://sca.jpeojtrs.mil/
[8] AUTOSAR, http://www.autosar.org