High Confidence Embedded Software Design: A Quadrotor Helicopter Case Study

Zhenkai Zhang Joseph Porter Nicholas Kottenstette
Xenofon Koutsoukos Janos Sztipanovits
Institute for Software Integrated Systems (ISIS)
Department of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
zhenkai.zhang@vanderbilt.edu

ABSTRACT

Traditional design methodology is not suitable for high-confidence embedded software due to the lack of a formal semantic model for software analysis, automatic code generation, and often designed embedded software is hard to reuse. In order to automatically generate high-confidence and reusable embedded software, we propose a TLM-centric, platform-based, time-triggered and component-oriented method. We use this new method to generate the control software for a quadrotor helicopter.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design

Keywords

Computer aided software engineering, Real-time systems, Embedded software, Digital control, Graphical models

1. INTRODUCTION

In many cyber-physical systems (CPS), how to automatically analyze and generate high-confidence embedded software becomes a key issue. High-confidence embedded software is needed in hard real-time systems, e.g. safety-critical systems. Moreover, designers also want to reuse successful software to save money and energy spent in developing these systems, and most importantly, to save time-to-market. However, there are some drawbacks that impede the automatic analysis, generation and reuse of embedded software: (1) Generating high-confidence embedded software requires a formal model (or models) containing the necessary semantics to enable software analysis; (2) Embedded software needs to be integrated with the underlying hardware, and the integration makes embedded software hard to develop, analyze, and reuse; (3) Timing requirements and software performance vary among different systems, so porting the software to another system might violate timing requirements; (4) When trying to port monolithic embedded software to a distributed system, it is hard to guarantee some correctness properties will remain (e.g. determinism and deadlock freedom).

In order to have a model containing all the semantics for automatic generation, deal with this tight coupling to the underlying hardware, make the timing of the embedded software easy to be analyzed and controlled, and provide a solution for distributed systems, many new design methods have been proposed. Among the model-based design methods, the *Transaction-Level Modeling* (TLM) is systematic and suitable for automatic code generation [4][2]. For dealing with tight coupling to the underlying hardware, platform-based design provides an abstraction layer that hides the details of several possible low-level refinements [3]. Time-triggered architecture is a particular platform abstraction which is used to analyze timing behavior of embedded software [5]. Actor-oriented design provides a method that specifies formal models of computation for execution of distributed components [7].

Although many different issues can be addressed by using these different methods, there is little work considering all these design concerns for a specific application. In order to automatically generate high-confidence and reusable embedded software, we propose a TLM-centric, platform-based, time-triggered and component-oriented method, and we use this new method to generate the control software for a quadrotor helicopter.

2. OVERVIEW

In our method, we start with a specification model (SM) of the control system using Simulink. After validation of this SM by simulation, we import the model into an automated embedded software development environment. The environment uses a suite of domain-specific modeling languages (DSMLs) called the Embedded Systems Modeling Language (ESMoL) to integrate analysis and code genera-

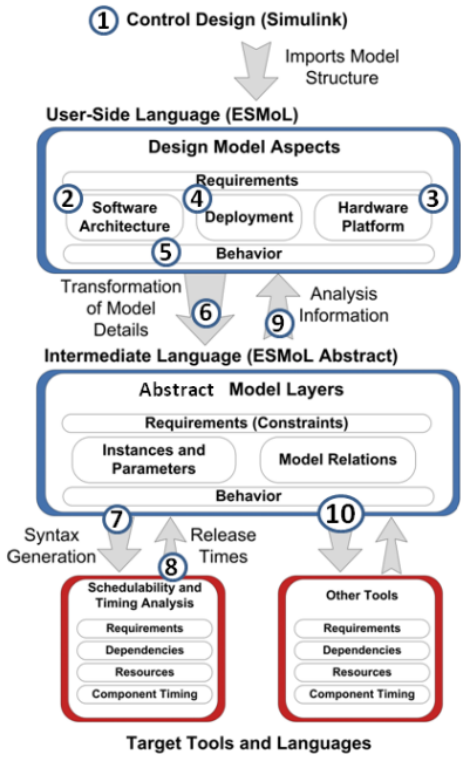

Figure 1: Design flow supported by the ESMoL language and modeling tools.

tion tools. In the ESMoL environment, we establish a TLM based on the imported SM. The TLM captures the hardware platform of the system, the mapping of tasks to the processors and messages to the communication ports, and the scheduling information of the tasks. Based on this TLM, we perform embedded software synthesis which consists of code generation and binary generation. We evaluate the binary code on the target platform to check performance with requirements.

ESMoL is a suite of DSMLs which function together as a system level design language (SLDL), providing a single multi-aspect design environment. Modeling, analysis, simulation, and code generation are all related to a single design model. The design language is specific to distributed embedded control systems, and is described in [9]. We follow the design flow shown in Fig. 1.

Step 1 is to specify the control system's functionality in the Simulink environment. After validation of this control system design, the model can be imported automatically into the ESMoL environment. The Simulink model will become a synchronous dataflow (SDF) model, and each subsystem in the Simulink model becomes an actor in the SDF model [8]. ESMoL model references to imported Simulink blocks become the functional specifications for instances of software components in a logical SDF model. C code fragments may also be used to specify component functionality. Component ports (shown in Fig. 2) represent instances of data message types. These types are defined as structures with individual data fields to which Simulink data ports can be mapped. These relations describe the marshaling, demarshaling, and transfer of data between software components [9].

Step 2 is to specify the logical software architecture which captures data dependencies between software component instances independent of their distribution over different processors.

Step 3 is to define hardware platforms hierarchically as hardware units with ports for interconnections. Primitive components include processing nodes and communication buses. Behavioral semantics for these network models come from the underlying time-triggered architecture. The time-triggered platform provides services such as deterministic execution of replicated components and timed message-passing. Model attributes for hardware also capture timing resolution, overhead parameters for data transfers, and task context switching times [9].

Step 4 is to set up a deployment model by mapping software components to processing nodes, and data messages to communication ports. The deployment model captures the assignment of component instances as periodic tasks running on a particular processor. In ESMoL a task executes on a processing node at a single periodic rate. All components within the task execute synchronously. Message ports on component instances are assigned to hardware interface ports in the model to define the media through which messages are transferred [9].

Step 5 is to establish a timing model by attaching timing parameter blocks to components and messages. For the time-triggered case the configuration parameters include execution period and worst-case execution time. The execution model also indicates which components and messages will be scheduled independently, and which will be grouped into a single task or message object [9].

The TLM scheduling information is added in steps 6–9. Step 6 translates an ESMoL model into the simpler ESMoL_Abstract model using the Stage 1 model transformation described in [9]. Step 7 is to use the equivalent model in ESMoL_Abstract to generate a scheduling problem specification according to a template. In step 8 a tool called *SchedTool* solves the generated scheduling problem. Step 9 is to import the results back into the ESMoL model and write them to the appropriate objects. For more details, please refer to [9]. Step 10 is to generate the corresponding C code, which will be described in next section.

3. MODELING AND CODE GENERATION

We use the above approach to design and implement the embedded software for a quadrotor helicopter. Quadrotor helicopters are agile aircraft which are lifted and propelled by four rotors. Because their attitude dynamics change so quickly, it is difficult if not impossible for a human to successfully fly and maneuver such vehicles [6]. Thus, these aircraft need an automated control system to help them fly. The controller, software and hardware design domains are highly specialized and conceptually incompatible. For example, control theory deals with a continuous system, software design is for a discrete environment, and computing hardware must deal with both. This makes effectively and efficiently implementing such a high-confidence embedded control system significantly difficult.

3.1 Simulink Control System Model

The control design for the quadrotor helicopter is intro-

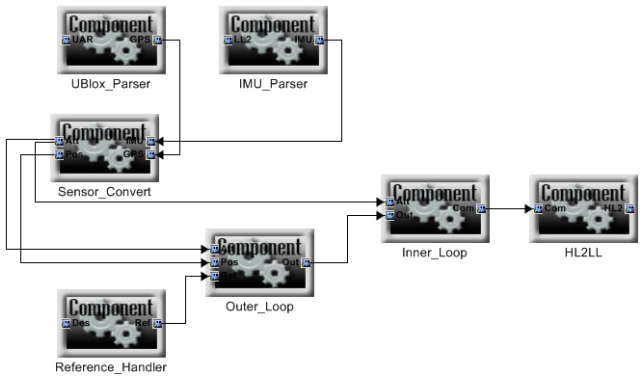

Figure 2: The logical software architecture for quadrotor's control system.

duced in [6], which uses passive attitude control. Specifically, two linear proportional derivative (PD) controllers are used, an inner loop and an outer loop. The outer loop controller is a "fast" PD inertial controller and the inner loop is a "fast" PD attitude controller.

The on-board sensors include a GPS and an IMU. In the Simulink model we do not capture the behavior and interfaces of the particular sensor chips, so their receiving message types are modeled not specifically but universally. The controller takes x, y, and z coordinates instead of longitude, latitude and height as position, so a specific subsystem *Sensor_Convert* is added to make the conversion. The *Outer_Loop* subsystem is for inertial position control and *Inner_Loop* is for attitude control. *Reference_Handler* is used to receive and handle destinations, and *Plant_Dynamics* is used to simulate the behavior (not realized as a software component).

3.2 Logical Software Architecture

Fig. 2 shows logical data dependencies between software component instances. There are 7 components needed: *Sensor_Convert*, *Reference_Handler*, *Inner_Loop* and *Outer_Loop* are all specified as Simulink subsystems; *UBlox_Parser* is for parsing the GPS data, *IMU_Parser* is for parsing the IMU sensor data and *HL2LL* is for coding the command data that can be sent to the actuators.

To establish functional determinism and deadlock freedom, we analyze the imported Simulink blocks in the logical architecture model as a SDF model. SDF guarantees that each actor (corresponding to a subsystem in the Simulink model) can fire at any time only if its input tokens (corresponding to messages) are available on its incoming arcs. In order to extend the execution semantics to include timing determinism while maintaining the benefits of synchronous execution implied by SDF, we employ a time-triggered model of computation (MoC). On a single processor we use a simple static task schedule without preemption. This allows us to implement a very simple scheduler which we can easily verify to ensure that deadlines are not missed due to task interference. In the case of a serious fault the scheduler could still miss a deadline, but failover to another controller is the only available recovery option. We have not yet implemented fault detection and recovery. The time-triggered MoC preserves function determinism and deadlock freedom

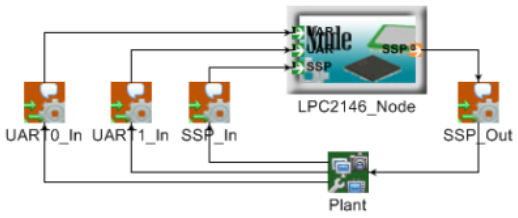

Figure 3: The hardware platform model of AscTec Hummingbird AutoPilot.

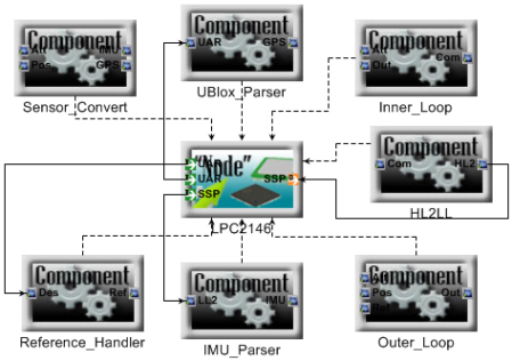

Figure 4: The deployment model of control system's software components.

of the SDF during distributed execution, as the actors all fire only at the scheduled times.

3.3 Hardware Platform Model

The quadrotor helicopter that we use is named AscTec Hummingbird AutoPilot from Ascending Technologies Company [1]. The quadrotor's hardware architecture is based on Philips LPC2146. Fig. 3 illustrates the hardware platform model. The processor LPC2146 is based on an ARM7TDMI-S CPU with two UARTs, SPI, SSP, and other peripherals. The peripherals are modeled in the diagram as objects connecting the input and output ports on the processor to the object representing the plant dynamics. A GPS device is connected through UART1, and a Zigbee module used to receive reference is connected via UART0. The IMU and actuators are connected through SSP. Each device can be configured by setting the *Configuration* attribute of the model object representing the device channel.

3.4 Deployment Model

In our case study, the model assigns each software component to its own task. In Fig. 4 the dashed connection from a component to a node reference represents an assignment of that component to run as a task on the node. The port connections represent the hardware channel through which that particular message will travel. Local data dependencies are not specified here, as they are represented in the logical software architecture. IChan (Input Channel) and OChan (Output Channel) objects on the node can also be connected to message objects on a component. These connections represent the flow of data from the physical environment through sensors (IChan objects) or the flow of data back to the environment through actuators (OChan objects).

3.5 Timing Model

Each component is assigned a *TTExecInfo* (Time-Triggered Execution Information) object that takes execution period (*ExecPeriod*) and worst case execution time (WCET) (*WCDuration*) as its parameters, and so is each external data transfer. For the processor-local data messages, transfer time is neglected, as reads and writes occur in locally shared memory. The quadrotor helicopter platform provides a fundamental sampling rate of 1kHz. The *ExecPeriod* attribute for all components is set as shown in Table 1. The fundamental rate required for the controller is 100Hz. Sensor and actuator data rates drive the other components. For example, since the time between two valid GPS samples is 100ms, the *ExecPeriod* for *Blox_Parser* is also 100ms, because it processes the GPS data. The worst case latency from sensors to actuators must be smaller than 10ms. Local message transfers may be specified as time-triggered, but in practice they take place in shared memory and are not scheduled. In ESMoL only distributed messages may be scheduled.

3.6 Code Generation

In order to generate the C code based on the TLM in ESMoL, two interpreters are used, which are in Stage 1 and Stage 2 respectively. The Stage 1 interpreter transforms the TLM to an equivalent model in an intermediate language called ESMoL_Abstract. The model in this intermediate language is flattened and the relationships implied by structures in ESMoL are represented by explicit relation objects in ESMoL_Abstract [9].

Stage 2 provides several interpreters, each of which uses the UDM model navigation API to generate either code or analysis models from the ESMoL_Abstract model. The deployment model objects are used to generate platform-specific task wrapping and communication code. Shared memory is used to implement the message passing through the ports.

The code generator uses the Google CTemplate engine called from C++ code to perform the generation tasks. We establish a template library containing initialization codes of different devices. This makes the control system code able to be used on different platforms with a variety of different sensors and actuators. Using the idea of separately generating functional and platform specific code is to realize the platform-based design concept.

Real-Time Workshop (RTW) generates functional ANSI C code for the subsystems specified as Simulink blocks.

Due to the lack of an operating system, we use interrupt-based multi-tasking. The timer interrupt service routine invokes the tasks according to the specified schedule.

4. EVALUATION

We empirically evaluate the execution time for each component using an external indicator. Timing requirements of the components are met. Each of them takes less than 10μs during normal operation. We also use a^3 tool from the AbsInt Angewandte Informatik company to analyze the WCET and stack usage for each component (shown in Tab. 1). From the table, we can see the total time of analyzed results is 649.7μs, which is less than 10ms that is the worst case latency from sensors to actuators, so the timing requirements can be met.

Table 1: Analyzed WCET & stack usage and sampling rate for each component

Component Name	WCET (μs)	Stack Usage (B)	Sampling Rate (Hz)
Outer_Loop	299	176	100
Inner_Loop	163	96	100
Sensor_Convert	70	52	100
Reference_Handler	0.45	4	100
UBlox_Parser	68.5	44	10
IMU_Parser	19	40	333
HL2LL	29.75	36	333

The memory system consists of 256KB on-chip flash memory (ROM) and 32KB SRAM (RAM). The corresponding binary code is about 110KB, so it fits in the system's ROM space. All the data variables for the communication are pre-allocated, and from the table we can see the maximum stack usage of a component is 176B. Empirically, we can evaluate that the RAM space is enough for data during normal operation.

5. FUTURE WORK

Our future work includes: (1) extending the control approach (and software implementation) for group coordination of multiple quadrotor helicopters; (2) modifying ESMoL to support time-triggered wireless network modeling; (3) analyzing the effect of fixed point implementation.

6. REFERENCES

[1] AscTec Hummingbird with AutoPilot User's Manual.

[2] L. Cai and D. Gajski. Transaction level modeling: An overview. In *Proc. of the Intl. Conf. on HW/SW Codesign and System Synthesis (CODES-ISSS)*, pages 19–24, Oct 2003.

[3] L. P. Carloni, F. D. Bernardinis, C. Pinello, A. L. Sangiovanni-Vincentelli, and M. Sgroi. Platform-based design for embedded systems. In R. Zurawski, editor, *The Embedded Systems Handbook*. CRC Press, 2005.

[4] D. Gajski, S. Abdi, A. Gertslauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer, 2009.

[5] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. of the IEEE*, 91:84–99, Jan 2003.

[6] N. Kottenstette and J. Porter. Digital passive attitude and altitude control schemes for quadrotor aircraft. In *ICCA '09: 7th IEEE Intl. Conf. on Control and Automation*, ChristChurch, New Zealand, 2009.

[7] E. A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[8] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proc. of the IEEE*, 75(9):1235–1245, 1987.

[9] J. Porter and G. H. et al. The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, Framework, and Analysis. Technical Report ISIS-10-109, ISIS, Vanderbilt Univ., 2010.