Multi-view Methodology for the Design of Embedded Mechatronic Control Systems

Marcel A. Groothuis, and Jan F. Broenink, Member, IEEE

Abstract—The design of embedded control systems for monitoring and control of mechatronic systems has a multidisciplinary development trajectory. These systems consist of heterogeneous components developed by different disciplines (control engineering, electrical engineering, software engineering and often many more). The design trajectory needs therefore a multi-disciplinary design methodology that enables concurrent design and interactions between all involved disciplines, reducing inconsistencies and conflicts that occur during the design phase. This paper proposes a multi-view methodology to address the above-mentioned issues. The main purpose is to shorten the design time and at the same time increasing the reliability and predictability of embedded (computer) control systems.

I. INTRODUCTION

The successful design and realization of control systems for mechatronic applications is an integrated effort of several technical disciplines.

A common practice in multi-disciplinary design is the design flow indicated in Fig. 1a. Each discipline uses its own design flow, terminology, models and domain-specific tools to develop their parts of the system. Furthermore, different disciplines regularly share the same terminology, but the used terms have often a different meaning. This can cause potential misinterpretation problems in the interaction between developers of different disciplines. Concurrent engineering is often based on global agreements on communication, during their design trajectory. The boundaries of their discipline in a more formal way than the development phase. It is well-known that this separation causes high development costs due to long feedback cycles [1]. Design decisions made by one discipline during the component design, have a direct effect on the validity of the decisions taken by others. Since a common understanding, modeling, and design framework is missing, keeping track of these effects and understanding them is very difficult.

The development of such systems requires a good integration of multiple engineering disciplines, such as mechanical engineering, electrical engineering, control engineering and software engineering.

The required change of the design flow, when incorporating these two aspects, is shown in Fig. 1b.

Industrial demands on near future embedded software and hardware are more flexibility, higher reliability, shorter design time and higher quality. These are a kind of non-functional properties, which must be incorporated in the design methodology.

In addition, the complexity of products is increasing rapidly, mainly caused by the increasing processing power of these embedded computing systems, which allows more complex information processing and more functionality at relatively low cost. However, this increasing functionality also increases the complexity of the design process for such a system. This puts extra demands on the design methodology to prevent increasing costs for testing.

This paper describes a multi-view methodology for the design of embedded (control) software for mechatronic systems. It is an attempt to create a design flow, which supports both system design and multi-disciplinary component design via an integration platform that helps developers of different disciplines to communicate across the boundaries of their discipline in a more formal way than social communication, during their design trajectory. The main goal is to discover the inconsistencies and conflicts created by design decisions in other disciplines in an earlier stage of the development process.

This paper is organized as follows. The next section describes the background for the multi-view methodology and the current state of the art for embedded control software design. Section three describes our multi-view methodology and section four describes a multi-view integration layer that is needed to support the methodology.
II. BACKGROUND

Before discussing the multi-view design methodology, first a more global approach about designing embedded control software is presented, which is the starting point from which our methodology is set up. In addition, some relevant existing (support) tools are discussed.

A. Embedded Control System Design

Typical steps in an embedded control system design process are (see Fig. 2):
1. Translation of the system idea into requirements.
2. Partitioning the system into components to handle complexity.
3. Physical System Modeling, i.e. model the plant parts of the embedded control system (dynamic behavior).
4. Control Law Design, using the models obtained in the previous step. Model reduction is necessary to get a model of adequate detail.
5. Embedded Control System Implementation: the control software is designed via refinement of the control law.
6. Realization of the embedded software via an ongoing refinement process. Models of the components are replaced by the real system parts in a stepwise manner (both for hardware, plant, and software [2]).

Fig. 2: Embedded system design process

This approach is model-based. Besides code generation (for the software part), it also gives developers a more precise language to express and discuss their ideas. Obviously, documentation also becomes easier and more consistent when models are used.

B. Existing Methods and Tools

Many existing commercial and research tools exist for embedded control system design. A short summary is given of some of the relevant representatives for the involved areas of engineering. Also relevant tools and methodologies that meet some of the requirements for concurrent engineering will be mentioned.

1) 20-sim, gCSP, CCE

The foundation of 20-sim [3] lies in area of control engineering and dynamic systems modeling. Besides the common IPM (ideal physical model) diagrams and transfer functions, 20-sim also supports the domain independent bond-graph notation for modeling dynamic systems. 20-sim supports multi-disciplinary modeling with library components for many engineering disciplines. It supports model checking and has analysis techniques to test the stability of designed controllers.

gCSP is our graphical modeling language for the design of parallel software based on the CSP (Communicating Sequential Processes) process algebra [4],[5]. It allows formal checking of the designed software to prevent deadlocks and live locks, resulting in software that is more reliable.

Both 20-sim and gCSP support template-based code generation for realization of the designed control software for many targets.

The coupling with I/O hardware is kept outside the models in both tools, to guarantee that the ECS implementation indeed means a refinement from control law to code. This means that there is no manual code adaptation. This part is covered in our Command and Control Environment (CCE) tool chain, which features rapid prototyping with real-time data logging [6], including Hardware-In-the-Loop (HIL) simulation capabilities.

2) Matlab, Simulink, StateFlow, RealTime Workshop

Simulink is a simulation and model-based design environment, similar to 20-sim. Matlab provides its mathematical foundation. Simulink has fixed-point extensions useful for embedded CPU’s without floating point unit. Stateflow can be used to extend Simulink models with event-based models. Automatic code generation and real-time rapid prototyping is supported with Real-time Workshop, Stateflow and XPC targets. Code-generation for FPGA based hardware is also supported via the commercial extensions of Xilinx and AccelChip.

Simulink models, used for code-generation, contain information about the connections to hardware. This limits the refinement possibilities and combination with code generated from other tools. Furthermore, switching to other target hardware requires changes in the higher-level Simulink models. Cooperative testing in combination with other tools is possible and these extensions are mainly provided by external tool vendors. An example is co-simulation between Simulink and Xilinx System generator.

3) Dymola, Modelica

Dymola is a multi-disciplinary simulation and model-based design environment based on the object-oriented modeling language Modelica. Dymola features mixed-mode integration techniques for HIL simulation, an open interface for cooperation with other tools, support for bond graph modeling, code generation and many more.

4) Ptolemy II

Ptolemy II [7] is a software framework supporting heterogeneous concurrent modeling and design, with toolboxes for multiple disciplines. The focus of Ptolemy is mainly on hybrid modeling and simulation. It provides the user with various models of computation (MoC) and hence allows different domains to be described in the same language. Ptolemy II allows the user to combine partial models of different kind into a common heterogeneous model by using hierarchical nesting. Ptolemy II supports currently basic code generation for specific processors.
5) Toolnet

The TU Darmstadt Toolnet [8] project is working on a tool integration framework, which realizes data integration and data consistency. Their approach leaves the data at the tools instead of exchanging it with a common repository. The additional information that is stored to relate the tools are references and consistency relations. This allows specialized Toolnet verification tools to do centralized consistency checking. As far as known, the Toolnet approach does not provide cooperative testing possibilities like co-simulation.

6) Meta-modeling

Meta-modeling is an area closely related to modeling. The main activity is to produce models for a particular purpose. Common purposes are to provide a schema or a language to support the modeling process. Typical examples of metamodels are UML, XMI, MetaH, SysML and AADL. Many of these languages are used by system architects to describe the global system architecture. AADL is, for example, a graphical and textual representation language, similar to UML, for the description of real-time embedded systems. AADL can model component interactions, interfaces, relations, implementations and many more.

This list is far from complete, but in general, many existing tools and methods exist for embedded (control) system design. They have all their specific application domain and are often used in combination with other tools.

A general methodological point of interest is that, while the current commercial tools indeed provide automated code-generation, this generated code often needs to be enhanced by hand to cover all required functionality and to reach the required predictability and functionality. In multi-design stages, predictability refers to the capability to make design decisions based on accomplished work. These design decisions hold not only at the current design stage but also for later design stages. The integration of all (software) functionality modeled in various tools, can only be done at the end. This results in an unnecessary long design and development time.

The addition of reconfigurable and/or different types of computer hardware (FPGA’s versus microprocessors) makes it even more difficult to generate automatically the correct device drivers that are required for the interaction between software running on a CPU and the hardware implemented in the FPGA.

III. MULTIPLE VIEW METHODOLOGY

A. Introduction

Our methodology emphasizes the usage of models throughout the development process (model-based design). It consists of three components: views, multidisciplinary core models and correctness preserving code generation (see also Fig. 3). The purpose is to facilitate the multi-disciplinary design of embedded control software.

The philosophy behind the multiple views methodology is that each discipline can perform best using its own kind of system description (i.e. views and tools). Partitioning in components and multiple (sub) models is used during the design phase to manage complexity of the whole system. To allow integrated design and cooperation between these system descriptions, a translation to multidisciplinary core models is proposed. We start with the description of views and their relations.

Fig. 3: Multi-view approach

B. Views & viewpoints:

A view is a representation of a whole system or a part of a system from the perspective of a set of concerns [9]. Views are just like most systems modular. A view could contain one or more architectural models and can utilize multiple notations (e.g. bond graphs and iconic diagrams).

A viewpoint is a pattern to construct a certain view. It defines a set of rules (e.g. the language, consistency checks, patterns and analysis techniques) for the view.

Viewpoints are independent of the system, while views are specific for each system. An example: a bond graph can be used to model the dynamic behavior of a system. This bond graph view is specific to the system, while the bond graph editor (=viewpoint) is system independent.

A combination of this definition with the knowledge that different tools are involved during system design to produce these views results in four kinds of inter-view connections (see also Fig. 4):

- Compatible/translatable data relations:
  1. Multiple views within one tool on the same model (e.g. a bond graph and a IPM). They share one viewpoint.
  2. Same views in different (overlapping) tools (e.g. controller design view in 20-sim and in Simulink).
- Cooperative/non translatable data relations:
  3. Multiple views within one tool that show different information (CSP view and the continuous time view in Ptolemy).
4. Different views in different tools (e.g. a controller design view and an UML software design view).

C. Core Models

Besides views, viewpoints, tools and the relations between them, some lower level core models are needed to translate properties of one view into another view. Depending on the type of views and their languages, such a core model can be subdivided in one of the following categories:

1) Core Models with a Common Language

This is the most extensive form of a core model. This common language can be a mathematical relation or just a common (hidden) language. An example for the mathematical relation is the modeling of the dynamic behavior of a physical system. This can be done by using bond graphs [10, 11] or iconic diagrams. The representations are different, but the mathematical nature is the same: differential equations. Translation between these two views on the physical system is possible [12]. Another example: the transfer function, a pole-zero diagram and a bode-plot can be used to analyze the system response. These are three views within the system response viewpoint. Translation between these system descriptions can be performed in an algorithmic and automated way, used by many existing modeling tools [3].

An example of a common language core model is the internal 20-sim SIDOPS language, the Modelica language or the Matlab language for all Mathworks tools.

2) Model of Computation

A second group of core models is based on the used model of computation. For example time triggered, versus event driven nature. In this case, the underlying core models can be used to do consistency checking at the boundaries and to facilitate integrated testing. Co-simulation with time synchronization or hybrid simulation can be used here to test whether the combination of the time triggered part and the event driven part of the embedded software results in the required and specified behavior.

3) Shared properties and interfaces

The third group of core models is nothing more than a set of inter-view relationships that describe the shared properties or interfaces at the boundary between views (e.g. a meta model at the architecture level). Some examples:

- Traceability relations
- Is the data shared?
- How is the data shared?
- Main source of the data (parent-child relations)
- System components
  - What components comprise the system?
  - What are the roles of these components?
  - What are the interfaces to other components?
  - How are these interfaces connected?
- Testing
  - Common test patterns

The ultimate core model would be a language that has a single unifying semantic model that can be used to translate every view into every other view. However, at the moment this is not yet possible, but it is a ultimate research challenge. The extensiveness of the core-models depends on the similarities and relations between views.

D. Correctness Preserving Code Generation

The third step is the generation of the embedded software (and reconfigurable hardware) in a property-preserving way. The key to correctness-preserving code generation is to keep the behavior consistency between a model and its software realization. Traditional manual code ‘generation’ is carried out in an ad-hoc way, which largely relies on developers’ experience and often introduces errors during the transformation from the model to its realization. Automatic code generation can transfer the model into a realization based on predefined mapping rules (or strategies). These rules (strategies) ensure the behavior consistency between the model and the realization. This seems rather straightforward but it is shown in [13], [14] that existing (commercial) tools often generate code that does not necessary behave as modeled. Even code generated from formally checked models that are proven correct may behave different in the final realization.

Besides the property preservation requirement, code-generation should meet more requirements that are important:

- Code should be generated in a property-preserving way
- Real-time constraints must be met. Not completing computations within the specified time frame can result in a serious catastrophe [15] in the case of hard real-time constraints and loss of quality in other cases.
- The generated code should be efficient and resource friendly in terms of code size, run-time efficiency and resource usage. This is a design tradeoff between the use of more powerful embedded computing resources and the costs.
- Limitations of target hardware. E.g. it should be possible to model and generate code for fixed point targets.
- Software component generation. Existing synthesis tools are usually only available and effective for a single design discipline. When several disciplines are involved at the same time during the development, a set of synthesis tools is required to generate the final code. To
ensure that the generated code from different synthesis tools can be integrated seamlessly, the code generators should support component-based code generation, including automated generation of glue-code to interface code originating from other tools.

- Generation of device driver (interface) code for the communication with I/O hardware and communication links
- Reliable and safe (prevent errors and in case of errors, good exception handling)

Our focus is currently mainly on the system level interconnection between views. Core model translations within tools are often already provided, but relating views, and their properties, between tools requires a central integration layer with new tooling. A generic tool-independent co-simulation framework is needed to allow for testing and verification between views.

For the code-generation, our focus will be mainly on the property preserving part and on the gluing problems of code from different sources and automated coupling with hardware outside the existing tooling.

There is no single tool-suite that is sufficient for the entire development of a mechatronic system and also for multi-view integration are not many existing tools. Toolnet provides the first step with the tracing of information and the Tangram project [16] is working on multi-disciplinary testing methods and Ptolemy is strong in hybrid modeling.

IV. Multi-view Integration Layer

The previous section described the multi-view methodology. This section describes a multi-view integration layer that is needed for the coupling of tools/view to core models. The required multi-view integration layer is shown in Fig. 5. It will be explained with an example of a mechatronic system: a robotic arm with three degrees of freedom [17].

In this example, the modeling and simulation package 20-sim (tool 1) is used in combination with the graphical CSP tool (tool 2).

20-sim is used to model the plant dynamics, quantization effects of the used I/O hardware, for the controller design and for simulation. The available code generation facility is used to get the (time triggered) software implementation of the final control law.

The second tool, gCSP, is used to model an event driven software framework with parallel running processes. Three of these processes contain the controllers designed within 20-sim and the fourth process contains a safety layer that protects the plant for outputs that will move the robotic arm outside its working area. The CSP background allows us to use a formal checker tool (e.g. FDR [18]) to check for possible deadlock, live lock or starvation situations, resulting in software that is more reliable.

This example shows a direct connection between two disciplines (control engineering and software engineering), three tools and even more views (software, formal checker result, controller, bode diagrams, pole-zero diagrams, plant dynamics). The generated controller code from tool 1 (controller code) should be inserted in the generated code from tool 2 (event driven software framework).

The core models in this example are the internal language of 20-sim (SIDOPS) and CSP. Translation is not possible in this case due to the different models of computation. The coupling between the two models in this example is via insertion of generated code, the required sample frequency for the controllers and the connection from 20-sim signals to CSP channels.

The developers of the involved disciplines need to know each other’s relevant design choices for a predictable end result. Some dependent information found in this example:

- Number of controllers versus the number of processes.
- Safety limits of the plant.
- Parameters: sampling frequency of the controllers and the scheduling.
- Interfaces (input/output).

Interfaces between views/viewpoints or tools can be checked by using contracts stored in the repository. A useful language to specify these relations is the AADL language [19]. In this case, there should be an extension or plug-in for both tools to read and/or modify the required dependent information.

Besides consistency checking at the boundaries, testing the complete embedded control system can be done by using co-simulation between 20-sim and gCSP. This requires time synchronization between the event driven CSP execution and the numerical integration for the controller simulation. In this case, the coupling between the two models of computation (and their simulators) is required. The interface layer in Fig. 5 should provide the tools to couple the two simulators by using the interface information stored in the repository. We have created a first prototype.
co-simulation facility that can use coupling information stored in an external repository. An important issue here is a common notion of time for the synchronization in simulation time between the continuous time and discrete event part. This is currently done by using fixed step integration methods in combination with periodic time events. Future work is the incorporation of hybrid modeling and simulation techniques like HyROOM or hybrid techniques used in Ptolemy.

The integration layer should provide support tooling to get the final software realization. Code generated from tools should be combined in an automated manner.

The last step is to couple the generated code to the chosen computing platform by connecting the input and output signals to the corresponding I/O hardware and drivers, which can be provided with the use of our CCE tool chain [6] in combination with the coupling information stored in the integration layer repository.

Summarizing, the multi-view integration layer consists of a set of support tools to facilitate the coupling between views, a data management layer for the storage of inter-view relations and shared information and support tools that facilitate combined code-generation. In this way we hope to address the problems described in the introduction. This research is

V. CONCLUSION

We have proposed a multi-view design methodology, which allows a better integration of the various tools used by the disciplines involved in the design of embedded mechatronic control systems.

Our methodology is currently tested in an industrial case study on the paper path of an Océ copier. Many models have been made in the past as part of the Boderc project [20] to model different aspects of the paper path. The main question here is “can we use these models in an integrated manner with automated consistency checking and in a predictable way for future product development?”. A second industrial case study on the development process of the control software for a power-assisted wheel chair will be executed end 2006.

The results of these case studies and the feedback of the industrial partners will help us to refine the ideas and methods to ensure later industrialization of the methods and prototype tools.

VI. ACKNOWLEDGEMENT

We thank our academic partners from the University of Eindhoven, Henk Corporaal, Jeroen Voeten and Jinfeng Huang for getting our ideas formulated sharper. We thank our industrial partners for their cooperation and giving us the opportunity to let our ideas be considered from an industrial point of view in such an early stage.

REFERENCES


