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)
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.**

***Keywords*-embedded software design; model-based design; TLM; time-triggered MoC; ESMoL; quadrotor helicopter;**

I. Introduction

In many cyber-physical system (CPS) designs, how to automatically analyze and generate the high-confidence embedded software becomes a key issue. High-confidence embedded software are needed in hard real-time systems, e.g. safety-critical systems. Moreover, designers will also want to reuse successful software to save money and energy, 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 needs a formal model (or models) containing the necessary formal semantics to enable software analysis. 2) Embedded software has to integrate with the underlying hardware, and such integration makes embedded software hard to develop, analyze, and reuse. 3) Timing requirements and software performance varies among different systems, so porting the software to another system might violate timing requirements. 4) If 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 [1][2]. For dealing with tight coupling to the underlying hardware, [3] gives a method called *Platform-based Design*, in which a platform is defined as an abstraction layer that hides the details of several possible low-level refinements. Time-triggered architecture is a particular platform abstraction which is used to analyze timing behavior of embedded software [4]. [5] proposes a method called *Actor-oriented Design* which specifies formal models of computation for execution of distributed components.

Although many different issues can be addressed by using these different methods, there is little published work considering all these issues together for specific applications. Therefore, we combine these methods and give a solution for high-confidence embedded software design.

The rest of this paper is organized as follows: Section 2 presents the background in more details about the techniques we are using; Section 3 gives a case about how to design an embedded controller for the quadrotor helicopter using our design method; Section 4 evaluates this design and finally Section 5 states our future work.

II. Background

Our embedded software design approach is TLM-centric, platform-based, time-triggered and component-oriented. In this 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 generation 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). It provides a single multi-aspect design environment so that modeling, analysis, simulation, and code generation are all clearly related to a single design model. This design language is specific to distributed embedded control systems, and is described in [6].

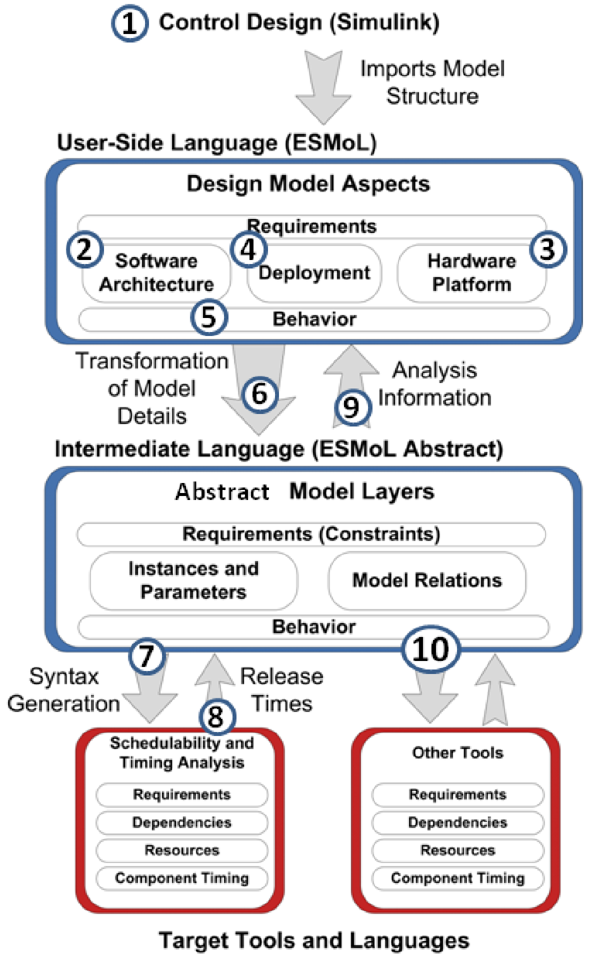

Figure 1. Design flow supported by the ESMoL language and modeling tools.

According to [6], 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 model (SDF), and each subsystem in the Simulink model becomes an actor in the SDF [7]. 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 instance 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 [6].

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 networks 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 [6].

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 [6].

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 [6].

The TLM scheduling information is added in step 6 through step 9. Step 6 translates an ESMoL model into the simpler ESMoL_Abstract model using the Stage 1 model transformation described in [6]. 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 to the appropriate objects. For more details, please refer to [6]. Step 10 is to generate the corresponding C code, which will be described in next section.

III. Modeling and Code Generation

In this case study, we use the above approach to design and implement the embedded software for a quadrotor helicopter. Our example is not distributed, but we are working towards distributed experiments and scenarios involving multiple vehicles. 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 [8]. Thus, these aircraft need an automated control system to help them fly. The controller, software and

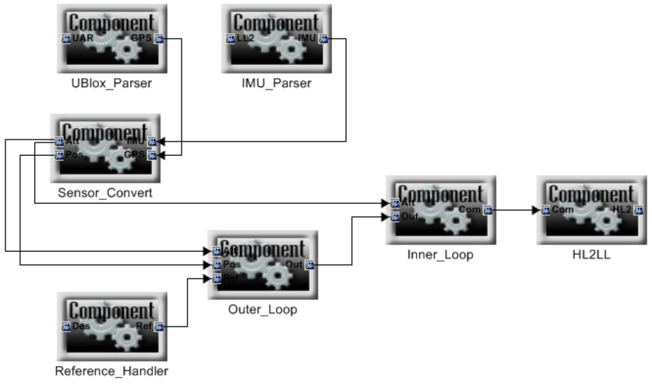

Figure 2. The logical software architecture for quadrotor's control system.

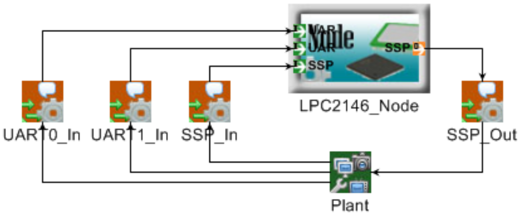

Figure 3. The hardware platform model of AscTec AutoPilot.

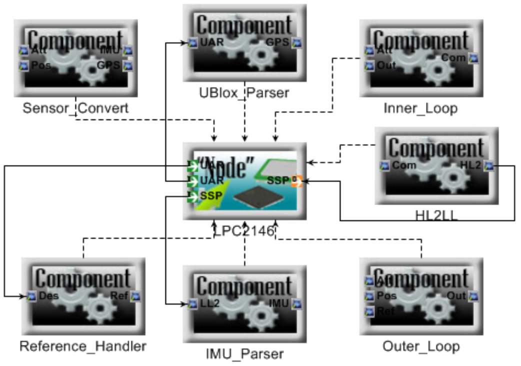

Figure 4. The deployment model of control system's software components.

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. Our embedded software design approach gives a state-of-the-art solution to this.

A. *Simulink model of the quadrotor's control system*

The control design for the quadrotor helicopter is introduced in [8], which uses passive attitude and altitude control schemes. In the control system of [8], 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. [8] describes the corresponding control approach in detail.

On most quadrotors, 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. [8] 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).

B. *Logical software architecture of the control system*

Fig. 2 shows logical data dependencies between software component instances. To determine 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 MoC. On a single processor we use a simple static task schedule without preemption, but we will extend our work to distributed cases. The time-triggered MoC preserves function determinism and deadlock freedom of the SDF, except that the actors all fire only at the scheduled times.

C. *Hardware platform model*

The quadrotor helicopter that we use is named AscTec Hummingbird AutoPilot from Ascending Technologies Company [9]. 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, etc... 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. For example, consider the string "*baudrate = 57600 8n1*" in the *Configuration* attribute of the channel port connected to device UART0.

D. *Deployment model*

In our case study, the model assigns each software component to its own task. In Fig. 4 the dashed connection from

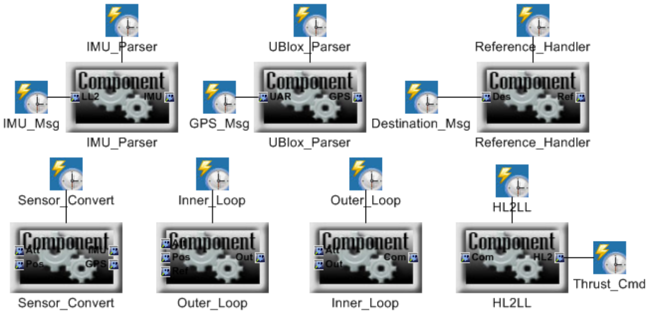

Figure 5. The timing model of each software component and communication channel.

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 and OChan 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).

E. Timing model

The global period (and deadline) is 1 ms, as that is the fundamental sampling rate of the hardware. The worst case latency from sensors to actuators must be smaller than the given timing requirement. In Fig. 5, each component is assigned a *TTExecInfo* (Time-Triggered Execution Information) object, so is each external data transfer. The processor-local data transfers transfer time is neglected, as reads and writes occur in locally shared memory. The quadrotor is controlled at 1kHz, so the *ExecPeriod* attribute for all components except for *Blox_Parser* is 1 millisecond. Because the time between two valid data of GPS is longer than 1 millisecond, the *ExecPeriod* for *Blox_Parser* is 100 milliseconds. Local message transfers may be specified as time-triggered, but in practice they taking place in shared memory are not scheduled. In ESMoL only distributed messages may be scheduled.

F. 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 [6].

Stage 2 provides several interpreters, each of which uses the UDM model navigation API to translate the ESMoL_Abstract model into either code or an analysis 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 the 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. The functional code implements the individual component behaviors, and the wrappers implement the component interactions.

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.

IV. Evaluation

We emperically evaluate the execution time for each component using an external indicator, and timing requirements of these components are met. Each of them takes less than 10 us during normal operation.

The memory system consists of 256KB on-chip flash memory (ROM) and 32KB SRAM (RAM). The corresponding binary code is about 130KB, so it fits in the system's ROM space. We also emperically evaluate that the RAM space is enough for data during normal operation. Since all the data variables for the communication are preallocated, this memory evaluation will not change in any cases.

V. Future work

Our future work includes: 1) multiple quadrotor helicopters control approach design and corresponding embedded software design 2) distributed experiments 3) fixed-point function generation 4) time-triggered wireless network.

References

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

[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)*, Oct 2003, pp. 19–24.

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

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

[5] E. A. Lee, "Embedded software," *Advances in Computers*, vol. 56, 2002.

[6] J. Porter, G. Hemingway, H. Nine, C. vanBuskirk, N. Kottenstette, G. Karsai, and J. Sztipanovits, "The esmol language and tools for high-confidence distributed control systems design. part 1: Language, framework, and analysis," Inst. for Software Integrated Sys., Vanderbilt University, Tech. Rep. ISIS-10-109, 2010. [Online]. Available: http://www.isis.vanderbilt.edu/sites/default/files/ESMoL_TR_opt.pdf

[7] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[8] 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.

[9] "Asctec hummingbird with autopilot user's manual." [Online]. Available: http://www.asctec.de/assets/Downloads/Manuals/AscTec-Autopilot-Manual-v10.pdf