2006-1761: ADDING SYSTEMS ENGINEERING ACTIVITIES TO THE SOFTWARE CURRICULUM

Harry Koehnemann, Arizona State University

Dr. Harry Koehnemann is an Associate Professor in the Division of Computing Studies at Arizona State University where he performs teaching and research in the areas of distributed software systems, software process, and modeling software-intensive systems. Prior to joining ASU in 2001, Harry worked several years as a software architect and software developer on software systems ranging from large enterprise applications to embedded control systems. Harry has also provided training and consulting services in software tools and technologies, software modeling, and software process.

Adding Systems Engineering Activities to the Software Curriculum

Abstract

This paper motivates the need for introducing systems engineering activities into the software curriculum and describes the changes made to an embedded software course to support systems engineering concepts. While still a hotly debated topic, the role of what some consider traditional software techniques are useful and becoming established activities during the systems engineering of large, complex systems. As software engineers play a larger role in the systems engineering activity, understanding those activities and their role in systems engineering are vital for software engineering education.

The addition of systems engineering activities in and embedded software course has led to many successful outcomes. Students understand the approach for solving complex systems. They also observed first hand the benefits of modeling a solution before committing to an implementation. In fact, student remark that once their model is correct, building for the target platform is relatively simple. Finally, the value of showing UML notation and the various diagrams in the context of systems development is vital for students, as these activities are becoming common in the systems engineering community.

1 Introduction

Systems engineering activities are responsible for many decisions in complex systems. They specify the system's behavior, partition behavior into hardware and software components, define the communication between components, establish the assembly and deployment strategies for components, and specify the associated hardware and software architectures. In practice, systems engineering teams have historically been under-represented by software advocates and their results have been historically weak in areas of software concerns, particularly lifecycle development process, tools, and architecture.

Systems are more commonly selecting off-the-shelf hardware resulting in less need for application-specific hardware solutions and therefore more demand on software specifications. At the same time, system integration responsibilities have become more prevalent, requiring interface and control though means such as networking and the web. For example, the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) is now commonly used to abstract communication across different processors in embedded systems ([1] as an example). In fact, most OMG meetings are now dominated by embedded systems developers in contrast with the information technology (IT) developers who dominated meetings less than a decade ago. Embedded systems are also employing other IT strategies such as web services and enterprise service bus architectures to handle their integration requirements ([2] as an example).

In addition, system size and complexity have changed dramatically over the past several years. While still a hotly debated topic, the role of what some consider traditional software techniques

are useful and becoming established activities during the systems engineering of large, complex systems. Solutions include use case requirements elicitation and management, modeling, component-based development, and the specification of lifecycle process and tool usage [3] [4]. As software engineers begin to play a larger role in the systems engineering activity, understanding those activities and their role in systems engineering are vital to software engineering education. Preparing software engineering students for a role in the systems engineering process has become an important factor for the student's success as well as the success of large systems.

This paper discusses the addition of systems engineering activities to an existing course titled "Internet-enabled Embedded Devices." The course is offered in the Division of Computing Studies at Arizona State University at the Polytechnic Campus. The course objectives originally introduced students to systems built from loosely coupled embedded devices communicating via a network. Projects were fairly substantial and ranged from making embedded devices accessible through the web (e.g., a browser-controlled sprinkler timer) to systems built from loosely coupled devices communicating via the Internet (e.g., integrated traffic control signals). The device control issues were relatively simple serial communication so the course focused on network communication via the web. While this paper describes modifications to a single course, the success of these additions has led to changes within other courses. The goal of these changes is to standardize on common methods and a common hardware platform for hardware and embedded students across the program.

2 Background

This section provides background material for the course changes. First, the term systems engineering is discussed and defined for the context for this paper. Second, modeling and its use for representing system behavior and then generate software artifacts (e.g. source code, deployment descriptors). The third section discusses system requirements and the final section discusses build and deployment processes.

2.1 Systems Engineering

Since this paper motivates and describes systems engineering additions to the software curriculum, defining the systems engineering discipline is an important first step. While the definitions vary across communities, below are some common characteristics of the systems engineering discipline:

- Problems are cross-discipline and require a wide vary of expertise
- Problems and solutions are complex and commonly hierarchically composed into subsystems to facilitate multi-site and multi-vendor development
- Solutions are highly heterogeneous including hardware (FPGAs, DSPs, general purpose processors), software (languages, descriptor files, binary formats), and tools (compilers, cross-compilers, debuggers, emulators, test platforms, build and deployment scripts)

- Projects have long life-spans, ten to twenty or more years, where methods for life cycle processes are vital to the system's long-term success

- Projects require significant management skills for budgeting (product development, staffing, tools, development platforms, manufacturing, and support and maintenance), utilizing subcontractors and vendor partners, considering time-to-market, and managing project risk

Like the characteristics, the activities required for systems engineering are also broad reaching and open for debate. However, typical activities performed for a systems engineering effort include specifying the system's behavior, making significant hardware and software architectural decisions including tradeoff analysis, and defining the system's lifecycle processes including development process and tools.

2.2 Modeling

Of all the engineering disciplines, software development as an overall community lags the others in its use of modeling to analyze problems and then synthesize solutions. Early modeling efforts in the software community tried to abstract software source code into models so a tool could translate the models into a complete software solution [5]. These efforts met with limited success. While they were good at generating a system's structure (classes and method declarations), they did little to generate the system's behavior (the method implementations).

Other, more successful efforts modeled the system's behavior based interactions between components. The goal was not to model the end source code, but to model the system itself by specifying its elements, their internal behavior, and their interactions. With this approach, a system is partitioned into components (and possibly nested components) that communicate through well defined interfaces, commonly called 'ports'. These ports are typed and provide the only communication path between components. The port types specify the signals (a.k.a. messages, events) both sent and received on those ports. Internal component behavior is specified with state diagrams whose transitions are the response to external events received on the component's ports. Many system engineering notations [6] [7] advocate component-based approach for systems development.

The example in Figure 1 shows a Switch component that provides two ports, one of type Electricity and one of type Toggle. A Bulb component consumes Electricity and can therefore be connected (a.k.a. 'wired') with a Switch component on the Electricity port. Switch and Bulb can be created completely independent of one another so long as they agree to a common communication protocol defined by Electricity.

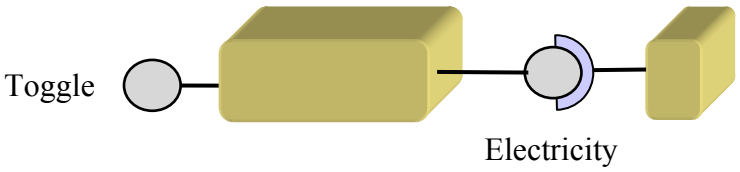

Figure 1: Component Diagram

Internal component behavior is modeled using state diagrams. State diagrams specify the component's response to messages (a.k.a signals, events) sent to its ports. The response can include an internal state change as well as possibly generating a message out one of its ports. Figure 2 shows an example state diagram for the Switch component. When entering the Off state, the switch sends the stop message out its Electricity port. When the Switch receives a click message on its Toggle port, it transitions to the On state which causes the Switch to send the flow message out its Electricity port. Other components connected to the Electricity port receive the 'stop' and 'flow' messages and respond accordingly.

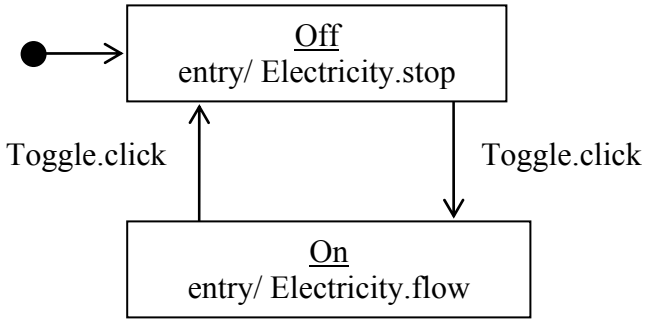

Figure 2: State Diagram for Switch

Many component models use port-based communication and several tools that support this approach to modeling systems. The Unified Modeling Language (UML) [6] and Systems Definition Language (SDL) [7] are industry standard notations for modeling large systems and provide facilities for partitioning a system (components) and defining the component's behavior (state diagrams). UML-based tools include IBM-Rational Rose RealTime and ILogix Rhapsody while SDL tools include Telelogic Tau. These tools are commonly used in large systems to define the control logic and, once modeled, can be used to simulate the system's behavior as well as generate significant portions of the source code.

2.3 System requirements

System success depends on all stakeholders (customers, developers, domain experts, end users, etc.) agreeing on a system's requirements. As behavioral requirements for embedded systems have grown in complexity, the community has looked for new approaches to discovering, specifying and communicating requirements. For example, early cellular phone systems provided the ability to dial and connect. Modern cellular phones include call history, phone book, settings, ring tones and images, messaging, games, web browsers, etc. Most of this added behavior has little if any association with the real-time communication control logic of the embedded device. Consequently, large-scale software systems have adopted more traditional software requirements techniques, such as Use Cases, to specify certain types of behavior.

The Use Case approach [10] to requirements managements and elicitation is well known in the software community and is slowly being adopted by the systems community. Use Cases view

the system from an external perspective, specifying how a system responds to external events from its users and other devices or systems with which it interacts. As behavioral requirements become more complex, Use Cases are a useful tool for managing them.

2.4 Run-time and deployment architecture

As discussed earlier, large systems are typically heterogeneous and include multiple types of devices, languages, and tools. In addition systems engineering is responsible for defining the lifecycle processes as well as defining the budget for development. System-level decisions play a significant role in the lifecycle costs of system development. Obviously, the hardware architecture directly dictates much of the product's material costs and for many large systems the production hardware environment represents a significant overall cost. However, there are more subtle decisions that impact the lifecycle costs of system development.

First, developers must be able to build and execute their portion of the system . Large systems may have hundreds of developers and it is unrealistic to expect they will all develop using production hardware configurations. Systems engineering commonly specifies tiers of development environments including simulators for host platforms and subsets of the production hardware for target-level testing. While final testing will be performed with production hardware, host platforms are sufficient for testing many behavioral requirements and production subsets can reveal many timing issues without requiring the full hardware environment.

Second, large systems require the involvement of many people with diverse skills. The hardware and software architectures must be designed to support concurrent development as well as the ability to outsource portions of the system to other organizations. The component-based designs discussed must consider development across organizational boundaries and address issues including intellectual property, safety, and security.

While constructing large, complex systems is challenges, releasing, deploying, and upgrading those systems presents similar challenges. Systems engineering efforts must define strategies for releasing new versions of the system, how those versions will be deployed, and how running systems will be upgraded. As with concurrent development, component-based designs provide assistance by partitioning the system. But system engineering must formulate a plan for the system's lifecycle.

3 Curriculum modification

This section defines course modification made to an existing embedded devices course offered each spring in the Division of Computing Studies at Arizona State University's Polytechnic Campus. The first offering began in spring 2002 and the modifications were implemented in spring 2004 and 2005. Those modifications drove several faculty discussions involving the hardware and embedded program offerings within the Division which is leading to changes in several embedded and hardware courses. These changes are discussed further in the conclusions.

Section 3.1 describes the original embedded course, Internet-Enabled Embedded Devices, and its goals and outcomes. Section 3.2 discuses modification made in spring 2004 and 2005 to add systems engineering activities presented in the background section. Section 3.3 discusses

changes made to the embedded track within the Division for 2006 including the modified courses and the changes made to those courses.

3.1 The Internet-Enabled Embedded Devices course

The course was originally designed to expose students to multi-processor embedded systems that communicate via a network and to connect devices to a network using socket-based protocols. The original course topics included some minimal device interfacing through serial and other communication but focused on network protocols and communication strategies for embedded devices.

The course uses the Dallas Semiconductor Tiny INternet Interface (TINI) [11] as the hardware platform which runs a scaled-down Java virtual machine. The device supports 1 meg of flash for the kernel and 1 meg of battery-backed SRAM for the file system and program execution. The TINI also supports a wide variety of interface ports including serial, 1-wire, and TCP/IP. Motivating factors for this device were its low cost (< $100), free development tools, and support for the Java language which is widely used in the program's curriculum. Using the Java programming language in an embedded software course was a questionable decision and one we will revisit later. The motivating factors for Java were its 1) extensive use in the curriculum, 2) excellent support for socket communication as well as open source solutions for web servers and distributed object communication and finally 3) support interfaces for connecting external devices.

The course addresses both system-to-system and system-to-person communication. The system-to-system protocols included custom, byte-level protocols, distributed object protocols such as Java's RMI, and standard-based protocols such as XML-RPC and SOAP. Students discovered the ability of limited embedded devices to process requests using each of these protocols. System-to-person communication protocols used HTTP and the ability to serve dynamic web pages with images and java script. Most projects have some form of user input requiring the embedded device to support a web interface.

An example project from the course might be an Internet-enabled sprinkler timer which provides a web-based user interface for settings and uses the weather forecast via the Internet in its decision process for watering. Projects like this included device interface to open and close sprinkler valves, networking to check the weather forecast from the web, and some web-based user interface to set the watering strategy for the various zones. There are several advantages of this Internet-enabled device over the current electronic devices. First the Internet enabled device provides a much richer user interface for scheduling watering timing based on time of year than can be offered economically through the push buttons of the electronic device. Second the device is more predictive by including weather forecasts from the Internet in the watering decision algorithm.

While the original course met its goals of exposing students to Internet-enabled embedded devices, glaring needs arose during the first two offerings (1992 and 1993). First, students had fewer problems with the technical parts of the course (the networking protocols and device interfaces) than they did with the problems themselves.

3.2 Adding systems engineering activities

This section discusses the added activities made to the course. The additions are specifying the system's requirements, modeling the system's architecture, and describing both the host and target build and deployment and physical architecture. All modeling and code generation, both host and target, is performed in Rational's Rose RealTime.

3.2.1 Specifying behavioral requirements

Students specify behavioral requirements for the system early with Use Cases. They represent the specifics of each Use Case with a UML activity diagram following the approach advocated in [4]. Defining behavior early in the project forces students to think about the entire project's before implementing a solution. An example for a traffic light system is shown below.

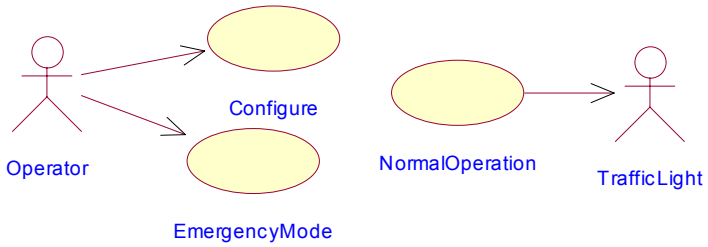

Figure 3: Traffic Light Use Case Diagram

In normal operation mode, the light cycles between green, yellow, and red lights. It must also detect and synchronize with other traffic lights. An operator can send the light into an emergency mode which flashes red and configure the traffic light for default light timing as well as configure properties for communication (network ports, etc.). As in [4], the system's behavioral response to each Use Case is further described with a UML Activity diagram. The authors in [4] advocate a single Use Case Event Diagram for each Use Case that describes the ordering of external events received by the system. Each event in the Event Diagram can also have an optional Event Response Diagram describing the externally visible system behavior that occurs as a result of that event. The figure below shows an example Use Case Event Diagram for the Emergency Mode Use Case.

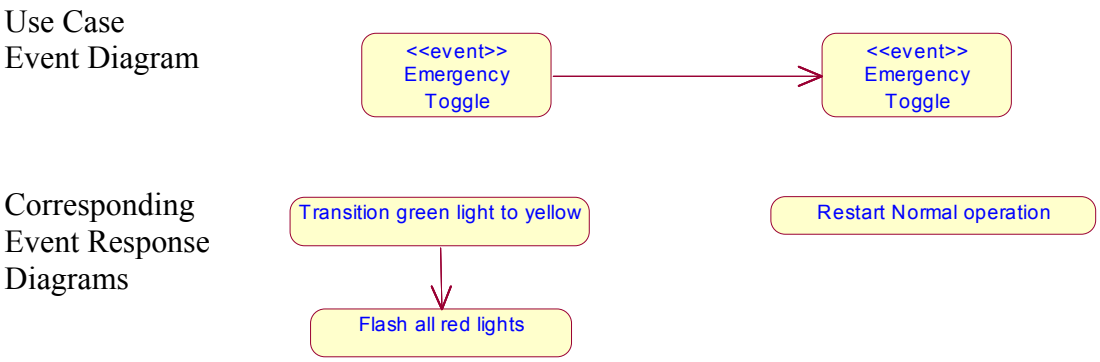

Figure 4: Emergency Mode Event Diagram and Event Response Diagrams

3.2.2 Modeling system architecture

Next, students design architectural solutions using UML models. Modeling the system's logical software architecture forces students to consider the system's components and the messages sent between them. Students are given an introduction to UML emphasizing diagrams and notation for describing system and software architecture, Use Case, Class, State, and Sequence diagrams. Particular attention is also given to diagrams in the 4+1 View Model [12] as those architectural views are heavily used in Rational Rose RealTime. The course also discusses other architectural views used to specify large systems [13].

The Figure 5 shows the components for the traffic light system. Of note is the Controller which contains three Lights (the 3 multiplicity on the relationship between Controller and Light) which represent red, green, and yellow light devices. The Light component abstracts the behavior of a light device which can toggle between on and off. The logic for this behavior is described with a simple state diagram moving the Light between on and off states. The controller determines which light is illuminated at any given time and sends appropriate messages via ports to the three Light components. Figure 6 shows a portion of the Controller's state diagram. The left diagram represents the controller's main diagram which toggles between normal and emergency states. The right diagram describes the internal behavior for the Normal state, which runs lights from red to green to yellow.

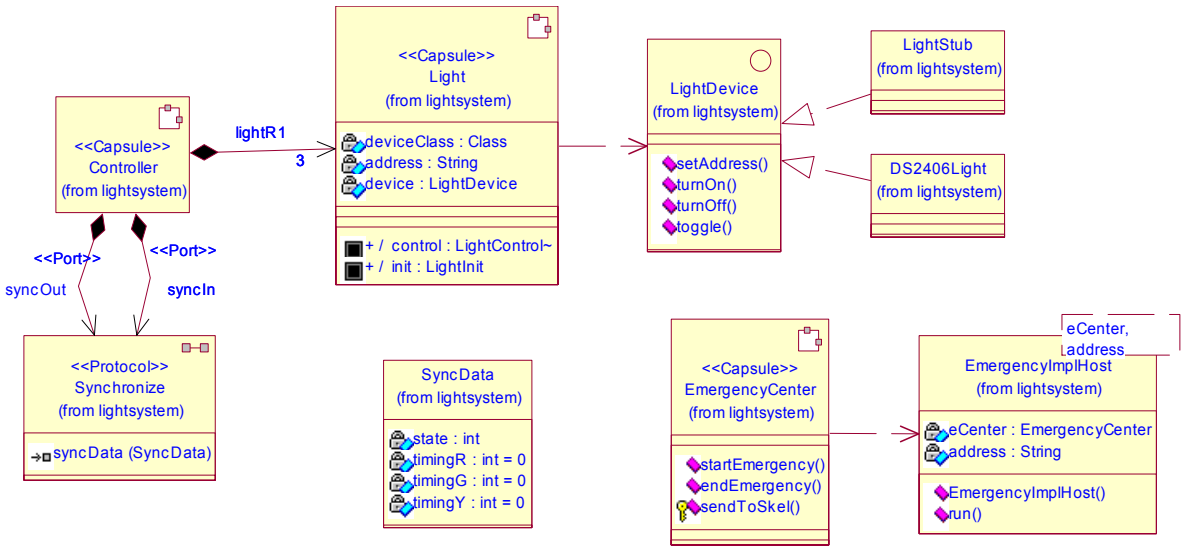

Figure 5: Traffic Light System Logical Components

Also of note in Figure 5 is the Light Device class. Recall systems engineering must consider both the host and target development environments. On the target system, the Light abstraction will be turning a physical device on and off, while in the host environment no such device will exist. Light abstract the system's behavior for the physical light and invokes the appropriate LightDevice to change its state. The system defines two implementation of LightDevice, a LightStub for operation on the host and a DS2406Light, which has appropriate device interface code to connect to the TINI LEDs.

The other classes, SyncData and EmergencyControl handle distributed communication. SyncData packets are transmitted to synchronize multiple traffic lights and the EmergencyController broadcasts the emergency event to all traffic lights when an operator transitions the system into emergency mode.

All components communicate through ports. The Controller turns Lights on and off by sending signals to the Light's control port. Likewise, synchronization is performed for each traffic light's sync ports – note the syncIn and syncOut for receiving and sending sync data.

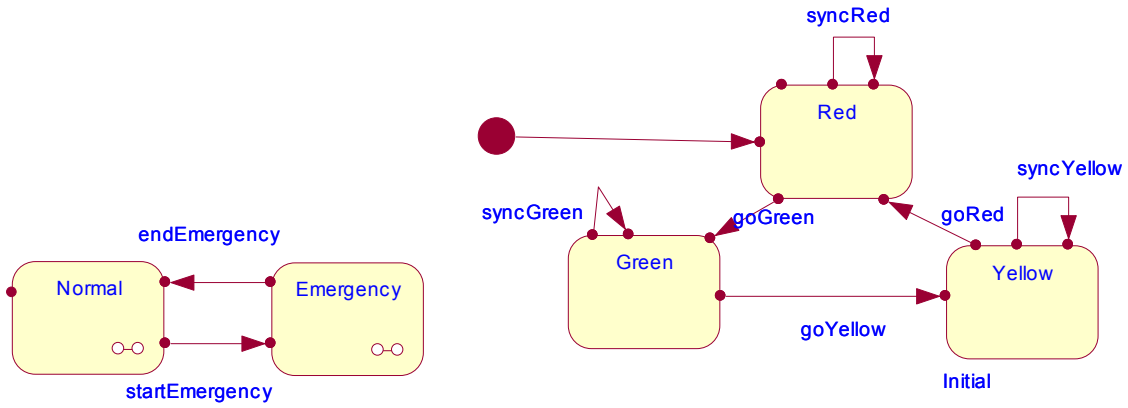

Figure 6: Controller's state diagram and expansions of the Normal state

3.2.3 Describing deployment and physical architecture

Once behavior has been assigned to logical components, students specify the system's physical deployment architecture where the logical components from above are mapped to physical components. Figure 7 shows the physical components for the Traffic Light system. Rose RealTime provides scripts to compile and deploy each component. Notice there are separate physical components built for the host (LightSystemHost) and target (LightSystemTini) environments. Those components will use different compilation tools and include different logical components (e.g. LightStub in the host and DS2406Light in the target). Also notice the application for toggling the emergency signal is its own application which runs on the host.

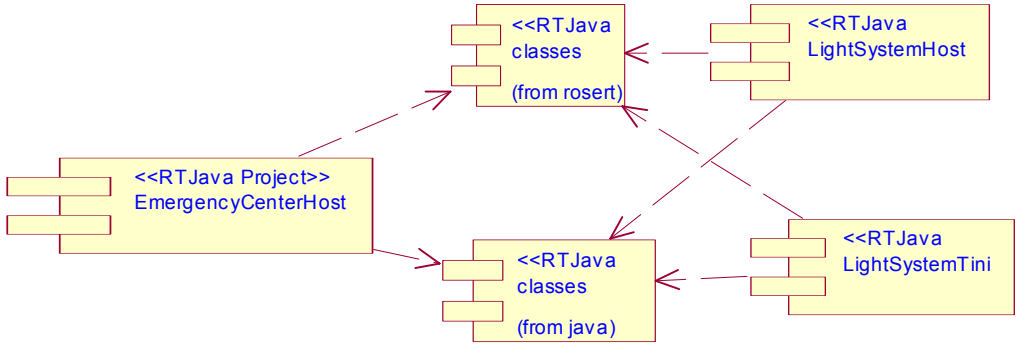

Figure 7: Traffic Light System Physical Components

3.2.4 System and project management

Student projects end up with many devices and wires connected to a breadboard for their final demonstration. To motivate student understanding of systems engineering, they must also report their project's final production and development environments including associated costs. They must consider what devices are needed, how they can be manufactured, and discuss opportunities for reducing those costs. In addition, students must report the development environment costs including devices required and what system behavior could be created without physical devices or by stubbing those devices on a host environment.

3.3 Future course modifications

As discussed earlier, several faculty members within our Division have discussed more collaborative projects. Large-scale embedded applications require a broad range of expertise and our goal is to provide some of that experience. Devices executing in those applications include FPGAs, DSPs, as well as general purpose processors. Our goal is to combine projects from these types of courses to create more realistic system opportunities for students.

We are currently reviewing the Xilinx XUP Virtex II Pro Development System [14] as a common platform for hardware and embedded courses. It uses a multi-core processor that supports two PPC 405 cores along with substantial FPGA space. It also provides several interface ports (serial, network, video), switches, LEDs, and buttons for interfacing and supports linux and the gcc toolset for compilation.

From a systems engineering perspective, this board will allow students to engineer more realistic systems and collaborate with students from other disciplines. An example project might be an MP3 player where buttons are used for fast forward and rewind. Such a system would require components that include hardware designs that are loaded onto the FPGAs. The systems engineering must consider this development process and design a solution for interfacing the FPGAs with the rest of the software as well as build and deployment strategies in both the host and target environments.

4 Results and conclusions

This paper described systems engineering activities added to an embedded software course. The additions were motivated by student's problems developing larger projects. While students understood the device interface and network communication issues, they struggled with understanding and implementing non-trivial embedded systems. The following system engineering activities were added to the course to address this problem: 1) specifying the system's behavioral requirements, 2) modeling the systems logical architecture and interfaces between logical components, 3) establishing the system's physical architecture and build and deployment processes, and 4) consider overall development and production cost implications of their architectural choices. The course uses Rational Rose RealTime for modeling system behavior and generating a significant portion of the resulting software code for both the host and target environments.

The first offering with modeling was spring 2004. Initially students did not like the learning curve associated with a commercial development tools like Rose RealTime. In addition, most

felt the added activities discussed above were simply busy work and did not add value to the creation of their projects. A couple practice exercises using Rose RealTime were added in the spring 2005 offering which helped the learning curve. By the end of the semester most students appreciated the value of modeling the system's behavior, primarily because it generated a majority of their code. By the end of the semester, students became so confident their host models would run on their target system they would wait until the last day to actually run on the target hardware. While obviously not a recommendation, it did emphasize the importance of modeling and understanding a system as well as the value a host development environment plays in system development. Unfortunately, most students never gained an appreciation for the behavioral requirements effort.

A side benefit also occurred for a handful of students who received job offers primarily from their understanding of UML and model driven development obtained in the course. Modeling is becoming more prevalent in the software community, particular in the systems area, and knowledge of how modeling can be used to drive development is a value for students.

On the negative side, using a commercial tool, commercial methods, and a unique hardware platform for a single course can be substantial effort and risk for faculty. First, the software licensing and installation issues must be resolved. Rose RealTime is available for academic use through IBM's Academic Initiative. Every student in the class installed the software on their personal or work computers, so there was no issue with campus IT for installation.

Faculty members can expect to become the technical support contact for installation and configuration issues as well as product usage questions. Assigning no-credit exercises using the tool (Rose RealTime in this course) helps as does a class discussion board so students can answer each other's questions. But faculty will be the final point of contact for technical problems. In this particular course, the author had prior experience with Rational Rose RealTime with industry applications. So, product knowledge was less of an issue than it may be for others.

In conclusion, the activities added to this course were typical of those performed in the practice and important for student's education. As systems become larger in scope, the need for software students to understand and interact with other disciplines becomes increasingly important. This paper provides a set of activities to use in the classroom.

Bibliography

[1] Dohse, D., Bush, L., Osborn, G., Christensen, E., "Successfully Introducing CORBA Into the Signal Processing Chain of a Software Defined Radio", COTS Journal, January 2003.

[2] The System of Systems Common Operating Environment (SOSCOE) for the Future Combat System, see http://www.army.mil/fcs/factfiles/overview.html for reference.

[3] Alexander, I., Zink, T., "An Introduction to Systems Engineering with Use Cases", Computing and Control Engineering Journal, December, 2002.

[4] Coats, M., Mellon, T., "Integration CMOS with UML", Dr. Dobb's Journal, June 2001.

[5] Booch, G., Object-oriented Analysis and Design with Applications, 1993.

[6] Object Management Group, System Engineering Domain Special Interest Group (SE DSIG), http://syseng.omg.org/

[7] SysML Forum, http://www.sysml.org/

[8] Unified Modeling Language, Version 2.0 http://www.omg.org/technology/documents/formal/uml.htm

[9] Specification and Description Language, Z. 100, ITU-T recommendation

[10] Jacobson, Ivar, Christerson, M., Jonsson, P., and Overgaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, 1992.

[11] The TINI Specification and Developer's Guide, Don Loomis, 2001.

[12] Krutchen, P. "The 4+1 View Model of Archite cture",IEEE Software, 12 (6), 1995

[13] Garland, J., and Anthony, D., Large-Scale Architecture: A Practical Guide using UML, 2003.

[14] Xilinx XUP Virtex II Pro Development System , http://www.xilinx.com/univ/xupv2p.html