Hardware-aware model-driven software development

Simon Barner[*], Christian Buckl[†],
Alois Knoll[*1]

[*] *Dept. of Informatics VI, TU München, Boltzmannstr. 3, 85748 Garching, Germany*
[†] *Fortiss gGmbH, Guerickestr. 25, 80805 München, Germany*

ABSTRACT

Acquisition of sensor data and generation of actuator control signals are two key properties of embedded systems. It is in the nature of these tasks that their realization is hardware-dependent and involves low level program code that is usually implemented manually. Typical requirements in this context are the actual implementation of device specific protocols and buffering strategies for efficient yet safe hardware access. In this paper, we argue why it is beneficial to consider these low-level components of embedded software in a model-driven development process that at first sight aims at raising the level of abstraction to foster more efficient and correct design and implementation. In particular, we investigate by means of a representative example application how I/O handling (including interrupt processing) can be integrated into a *hardware-aware* model-driven software development process.

1 Introduction

Control systems can be subsumed as systems that process input data and generate corresponding control output signals. While such systems are traditionally implemented using proprietary and costly PLC systems, microcontroller-based embedded systems are more and more considered as a low priced and flexible alternative.

Since manual programming of such systems is tedious and error-prone, we propose a *hardware-aware* model-driven software development process that in our opinion offers several advantages. While on the one hand it raises the level of abstractions and fosters the reuse of existing components, it also allows the developer to access specific properties of the underlying hardware platform.

The rest of this paper is structured as follows: In section 2, we point out the benefits of hardware-aware model-driven software development where the developer can access

[1]E-mail: barner@in.tum.de, buckl@fortiss.org, knoll@in.tum.de

platform-specific features using the same metaphors as for higher levels of abstraction. Subsequently, we present our current experimental setup in section 3. Finally, we summarize and present our future research goals in section 4.

2 Hardware-aware software development

In our development approach, several modeling languages can be used to describe the behavior and properties of an embedded control or signal processing system. A prototype has been implemented in the *EasyLab* tool [BGBK08] that will briefly be presented next.

Application model. This set of languages is a *functional* and *temporal* specification of the control application and currently contains two modeling languages.

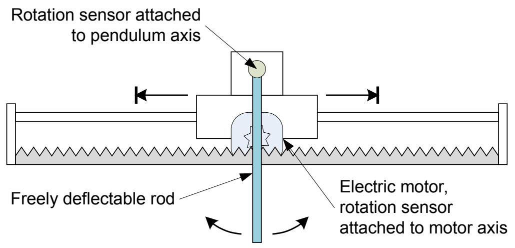
The *SFG* language is an adapted version of the Sequential Function Chart (SFC) language of IEC 61131-3 [Int03] and describes program states and transitions between them. Let $\mathcal{S}$ be the set of states in the SFG. Every SFG state $s \in \mathcal{S}$ is annotated with a worst-case deadline $\mathrm{d}(s)$ and may have one reference to a sub-model that is being executed while the program is in that state. States are composed using sequential, alternative, parallel and jump operators. There is exactly one designated initial state. State transitions are guarded by *transition conditions*

The *SDF* language is similar to the Function Block Diagram (FBD) language of IEC 61131-3 [Int03]. An SDF model is a directed multigraph where each node depicts an instance of a certain actor type and edges denote the data-flow between actor instances. An actor type is defined by a set of typed input and output connectors as well as internal state variables. SDF (sub-)graphs can be subsumed to yield composed actor types. Each actor type has a set of associated actions for the events *start*, *step* and *stop*. For (multi-rate) SDF graphs, the execution order of actor instances can be statically determined if the graph meets certain structural restrictions [BBHL95].

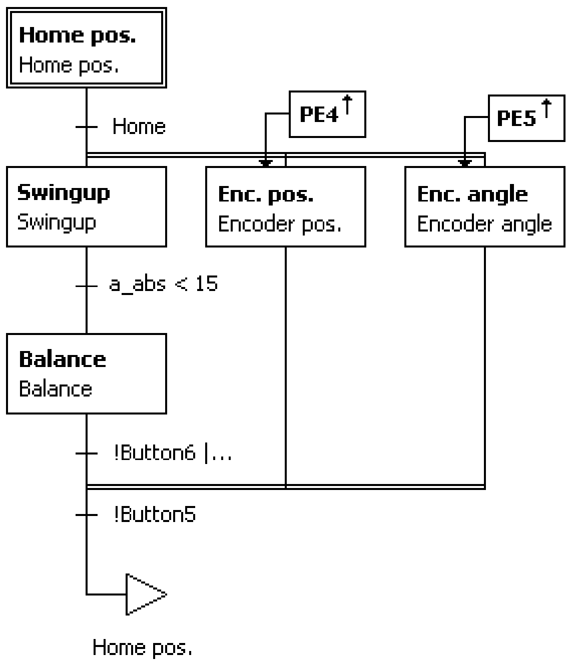
Hardware model. While model-driven development is usually employed to abstract from hardware, the domain of control and signal processing system often poses the requirement to access hardware-dependent aspects of a system. In our approach, the *hardware model* is used to describe the execution platform. It covers controller hardware such as the underlying MCU, as well as the sensors and actuators attached to the system and also provides a mapping of hardware features to corresponding representatives in the application model.

On the one hand, this mapping comprises device-specific actors types that can be instantiated in SDF models. Due to the execution semantics of SDF models, this approach is only suitable for synchronous hardware access (a.k.a. *polling*). Hence, code generators in model-driven approaches are often supported by hardware abstraction layers (HALs) that hide implementation details such as buffered access to devices requiring interrupt-driven data-exchange behind a high-level API. Since HALs are typically implemented manually, there is a break in the development methodology which complicates the integration of unsupported or custom hardware. The hardware model also contains *triggers* as well as *(input and output) buffers* that allow the developer to refine models with low-level implementation aspects, e.g. device-specific communication protocols.

A *trigger* can be used to specify the system's reaction to internal and external events such as timer ticks or external interrupts on GPIO pins. Consequently, the specification of a trigger


Rotation sensor attached to pendulum axis
Freely deflectable rod
Electric motor, rotation sensor attached to motor axis

(a) Overview



(b) SFG-model with 2 interrupt-driven branches.

Figure 1: Experimental setup: inverted pendulum

consists of a condition and a referenced *handler* that can be specified using the languages provided by the application model. For the actual specification of triggers, *SFG*'s notion of parallel branches can contain event-driven branches that are triggered a condition and that preempt the current thread of execution. *Buffers* are used to decouple the application logic and the process environment in the case of asynchronous (e.g., interrupt-based) hardware access protocols. When an interrupt is triggered, the interrupt handler performs its tasks on a set of private variables that are synchronized with the device's (public) buffer variables when the application is in a consistent state, i.e. before and after transition conditions in the *SFG* model are executed.

Code generation. A template-based *C* code generator can be used to automatically generate implementations for different hardware architectures. Here code templates are assigned to primitive modeling elements such as actor types or device buffers. Triggers are implemented using interrupt service routines (ISRs) and require code templates using appropriate compiler specific statements. The implementation of the actual handler code is based on existing strategies for *SFG* and *SDF* models.

3 Experimental setup

In this section, we will demonstrate how the concepts proposed above can be used for the entirely model-driven development of the control application for a self-erecting inverted pendulum. The experimental setup consists of an electric motor that drives a cart mounted on a linear rail and a freely deflectable rod that is attached to the cart (see figure 1(a)). A suitable controller first heuristically swings the rod back and forth until it is almost in upright position. Then, the controller switches to the balancing state who's PID controller can be derived from a linear state-space model of the pendulum. Both controllers use the inclination angle of the rod $\alpha(t)$ and the position of the cart $c(t)$ as inputs. Each of these values is measured by a quadrature encoder which generates a signal consisting of two digital channels A and B. An interrupt-driven method to analyze the encoder signal is to trigger a driver program on rising edges of channel A that evaluates the value of channel B.

Figure 1(b) shows an EasyLab model implementing the controller for a self-erecting inverted pendulum. It consists of the actual control task (initialization, swing-up, balancing and cleanup), two triggers to implement the evaluation of the quadrature encoder signals and two devices enc_{pos} and enc_{ang} representing the encoders (not displayed here). Using the position encoder as an example, a rising edge on pin `PE4` (channel A) triggers the execution of the `Encoder position` subprogram. In the current implementation, our setup is controlled by a 8-bit microcontroller (ATmega128 running at 16 MHz) which performs the entire control task: interrupt-based evaluation of quadrature encoder signals, computation of the control value and generation of actuator signals. Although the interrupt rate for the position encoder reaches 25 kHz when the pendulum performs rapid balancing movements, the period times are comparable to a manually implemented evaluation of the encoder signals (swing-up: 400 Hz, balancing: 1.1 kHz).

4 Conclusion and future work

In this paper, we introduced the notion of *hardware-aware* model-driven software development where *triggers* and *handlers* allow explicit modeling of event-driven parts of control applications. The motivation was to free the developer from the burden to manually implement these low-level aspects. An interrupt-intensive control system for an inverted pendulum demonstrates the applicability of the approach.

As part of our future work, the experimental setup is ported to a multi-core control system (implemented in a FPGA using appropriate soft-cores) that will additionally be equipped with a camera. Our primary research interest is the challenge of integrating the high throughput requirements of a vision-based tracking system with the low latency constraints of the interrupt-driven parts within our model-based development approach.

References

[BBHL95] Shuvra S. Bhattacharyya, Joseph T. Buck, Soonhoi Ha, and Edward A. Lee. Generating compact code from dataflow specifications of multirate signal processing algorithms. *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, 42:138–150, 1995.

[BGBK08] Simon Barner, Michael Geisinger, Christian Buckl, and Alois Knoll. EasyLab: Model-based development of software for mechatronic systems. In *IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications*, Beijing, China, October 2008.

[Int03] International Electrotechnical Commission. *Norm EN 61131*, 2003.