EasyLab: Model-Based Development of Software for Mechatronic Systems

Simon Barner, Michael Geisinger, Christian Buckl and Alois Knoll
Department of Informatics
Technische Universität München
Garching b. München, Germany
{barner,geisinge,buckl,knoll}@in.tum.de

Abstract—Model-based development tools are one possible solution to handle the increasing complexity of mechatronic systems. While traditional approaches often separate design of hardware and software, especially in mechatronic systems hardware/software interaction is the most critical component. Hence, both aspects must be considered in this context. The goal is a model-based development tool for software/hardware co-design including the generation of efficient code for the respective hardware. EasyLab is a modular and easily expandable development tool that is especially suitable for such applications. Its objectives are to facilitate reusability and to accelerate the development process. It raises the level of abstraction and thus simplifies the development of mechatronic systems even for unexperienced users. A graphical user interface provides various modeling languages that are easy to use. By employing platform optimized generation of the code, efficiency of the resulting programs can be guaranteed, which we demonstrate on a set of experimental mechatronic systems.

I. INTRODUCTION

Mechatronic and embedded systems are becoming increasingly complex. While sophisticated tools for the development of the mechanical and electronic parts are available, the implementation of the software is typically done from scratch. Due to shortened product life cycles and an emerging need for flexibility, this approach is not feasible anymore. Development tools for hardware/software co-design are required that raise the level of abstraction to accelerate the development process, but guarantee an efficient and reliable implementation to match the resource constraints of mechatronic systems.

For standard software, model-based development [1] has become state of the art in software engineering. Several model-based development tools, such as Matlab/Simulink [2] or Scade [3] are available for the domain of embedded systems. However, these tools rely on the generation of ANSI-C code and therefore can only be used to implement the application functionality. Code for hardware related aspects and other non-functional properties has to be manually implemented by the developer. This code however forms the majority of the code required for mechatronic systems. The major reason why the generation of such code is not supported is the platform dependency of the code. Due to the vast heterogeneity of hardware [4], it is not possible to implement a code generator that supports all possible hardware platforms a priori. Rather, a suitable development tool must be designed in a way that it can be easily expanded to support further platforms.

This paper presents the development tool EasyLab that targets the development of mechatronic systems. The main contributions of this work are a tool with a high level of abstraction that simplifies and accelerates development, a fully modular design to support reusability, a completely integrated solution for hardware/software co-design, and the possibility to generate efficient, hardware-dependent code.

The developer benefits from the abstraction of a graphical development interface similar to widespread graphical environments such as Matlab/Simulink or LabView. In contrast to existing tools, EasyLab allows the specification of hardware characteristics and generation of corresponding code. The tool is expandable in two dimensions: regarding the modeling and the code generation functionality. Expandability with respect to the modeling functionality is achieved by relying on actor-oriented design [5]. An actor is a software component and can for instance realize a PID controller, but also the triggering of a sensor or actuator. Expandability with respect to the code generation functionality is achieved by a template-based approach [6]. The main difference between template-based and component-based [7] approaches is the high adaptability of templates. It is therefore possible to generate very efficient code [8].

The outline of this paper is as follows. First, we introduce the Match-X construction kit, which is one of the main target platforms of EasyLab. The main part of the paper covers design and implementation of EasyLab. We introduce two graphical programming languages as well as other key features of the application, such as code generation and simulation. To illustrate the software modeling process with EasyLab, we present two experimental setups that were built during the development of EasyLab. Finally, we list related and future work and summarize the goals and results presented in this paper.

II. HARDWARE

EasyLab is designed to support different hardware architectures with a focus on resource-constrained systems. To demonstrate the potential of our approach, a modular hardware architecture is preferable. We therefore chose the Match-X construction kit, which is explained in the following, as a reference platform.

A. The Match-X Construction Kit

The Match-X construction kit consists of modular components that can be flexibly assembled into complex mecha-
tronic systems. The construction kit forms the first stage of hardware development and provides a way to efficiently prototype such systems. The development process along with mechanical and electrical interfaces of the hardware building blocks are specified in a standard of the VDMA\(^1\) [9]. The standard also describes the transition to small batch production as well as series production. Figure 1 shows the geometry of a building block from the Match-X construction kit. Figure 2 shows a completely assembled stack of three building blocks. The block at the bottom contains a voltage regulator and features an RS485 interface, the block in the middle contains a Microchip PIC18F2520 microprocessor and the topmost block allows attachment of sensors and actuators. Although the size of these building blocks is very small, the microcontroller is clocked with 20 MHz and is thus suitable to perform complex controller tasks.

Fig. 1. Geometry of a building block from the Match-X construction kit

**B. Other Target Platforms**

To point out the universality of this approach, EasyLab does not only focus on the Match-X hardware construction kit, but also on a variety of other microcontroller platforms. In this paper, we show the results of an evaluation where we used EasyLab to design the application logic for a complex control task running on an ATMEL ATmega128 microprocessor. We also plan to support other microprocessor types like ARM and Fujitsu processors.

**III. EASYLAB**

EasyLab provides a high-level programming environment for the design of software for mechatronic systems. Due to the raised level of abstraction, even people inexperienced in microcontroller programming can develop complex applications using various graphical modeling languages. The design process is based on the selection of functional components (actors) that can be connected to form the application. One can differentiate between two different types of actors: hardware dependent actors, e.g. a software component controlling a sensor, and hardware independent actors, e.g. basic mathematical operations or more complex ones like various controllers. As model of computation, the synchronous data flow model is chosen as it reflects the typical engineering approach. However, a pure data flow model is too inflexible to express typical application behavior. This issue can be targeted by the combination of different models of computation (modal models [5]). EasyLab combines state machines with data flow graphs to allow an easy and intuitive modeling of the application. Based on the models, the tool allows both simulation and the generation of efficient code. The core features of EasyLab are explained in the following.

**A. Graphical Modeling Languages**

EasyLab supports two graphical modeling languages:

1. **Structured flow chart**: The structured flow chart language (SFC) describes the states of a program and how state transitions are performed. The language has been designed in the style of EN 61131-3 [10] (part “SFC”).

2. **States of SFC programs are references to sub-programs that can be described in any of the available languages. Thus, a state is either an SFC program itself or a reference to an SDF program (see below) which is executed periodically. Consequently, SDF programs are the leaves in the recursive specification of an SFC program. An SFC program has exactly one initial state.

Elements in the SFC language that determine the control flow are state sequences, alternative branches (conditional execution), parallel branches, jumps and program termination.

Both sequential and alternative composition are based on conditions that are defined as Boolean expressions consisting of Boolean constants and variables as well as comparisons.

\(^1\)The German Association of Machinery Manufacturers.
Comparisons contain arithmetic expressions composed of functions and operators on constants and global and local variables. Local variables refer to values computed right before the respective condition.

Figure 3 shows two example programs in SFC language.

![Diagrams]

Fig. 3. Example programs in SFC language: (a) alternative branches with Boolean conditions $c_1$ and $c_2$, (b) parallel branches; $t_1$, $t_2$ and $t_3$ are Boolean transition conditions.

b) Synchronous data flow: The synchronous data flow language (SDF) consists of a directed multigraph where each node is the instance of a certain actor type. Edges in the graph 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. Furthermore, the actions start, stop, step are defined for each actor type which are used for initialization and finalization as well as to define the actor’s effect. For code generation and simulation, actions are implemented as code templates and shared libraries respectively.

To support reuse and abstraction, actor types can be defined using hierarchical composition. Input and output connectors of a composed actor type $A_c$ are mapped to distinguished connectors (named input and output interfaces) of an embedded SDF graph defining the step action of $A_c$. Output interfaces can be used as local variables in SFC transition conditions as pointed out above.

Variables, namely input/output connectors and state variables, are typed to ensure that only compatible interfaces can be connected. To disambiguate, there may be at most one edge connected to each input connector.

A set of predefined primitive data types as well as higher user-defined data types like arrays and structures are available. A template mechanism allows for efficient definition of polymorphic operations (e.g., arithmetic operations), while a type coercion algorithm guarantees the generation of correct and memory-optimal code [11].

The semantics of the SDF language is as follows: When a graph is executed for the first time (i.e., the corresponding SFC state has been entered), the actors’ start actions are executed (in arbitrary order). While the corresponding SFC program remains in the same state, the actors’ step actions are periodically executed according to a static schedule. Accordingly, when the SFC state is left, the stop actions are executed (in arbitrary order).

Various algorithms have been found that statically compute valid schedules for multi-rate synchronous data flow graphs (if they exist) and have bounded buffer memory requirements [12], [5]. Depending on the structure of the graph, single appearance schedules can be found to minimize the amount of code [13], [14] and buffer sizes [15].

The SDF language is designed in the style of EN 61131-3 [10] (part “FBD”). We will present an example program in SDF language in figure 5 later in this paper.

B. Hardware Model

A key feature in mechatronic systems is the interaction between software and hardware. However, the interface between these components and of course also the hardware itself varies between different systems. This means that designing applications for a mechatronic systems can only succeed if the modeling tool also allows modeling of the specific hardware that is being employed in the respective systems.

Hence, EasyLab provides a device type library that specifies which actor instances may be used in a certain hardware environment while the integrated resource management restricts the user to the set of hardware resources that are actually available.

The device descriptions also specify how hardware resources are represented in the application logic and how these representations are mapped to the real hardware. As for actor types in the SDF language, device types are defined using actions start, stop and step specifying the behavior when the application starts, terminates or when a single step in the SFC program is performed. To guarantee correct operation, access to the underlying hardware is buffered using a set of input and output variables.

C. Code Generation

EasyLab features an integrated code generator that transforms the application model into code suitable for the respective compiler. This is achieved by assigning a code template to each primitive element in the respective modeling language. Currently, models are transformed into C code (supporting the mcc18 and avr-gcc compiler tool chains).

In the generated code, each state and transition condition of an SFC program is represented as a function performing an action and returning the address of the function to be executed next. References to sub-programs have the effect of executing the respective program and returning a fixed function to be executed next (usually the subsequent transition condition). Functions representing transition conditions have no side effects and evaluate the respective Boolean expression.
network connection using appropriate device plugins (see R
control the mobile robot platform Robotino
perform control tasks directly in simulation mode.
Therefore, we added another layer in between the code
templates and the actual hardware: a set of platform-specific runtime libraries.
Each runtime library provides basic hardware-related functionality that is common among almost all types of microprocessors (digital and analog I/O, communication via UART, etc.) as well as some software-only features (data structures optimized for low resource usage, fixed-point arithmetic, etc.). The runtime libraries for all target platforms implement a common interface that is designed to allow implementation of a subset of all possible features, if some features are not supported by the respective microprocessor.

After code generation, the runtime library corresponding to the selected microcontroller is linked against the generated program. This allows code templates to use the common interface exposed by all runtime libraries, hence making them much more compact and less error-prone. Furthermore, this design will improve the possibility to optimize the code contained in the code templates because they have less dependencies on the actual hardware and are formulated on a higher level of abstraction.

Actually, EasyLab’s runtime library offers at least a subset of the functionality of operating systems for resource-constrained platforms (e.g., Contiki [16], TinyOS [17] and FreeRTOS [18]). With these systems, application code is developed against a given API and a firmware image is obtained by linking both the user-supplied and the operating system’s code into a monolithic firmware image. In fact, EasyLab’s modular approach (see section III-F) allows for the implementation of an alternative runtime library on top of existing operating systems like those mentioned above.

E. Simulation
Besides the generation of code, EasyLab also features direct simulation of models (i.e., without code generation). In order to simulate as much of the mechatronic system as possible, EasyLab simulates both application code and hardware devices. While simulation can be used to detect design errors early in the development process or if the hardware is currently not available, it is also possible to perform control tasks directly in simulation mode.

In this context, EasyLab has successfully been used to control the mobile robot platform Robotino® over a wireless network connection using appropriate device plugins (see next section). A line follow application that is based on the robot’s camera and several image processing actors has been used to verify the performance of the simulation component.

F. Expandability
The application is built up in a modular way to ensure reusability of programs developed for one target application in other projects. A developer may expand the functionality of EasyLab in the following dimensions:

- New actor types can be added to the actor type library. For each actor type, an annotated code template as well as a simulation plugin have to be specified. Actors can be reused in any project developed with EasyLab.
- New hardware models may be added to the device type library. This allows a developer to add completely new combinations of hardware components. This is especially useful for the Match-X construction kit, where hardware modules may be combined in many ways. Hardware-specific actor types can be configured to be only available if a certain device instance is added to the project.
- New compiler tool chains may be added. This is necessary if a new type of processor is to be used as target platform for EasyLab. Tool chains specify which external programs are used to build the application and transfer it to the target device. They also influence how the code is generated. As this approach may not provide enough flexibility for all situations, a developer also has the choice of implementing the requested functionality as part of the EasyLab runtime library that is then linked to the project according to the selected tool chain.

IV. Evaluation
Two sample applications were implemented to demonstrate the benefits of the developed tool, namely the modular design of the system with respect to both hardware and software and the efficiency of the generated code.

A. Pneumatic Cylinder
The first evaluation scenario involves a pneumatic cylinder that is controlled by a microcontroller from the Match-X construction kit. The cylinder has two magnetic valves for expanding and retracting a piston. Furthermore, an analog sensor measures the current position of the piston. Figure 4 shows the experimental setup. The Match-X stack used in this experiment consists of the following building blocks: voltage regulator, CPU, A/D converter (for analog sensor input) and two drivers for inductive loads (one per valve).

The control task that should be achieved is to move the piston to a predefined position and to hold that position even if some force is applied to the piston. Since no proportional valves where available at the time of this writing, the corresponding control program is quite simple.

For all hardware sensors and actuators, adequate software actors are available in EasyLab. Figure 5 shows the program as an SDF graph. The constant pos defines the set point and
tol gives a certain tolerance for the position of the piston to avoid oscillation. The controller regulates the piston to a position in the interval $[\text{pos} - \text{tol}, \text{pos} + \text{tol}]$.

B. Inverted Pendulum

The second experiment, a more complex control task points out the efficiency of the generated code. The inverted pendulum is a well-known experimental setup to demonstrate control tasks [19]. It consists of an electric motor that drives a cart mounted on a linear rail. A rod that can be freely deflected is attached to the cart. Figure 6 shows the experimental setup.

The experimental setup features two sensors and one actuator: one sensor measures the position of the cart, the second one measures the inclination of the rod. The only actuator is the electric motor, for which we regulate the voltage.

To demonstrate the efficiency of the code generated by EasyLab, we implemented the system using an ATMEL ATmega128 microcontroller with a clock frequency of 16 MHz. First, we augmented EasyLab to allow us to access the sensor values necessary to achieve the task. For this purpose, we added a new actor type that returns the current value of the respective sensor as well as its derivative.

A suitable controller for a self-erecting inverted pendulum consists of two modes (swing-up and balance), which are implemented in terms of the SFC program depicted in figure 7. The controller used to erect the rod is based on a heuristic using a proportional-velocity cart position controller that swings the rod back and forth until the rod has a maximum deviation of $\varepsilon$ to the upright position (first state in SFC program and subsequent transition condition). The second state corresponds to the execution of the balance SDF program according to equation (1) (see below), which can be derived from a linear state-space model of the pendulum using the linear quadratic regulator (LQR) technique. While the balance state is never left, the final jump was added to obtain a well-formed program.

For the $i$th step, let $\alpha_i$ be the inclination of the rod (in radians), $\partial \alpha_i$ the angular velocity and let $\alpha = 0$ if the rod is in erect position. Let $c_i$ be the position of the cart, $\partial c_i$ its velocity and $c = 0$ if the cart is in its center position. Then a PID controller used to balance the pendulum in upright position can be defined as follows, where $U_n$ is the voltage to drive the motor with at step $n \geq 0$ (constants $k_1, \ldots, k_6$ depend on the physical characteristics of the rod):

$$U_n = k_1 \cdot \alpha_n + k_2 \cdot \partial \alpha_n + k_3 \sum_{i=0}^{n} \alpha_i + k_4 \cdot c_n + k_5 \cdot \partial c_n + k_6 \sum_{i=0}^{n} c_i$$

(1)

The transformation of above formula into an SDF program is straightforward.
V. RELATED WORK

EasyLab is unique in the sense that it supports hardware/software co-design without focusing on a specific set of hardware platforms. It is expandable both regarding modeling and code generation.

Several other tools and projects influenced the development of EasyLab. The graphical user interface is in the style of state-of-the-art tools like Matlab/Simulink [2] that also provide data flow design based on actors. The theory behind actors is however derived from the project Ptolemy [5]. In contrast to both tools, EasyLab augments the notion of actors by introducing hardware related actors which is a precondition for successful hardware/software co-design. The introduction of typed data flow allows for the generation of code tailored to resource-constrained target architectures.

EasyLab uses synchronous data flow graphs and structured flow charts as model of computation, as defined in EN 61131-3 [10]. While there are several other tools such as CoDeSys that are compliant with this norm, EasyLab is the only one that can be augmented for a specific hardware platform and can thus generate optimized hardware-dependent code. The concept for code generation is based on previous work on template-based code generation [6], [20].

VI. SUMMARY

In this work, we presented a new model-based programming tool for mechatronic systems. In contrast to well-known tools from the world of embedded systems, EasyLab also provides components that are specialized for mechatronic systems, where the diversity of the used hardware is much more relevant than in other types of embedded systems.

We have also shown that EasyLab allows both modeling of software and hardware functionality and introduced the two graphical modeling languages that have been implemented so far, namely synchronous data flow (SDF) and state flow charts (SFC).

Finally, we presented two demonstration setups that showed that the workflow of programming mechatronic systems is the same for different types of microprocessor platforms. The inverted pendulum setup demonstrates that the code generated by EasyLab performs well and is suitable for real-time control tasks.

VII. FUTURE WORK

We are currently working on a remote debugging facility for programs generated by EasyLab whose purpose is to provide transparent real-time access to the state of a mechatronic system at the granularity of the underlying model. It is based on code instrumentation and a marshaling layer that is tailored to the resources and communication interfaces that are available in typical embedded systems. Thus, firmly integrated inspection and manipulation of both data and program state at the model level will increase the abstraction also in the debugging phase of the product development cycle.

We also plan to extend EasyLab to support networked (distributed) mechatronic systems. The time-triggered model [21] seems to be a feasible approach that integrates well with the synchronous data flow employed in the current stage.

VIII. ACKNOWLEDGMENT

This work is funded by the German Ministry of Education and Research.

REFERENCES