A Model-Driven Approach to Embedded Control System Implementation

Jan F. Broenink, Marcel A. Groothuis, Peter M. Visser, Bojan Orlic
University of Twente, Control Engineering, Faculty EE-Math-CS
P.O.Box 217, NL-7500AE, Enschede, the Netherlands
{J.F.Broenink, M.A.Groothuis, P.M.Visser, B.Orlic}@utwente.nl

Keywords: co-simulation, embedded control systems

Abstract

The work presented here is on setting up methodological support, including (prototype) tools, for the design of distributed hard real-time embedded control software for mechatronic products. The use of parallel hardware (CPUs, FPGAs) and parallel software is investigated, to exploit the inherent parallel nature of embedded systems and their control.

Two core models of computation are used to describe the behavior of the total mechatronic system (plant, control, software and I/O): discrete event system (DES) and continuous time system (CTS). These models of computation are coupled via co-simulation, to be able to do consistency checking at the boundaries. This allows for integration of discipline-specific parts on the model level (during design phases) instead of on the code level (during realization and test phases). Cross-view design-change influences get specific attention, to allow for relaxation of the tension between several dependability issues (like reliability and robustness), while keeping design time (and thus design costs) under control.

Furthermore, the design work can be done as a stepwise refinement process. This yields a shorter design time, and a better quality product.

The method is illustrated with a case using the tools being prototyped.

INTRODUCTION

Present-day design and realization of reliable and efficiently updateable software for embedded systems requires the availability of proper software tools, supporting integration of the various technical disciplines involved. We consider *Embedded Control Systems* (ECS) as a separate class of embedded systems, because the dynamic behavior of the plant / machine to be controlled is essential for the functionality of the embedded system (see Fig. 1). This implies that it is crucial for effective design work to have the behavior of the *overall* system available as a dynamic model in the design tool. This also corresponds with our *mechatronic* or *systems* approach for design. Furthermore, the I/O interface boards are treated separately, because of their specific role in the design trajectory.

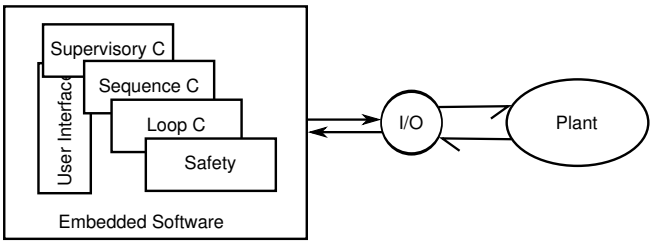

Figure 1. General architecture of embedded control systems

The software part consists of a layered structure of *controllers* (Bennett, 1988). The *Loop Controllers* implement the control laws and are *hard real-time*, because missing deadlines may result a catastrophic system failure (Kopetz, 1997). *Sequence Controllers* implement sequences of activities based on logical actions in time commanding the loop controllers. *Supervisory Controllers* control the sequence and loop controllers. For instance, mode-switching between sequence and loop controllers, optimization algorithms or expert systems adapting parameters of the lower-level controllers. The *Safety* layer deals with checking whether the signals to and from the plant are within their working envelope: outside that area, hazardous situations can occur.

The nature of these controllers imply that for implementing them, not only software engineering skills are needed but also insight in the real-time issues originated by the dynamic plant behavior and controller demands. These real-time demands dictate the responsiveness of the ECS, i.e. the allowable computational and network delays.

The design of embedded control systems for control of mechatronic systems has a multi-disciplinary development trajectory where many views, disciplines and tools are used. To shorten the design time and time-to-market, a multi-discplinary design traject is needed that allows concurrent engineering and cooperation between disciplines.

Several other research and commercial tools exist for embedded control system design. Each tool has its own specific properties and thus its specific target application area. Exam-

[1]This work has been carried out as part of the Boderc project under the responsibility of the Embedded Systems Institute and as part of the STW/PROGRESS ViewCorrect and Fieldbusses projects. This work is partially supported by the Dutch Ministry of Economic Affairs under the Senter TS program, and by PROGRESS, the embedded system research program of the Dutch organization for Scientific Research, NWO, and the Technology Foundation STW.

ples are Matlab, Simulink, Stateflow, Real-Time Workshop; Modelica, Dymola (Modelica, 2006); Ptolemy II (Lee, 2006). Surveys are given in (El-khoury et al., 2003; Törngren et al., 2006)

Current research deals with the development of a methodology and tools for the design of embedded control software, based on virtual prototyping (i.e. simulation) that allows the integration of discipline-specific parts on the model level (during design phases) instead of on the code level (during realization and test phases). This way, developers of different disciplines involved are helped to communicate across the boundaries of their discipline in a more formal way than social communication. 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 yields a shorter design time, and a better quality product.

Integration on the model level would be supported best by using one core model of computation. However, we believe that using *two* different models of computation, namely discrete event system (DES) and continuous time system (CTS), coupled via co-simulation, is a feasible and practical approach (Groothuis and Broenink, 2006).

This paper is organized as follows: First our design approach is presented globally. After that, the modeling formalisms we use are indicated. The third step of our design approach, Embedded Control System Implementation, is discussed in the fourth Section. A case study and conclusions complete this paper.

DESIGN APPROACH

The integrated model approach, using two co-simulatable models of computation, demands sophisticated tools, such that dependencies can be checked more or less automatically. Exploiting the simulate-ability of the models enables the design work to be done as a *stepwise refinement* process. This implies that the model will gradually change from a basic functional and conceptual model towards a detailed model from which the code for the embedded control computers can straightforwardly be generated. Our approach on designing embedded control software is the following rather common procedure (Broenink and Hilderink, 2001), see also Fig. 2:

1. *Physical System Modeling*, i.e. model the plant parts (dynamic behavior).
2. *Control Law Design*, using the models obtained in the previous step. Model reduction often is necessary to obtain a model of adequate level of detail.
3. *Embedded Control System Implementation*: the control software is designed via refinement of the control law(s).
4. *Realization* of the embedded software via an ongoing refinement process. Models of components are replaced by the real system parts for both the (I/O)-hardware, plant and software in a stepwise manner (Visser and Broenink, 2006).

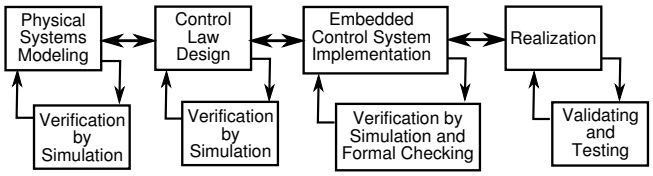

Figure 2. Embedded Control Systems Design Process

During each step, verification tests by means of simulation are carried out, also during the last phase (Realization) where some parts are still a model. During the ECS Implementation, formal verification is also used as a means of verification.

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. Note that this design approach focuses on the embedded control software. The design of the plant itself is a comparable approach, but will not be discussed here.

The focus in this paper is on step 3 and step 4. Step 1 and 2 are present for completeness, and only a summary is given when discussed (see Amerongen and Breedveld (2003) for more on step 1 and 2). Before elaborating on how this approach is supported by the software tools we are prototyping, first the modeling formalisms we use are presented.

MODELING FORMALISMS

Because we adhere a *mechatronic* or *systems* approach while designing the embedded control software, the dynamic properties of the *total* system, and not only the control software, play an important role. Thus, in order to verify the control software, also the dynamics of the plant must be taken into account. This implies that in a simulatable model, both the control software and the plant behavior need to be specified. Of course, the level of detail should fit the problem at hand. Furthermore, to really forecast the behavior of the system, relevant aspects of the computer hardware and interfaces must be taken into account. For example: computational and network delays, resolution of sensors, accuracy (resolution) of computation, the availability of a floating point unit and the CPU speed and memory footprint may influence the design to get the wanted behavior.

We use the following two modeling formalisms:

- *Communicating Sequential Processes* (CSP) for the embedded software parts. The inter-process communication is implemented by CSP-based channels (Hoare, 1985; Abdallah et al., 2004).

- *Bond Graphs* (directed graphs describing both the dynamic structure and dynamic behavior of th device) for the dynamic behavior of the plant (Karnopp et al., 2000; Breedveld, 1985).

CSP and bond graphs are both *port based*, and have their implementation independent of how they are connected. This really supports reusability. Furthermore, modeling and controller design activities (step 1 and 2) are separated from code generation and interface configuration activities (step 3 and 4). Models stay as long as possible hardware independent. This way, the models will not be polluted by I/O connections when one has to go back from step 3 / 4 to step 1 or 2.

Furthermore, the modeling formalisms are all directed graphs with facilities for hierarchy. Of course, on the lowest level, model equations / model code must be given to specify the algorithms / code blocks. Note that the designer can rather easily play with the border between graphs and equations / code, thus the resulting hierarchical models can be tuned towards a particular situation and preferences of the designer.

Combining these model descriptions into *one* diagram is naturally, because both formalisms are directed graphs, in which the activities occur in the vertices and the ideal (or idealized) connections are shown by the edges. In the following subsections, the formalisms will be described briefly.

Communicating Sequential Processes (CSP)

CSP (e.g. Hoare, 1985; Nissanke, 1997) is the process algebra we use to describe the embedded software as communicating processes, drawing it as directed graphs (a kind of Data Flow Diagram) (Hilderink, 2005; Jovanovic, 2006). The vertices denote the processes and the edges denote the communication of data (arrows in Fig. 3) as well as the composition structure, i.e. run in parallel, sequence and with different priorities (two vertical bars in Fig. 3 indicate: run in parallel). The filled bubbles in Fig. 3 denote the connections to the I/O via *link drivers*, device drivers having an interface like normal connections between processes.

CSP as process algebra allows for formal checks on these models. Futhermore, it allows easier distribution of parts of the embedded software to a different computing node in the system.

The edges in the graph (for the data communication) are so-called *channels*, which, besides exchange of data also govern the synchronization and thus scheduling between the processes (Hoare, 1985). The use of channels hides threads and priority indexing for the user, thus relieving the distributed software-writing problem significantly.

We have developed the CT (Communicating Threads) library delivering basic elements for creating building blocks to implement a communication framework using channels (Hilderink et al., 2000; Orlic and Broenink, 2004; Hilderink, 2005)

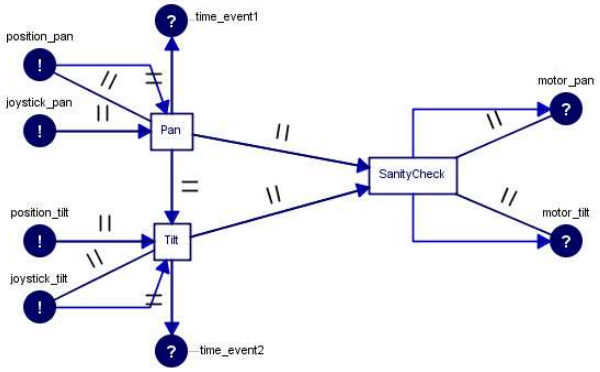

Figure 3. Example CSP graph

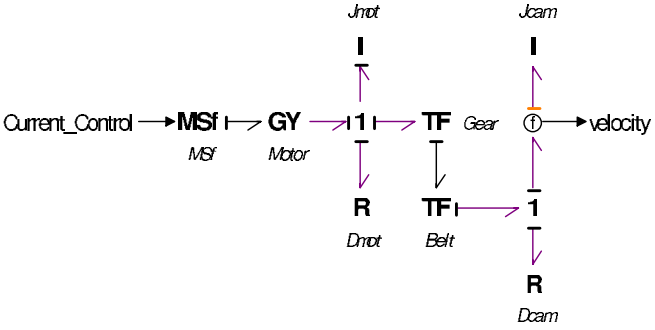

Figure 4. Example bond graph

Bond Graphs

For modeling the plant (machine)-part for the embedded system, we use *Bond Graphs* (Karnopp et al., 2000; Breedveld, 1985). Bond Graphs are directed graphs, showing the relevant dynamic behavior. Vertices are the submodels and the edges (called *bonds*) denote the ideal (or idealized) exchange of energy (Fig. 4. This is described by two variables, which product is the power exchange through the port. For each physical domain, such a pair can be specified, for example, voltage and current, force and velocity. The half arrow at the bonds shows the positive direction of the flow of energy, and the perpendicular stroke indicates the computational direction of the two variables involved.

Entry points of the submodels to connect the energy flows (bonds) to, are the so-called *ports*, and are the connection points for the two variables of the bond. The submodel equations are specified as real equalities, and not as assignments. These two properties are essential and ensure true encapsulation and thus reusability.

Bond graphs are physical-domain independent, due to analogies between these domains on the level of physics (i.e. essential building blocks). Thus, mechanical, electrical, hydraulical etc system parts are all modeled with the same graphs.

Bond graphs may be mixed with block diagrams in a nat-

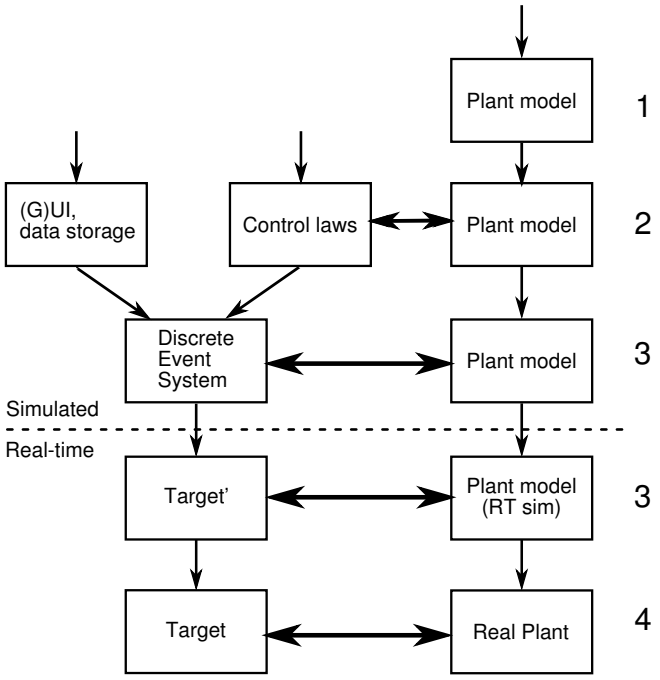

Figure 5. Work flow, with emphasis on step 3

ural way to cover the information-domain part. For example, to model a controlled mechatronic system, the control law(s) are specified with block diagrams, and the plant is specified with bond graphs (see Figs. 1 and 9).

Differential equations are generated after model compilation, whereby the port variables obtain a computational direction (one as input, the other as output) and the equations are rewritten as assignment statements. This process is rather efficient, because *computational causal analysis* on graph level is used. Thus the structure of the graph is exploited, since the computational direction depends on how the submodels are interconnected.

EMBEDDED CONTROL SYSTEM IMPLEMENTATION

The process of the model-based approach to embedded control system implementation is basically transforming the combined graphs denoting the dynamic behavior of the ECS to respectively code (embedded software), which can run on the selected target control computer and simulation models (plant). This transformation process is a stepwise refinement process, allowing for taking small and testable steps. Via simulations, the refinements can be verified, i.e. checked whether the refined models still comply with the requirements. The four steps as presented in Fig. 2 are illustrated in Figure 5.

The properties of the target platform heavily influence what functionality of the code to be generated will be covered. Hence, we explicitly discuss the target and its facilities for real-time embedded control, i.e. the CTC++ library. After this, the ECS implementation and Realization steps are treated. Our prototype tools are presented thereafter.

Controller models

As stated before, the controller models (CSP descriptions) are transformed to code on the target, while the plant models (bond graphs) are used in simulations.

The controller models originate partly from the control laws, as designed in step 2 of the design approach (middle starting point in Fig. 5), while most of the code deals with exception handling, user interfacing, communication, data filtering etc (left starting point in Fig. 5). The control law parts fill in the control code blocks specified in the embedded software dataflow diagrams. The interface of the control code block must be the same as the interface of the control law blocks resulting from step 2. This ensures that *no* manual adaptation is needed, which would otherwise be a serious source of errors.

The resulting combined DES - CTS system can be co-simulated to check whether the add-ons of the software parts let the total system still behave according to the requirements.

For treating exceptions, it is relevant to separate the normal flow of control from the exception handling. The CSP approach is useful here, where the exception-handling process is coupled to the normal-flow process via the CSP exception operator (Jovanovic et al., 2005).

This building block approach to the embedded control software elegantly allows for applying design patterns concerning dependability of the software (Jovanovic, 2006, Chapter 6). Examples are system load watchdogs, N-version programming and logging & monitoring. This last pattern can be used as a basis for remote diagnostics and remote maintenance.

Target platforms

In order to have a generic target with the proper functionality, we have implemented a target abstraction layer: the CT (*Communicating Threads*) library (Hilderink et al., 2000; Orlic and Broenink, 2004; Hilderink, 2005). Equal to the CSP and occam languages, our CT library is based on a message passing process architecture, where concurrent processes communicate exclusively via rendezvous channels. The targets platforms are in principle distributed computer systems, interconnected via fieldbusses. Using the CT library, this distributedness is abstracted away (both a connection internally or over a network are specified as channels), although a fieldbus interconnection mostly has a significant influence on the control system performance. Issues like (varying) communication delays and unreliable communication links become significant. Hence, besides control theory and techniques, communication theory and techniques are necessary to reason on the behavior of the total (control) system.

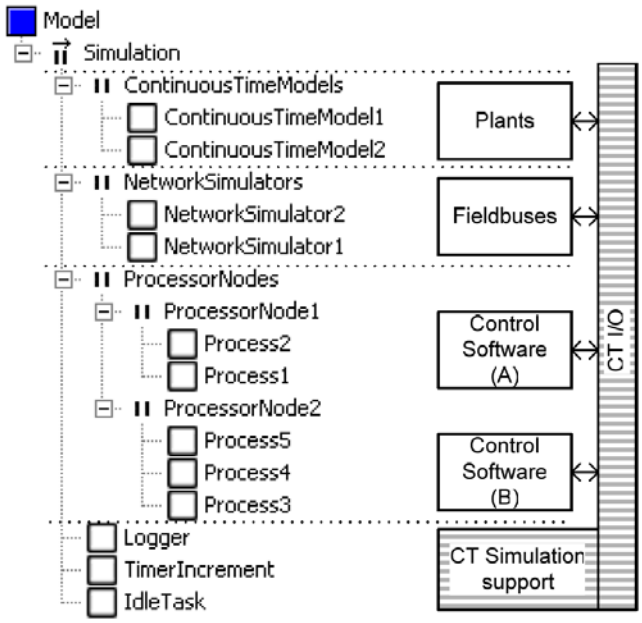

Figure 6. Network Simulator structure

The nature of this distributed embedded computer system is captured in our network simulator, with which a complete distributed control system can be simulated (ten Berge et al., 2006). Its communication framework is designed such that the transition from simulated system to real control system is straightforward, see Fig. 6.

Stepwise refinement

The refinement process now is in step 3. Plant models have been made and verified (and maybe validated if a prototype is available). The control laws have been designed, and verified by simulation using the plant model of step 1. Computer equipment and instrumentation are assumed ideal in these first two phases.

Note that classically there is a breakpoint in the way of working. The first two phases are done by control engineers / engineering specialists, while for this third phase, mostly (embedded) software specialists are engaged. Parallel to this, also the tools used are different. Consequently, the results out of phase 1 and 2 cannot smoothly be used in the next phases. This is a major source of errors. It is also a major motivation for our work, and directs how tools can be connected together.

The stepwise refinement procedure for the *embedded sofware* consists of the following parts:

1. *Integrate Control Laws.* Combine the control law(s) with the safety, sequence and supervisory control layers. Reaction to external events (operator commands or events from connected systems) are taken into account. The implementation is still assumed to be ideal
2. *Capture non-ideal components.* Those components considered ideal previously, are now modeled more precisely by adding relevant dynamic effects. Also, signal processing algorithms are added to obtain signals which cannot be measured directly in the practical situation.
3. *Incorporate safety, error and maintenance facilities.* These additions are added and the effect on the behavior of the total system can be checked by simulation.
4. *Effects of non-idealness of computer hardware.* The control computer hardware and software architecture is added. Effects on accuracy and latency can be checked.

These parts need not to be performed strictly in this order, giving the designer the freedom to tailor this step to the specific problem at hand. For example, safety is crucial when dealing with a fast-moving robot arm, while resolution and latency may be essential when dealing with a low-cost networked embedded control system.

Tools

By stimulating an iterative approach, which is a quite natural way of working, tool support becomes inevitable. This motivates our research on the design framework and tool development. Note that iterative ways of development are also used in areas of software engineering (e.g. Douglass, 2004), systems engineering (e.g. Blanchard and Fabrycky, 2006) and control law design (e.g. Amerongen and Breedveld, 2003).

The tools we use here, are prototyped at our Laboratory:

- *20-sim*, for modeling, analysis and simulation of continuous time models, using bond graphs, idealized physical models, block diagrams or differential and algebraic equations (DAE) as modeling formalisms (combined in one model). Also control laws can be specified in 20-sim, as a combination of difference equations and block diagrams (CLP, 2006). C code can be generated from these models or from specific model parts. The generated code is used further in the tool chain. Code generation templates can be developed to support specific target boards. Special tokens in these templates represent relevant model attributes (variables, parameters, etc).
- *gCSP*, for modeling software structures and code from a CSP perspective. At the lowest level in the graph, code blocks specify the actual algorithms, for which either code can be specified, or imported from 20-sim. A prototype is available (Jovanovic, 2006).
- *CTC++*, a real-time middleware layer, which implements the CSP approach of concurrency. It runs on DOS, Windows (of course not real-time), Linux, real-time Linux (RTAI, Xenomai) and on a Analog Devices DSP board (Hilderink et al., 2000; Orlic and Broenink, 2004; Hilderink, 2005).
- *NWsim*, a distributed simulator based on CTC++ and TrueTime, capable of simulating DES and CTS parts together. The network simulator is constructed as a set of

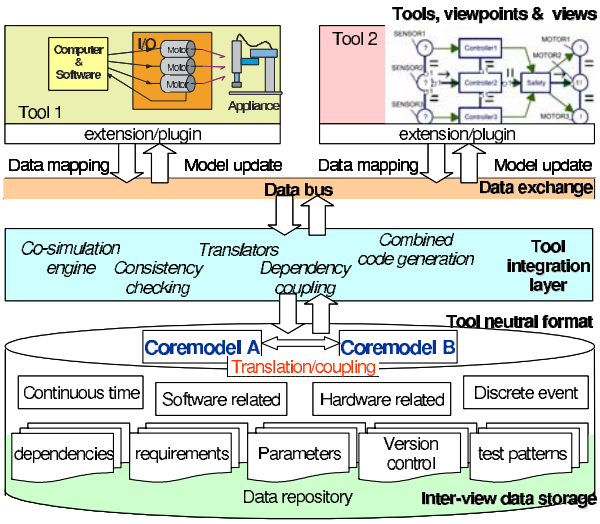

Figure 7. Multiple view integration framework

rendezvous communicating components. Synchronization of the simulated time is done by letting each component connect to a central timer object via a dedicated, specific rendezvous (blocking) timer channel (see Fig. 6), together with priority assigned to the constituting parts (ten Berge et al., 2006). Fieldbus communication delay effects on the controller performance can be tested.

- *ForSee*, a tool chain to support the user in the path from generated code to deployment and testing on the embedded target. The process of connecting model signals to I/O pins on the target hardware is done here to keep our models largely independent of the embedded target. This connection process is independent of the used modeling tool, and as such prevents vendor lock-in. The toolchain facilitates the (cross) compilation and deployment (transfer to the target and run) activities. It supports rapid prototyping via real-time logging and on-target parameter adjustments (ECSSoftware, 2006).

These tools are on different levels of maturity. 20-sim is commercially available. gCSP, CTC++ and parts of the ForSee tool chain are used at our Laboratory during research projects and MSc courses and are available (ECSSoftware, 2006). NWsim is an experimental prototype.

The tools cover together the total workflow (see 5), although the real support for cooperating development engineers is far from complete. Extending the approach indicated in Fig. 5 and in the subsection on Controller models, the tools have to be tightly coupled, but in such a way that adaptation of the framework and exchange of tools is easily doable. A basic layout of such a tool framework is given in Fig. 7. The tools presented here will become part of this framework.

In order to give an idea about what functionality is needed, consider the following two examples:

1. If a mechanical engineer changes the ratio of the gears in a gearbox, the control engineer (and/or the software developer) should be informed about that change, because this ratio has influence on their work. If this dependency relation is known, it should be possible to let the tools check this relation for the developers. In this way, possible problems and conflicts later on during the integration phase can be prevented.
2. The framework supports model refinement. Models are often developed in various steps, starting from simple rough models to more detailed and accurate models. The current approach is often that people create a new model based on the old one and extend/improve this model. The result is that multiple models will live next to each other with a lot of common parts and parameters. The usage of multiple (part/sub) models for one design with overlap requires synchronization between common parts (mostly parameters, constants) just like version control and merge/diff tools for software development.

When co-simulation techniques are used to simulate two or more (sub)models together (e.g. electical and mechanical), information about inter-connections between signals in these models is needed. Furthermore, information about how the simulation will progress in time is needed (fixed time steps or variable step). For code-generation connections from model signals to device drivers are needed. This information is about the boundary and interconnection between views and tools. This coupling information can be stored in a tool-independent shared repository. This is the generalization of the storage of connector information currently implemented by the ForSee tool.

CASE STUDY

As case the twin axes device JIWY has been used (see Fig. 8 for a photo). It has 2 rotational joints, and endstops are mounted limiting the operational area to prevent cable damage while turning around. Each joint is equipped by a DC motor and an incremental encoder (angle measurement). The motor drives the load via a gear box and a toothed belt. The set up is controlled via an USB joystick. The I/O interfaces consist of quadrature encoder inputs and PWM signal generators, which drive the H-bridge power amplifiers, both implemented in an FPGA. Both axes are independent of each other. The signal coupling between the controllers is only for information purposes (see Fig. 9). The plant model is given in Fig. 4. The software structure is shown in Fig. 3.

The prototype tool chain did function rather smoothly on the JIWY in our academic lab environment. We saw a shorting of the design time, but this observation is rather subjective.

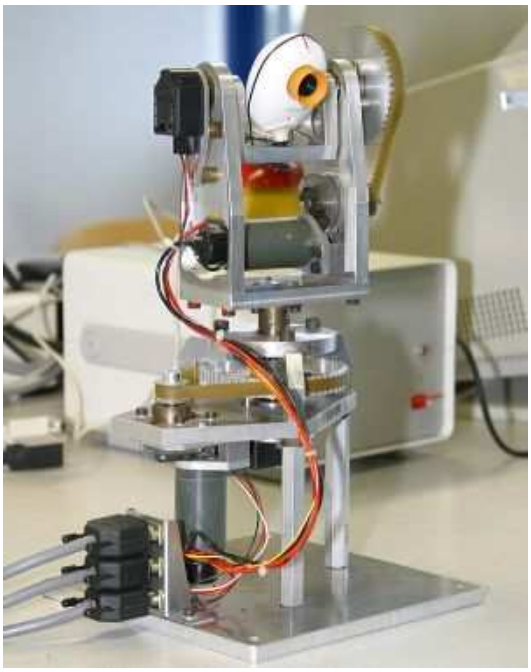

Figure 8. JIWY test case

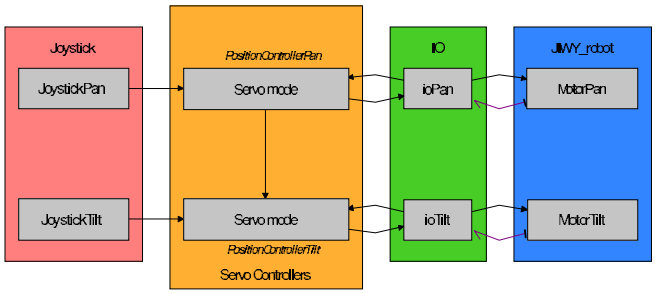

Figure 9. Position controllers of JIWY

After these initial tests, the workflow has been tried out at our Laboratory in the following situations:

- 2nd year Electrical Engineering students, doing their Lab work on the integration of courses on Modeling and Simulation, Measurements, Transducers and Control Engineering. Students have to design and construct a mechatronic demonstrator using some given transducers (e.g. loudspeaker, voice coil, miniature electromotor), controlled via a DSP-based embedded control computing board. The embedded software part is completely hidden, because 'only' a control law needs to be implemented, thus only the right and middle entry of Fig. 5 are necessary (Jovanovic et al., 2003).
  Indeed, the students are supported by the tools and can work effectively concerning the embedded software.
- Master students Electrical Engineering, for their lab work of a course on Real-Time Software Development. They had to develop embedded control software for the JIWY set up. The series of exercises led the students through the workflow, such that the final test on the hardware could be done as "doing first time right", while all preparations were done at home using the tools. A.o. they had to design and test by simulation a link driver of the PWM output.
  It turned out that all groups managed to control the physical device during one time slot (2 or 3 hours) on our lab.
- MSc final project students test the tool chain by using it for their robotic projects. Depending on the nature of their projects, different aspects of our tools were stress tested. For example, a 6 DOF production cell like test setup with rather extensive sequence and supervisory controllers pushed our prototypes to its limits.
  The expected shortening of time to market was not reached in this case, because of the inmatureness of the tools. At other projects, the tool chain did function rather smoothly.

CONCLUSIONS

The prototype tool chain does function rather smoothly in an academic lab environment. We saw an sophisticated use of the tools, such that design time was used effectively. Really a shortening of the design time was not yet significant enough, but promising.

Future work is to continue working on the development of the tools. Also larger cases are worked on in cooperation with industry, to test whether our claim (shorter design time) can be met.

REFERENCES

Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. *Communicating Sequential Processes, the first 25 years*. Springer, London, 2004.

Job van Amerongen and Peter C. Breedveld. Modelling of physical systems for the design and control of mechatronic systems (ifac professional brief). *Annual Reviews in Control*, 27:87–117, 2003.

S. Bennett. *Real-Time Computer Control: An Introduction*. Prentice-Hall, London, UK, 1988.

B.S. Blanchard and W.J. Fabrycky. *Systems Engineering and Analyis*. Prentice Hall, 4 edition, 2006.

P.C. Breedveld. Multibond-graph elements in physical systems theory. *Journal of the Franklin Institute*, 319 (1/2):1–36, 1985.

Jan F. Broenink and Gerald H. Hilderink. A structured approach to embedded control systems implementation. In M.W. Spong, D. Repperger, and J.M.I. Zannatha,

editors, *2001 IEEE International Conference on Control Applications*, pages 761–766. IEEE, Mexico City, Mexico, 2001.

CLP. 20-sim, 2006. http://www.20sim.com.

Bruce Powel Douglass. *Real Time UML: Advances in the UML for Real-Time Systems*. Addison-Wesley, Boston, 3 edition, 2004.

ECSSoftware. Ce laborary's website on ecs software, 2006. http://www.ce.utwente.nl/ECSSoftware.

Jad El-khoury, DeJiu Chen, and Martin Törngren. A survey of modelling approaches for embedded computer control systems (version 2.0). Technical Report KTH/MMK/R-03/11-SE, Mechatronics Lab, Department of Machine Design, Royal Institute of Technology, KTH, 2003.

Marcel A. Groothuis and Jan F. Broenink. Multi-view methodology for the design of embedded mechatronic control systems. In *Proc. IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*, pages 416–421. IEEE, Munich, 2006.

Gerald H. Hilderink. *Managing Complexity of Control Software through Concurrency*. Phd thesis, University of Twente, Netherlands, 2005.

G.H. Hilderink, A.W.P. Bakkers, and J.F. Broenink. A distributed real-time java system based on csp. In *The third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing ISORC 2000*, pages 400–407. IEEE, Newport Beach, CA, 2000.

C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, 1985.

D.S. Jovanovic. *Designing dependable process-oriented software, a CSP approach*. Phd thesis, University of Twente, Enschede, NL, 2006.

D.S. Jovanovic, B. Orlic, and J.F. Broenink. On issues of constructing an exception handling mechanism for csp-based process-oriented concurrent software. In Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors, *Communicating Process Architectures CPA 2005*, pages 29–41. IOS Press, Eindhoven, NL, 2005.

Dusko Jovanovic, Bojan Orlic, Jan F. Broenink, and Job van Amerongen. Inexpensive prototyping for mechatronic systems. In *WESIC 2003*, volume II, pages 431–438. IEE, Miskolc, Hungary, 2003.

Dean C. Karnopp, Donald L. Margolis, and Ronald C. Rosenberg. *System Dynamics: Modeling and Simulation of Mechatronic Systems*. Wiley-Interscience, 3rd edition edition, 2000.

Hermann Kopetz. *Real-Time Systems, Design principles for Distributed Embedded Applications*. The Kluwer international series in engineering and computer science, ISSN 0893-3405; 395. Real-time systems. Kluwer Academic Publishers, Boston, 1997.

Edward A. Lee. The future of embedded software. In *Artemis Annual Conference*, page 44 Slides, Graz, Austria, 2006. Ptolemy Project.

Modelica. Modelica website, 2006. http://www.modelica.org.

Nimal Nissanke. *Realtime Systems*. Prentice Hall Series in Computer Science. Prentice Hall, London, 1997.

Bojan Orlic and Jan F. Broenink. Redesign of the c++ communicating threads library for embedded control systems. In Frank Karelse, editor, *5th PROGRESS Symposium on Embedded Systems*, pages 141–156. STW, Nieuwegein, NL, 2004.

Matthijs H. ten Berge, Bojan Orlic, and Jan F. Broenink. Co-simulation of networked embedded control systems, a csp-like process-oriented approach. In *Proc. IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*, pages 434–439. IEEE, Munich, 2006.

Martin Törngren, Dan Hendriksson, Karl-Erik Arzen, Anton Cervin, and Zdenek Hanzalek. Tool supporting the co-design of control systems and their real-time implementation: current status and future directions. In *Pro. 2006 IEEE Conference on Computer Aided Control System Design (CACSD)*, pages 1173–1178. IEEE, Munich, 2006.

Peter M. Visser and Jan F. Broenink. Controller and plant system design trajectory. In *Proc. IEEE Int'l Symposium on Computer Aided Control Systems Conference, CACSD 2006*, pages 1910–1915. IEEE, Munich, 2006.