Electronic Communications of the EASST
Volume 21 (2009)

Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009)

Concurrent Design of Embedded Control Software

Marcel Groothuis, Raymond Frijns, Jeroen Voeten and Jan Broenink

10 pages

Guest Editors: T. Levendovszky, L. Lengyel, G. Karsai, C. Hardebolle
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Concurrent Design of Embedded Control Software

Marcel Groothuis[1], Raymond Frijns[2], Jeroen Voeten[2] and Jan Broenink[1]

[1]University of Twente, Control Engineering, Faculty of EE-Math-CS, P.O. Box 217, 7500 AE Enschede, The Netherlands, {m.a.groothuis, j.f.broenink}@utwente.nl,
[2]Faculty of Electrical Engineering, Eindhoven University of Technology, Den Dolech 2, 5600 MB Eindhoven, The Netherlands, {r.m.w.frijns, j.p.m.voeten}@tue.nl

Abstract: Embedded software design for mechatronic systems is becoming an increasingly time-consuming and error-prone task. In order to cope with the heterogeneity and complexity, a systematic model-driven design approach is needed, where several parts of the system can be designed concurrently. There is however a trade-off between concurrency efficiency and integration efficiency. In this paper, we present a case study on the development of the embedded control software for a real-world mechatronic system in order to evaluate how we can integrate concurrent and largely independent designed embedded system software parts in an efficient way. The case study was executed using our embedded control system design methodology which employs a concurrent systematic model-based design approach that ensures a concurrent design process, while it still allows a fast integration phase by using automatic code synthesis. The result was a predictable concurrently designed embedded software realization with a short integration time.

Keywords: Embedded Systems, Case Study, Design Methodology, Mechatronics

1 Introduction

The design process of modern mechatronic systems is becoming more and more cumbersome due to the increasing complexity and heterogeneity of such systems. This heterogeneous and multi-disciplinary nature (involving mechanical, electrical, control and software engineering) makes the design and especially the integration of all parts (both physical and software) a time-consuming and error-prone task. Traditionally, a sequential design approach is used where each discipline uses its own design flow, domain-specific terminology, models and tools to (independently) design parts of the system. The main reason for this typical design flow is the lack of a well-integrated method, supported by a heterogeneous toolchain to employ an efficient multi-disciplinary design methodology. Figure 1 shows the impact of the design process on the overall design time. To speed up the traditional sequential design process (**a**), concurrent engineering techniques should be applied (**b**). The usage of a model-driven design trajectory (**c**) can accelerate the design process even further because models allow a more thorough and a faster design space exploration.

There is however a trade-off between a largely concurrent design flow and the integration efficiency (Figure 1**d**). On one hand, since the concurrent tracks start at the same time, they lack details of other (non-finished) parts of the system, and therefore have to rely heavier on assumptions. This can result in extra integration problems and inconsistencies, since less (or no) information is exchanged between the concurrent design trajectories. On the other hand,

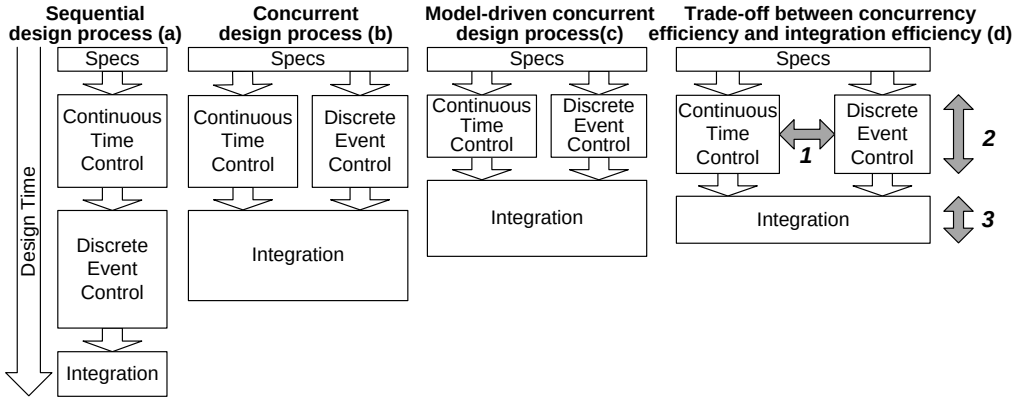

Figure 1: Impact of different multi-disciplinary design flows on the overall design time

an extensive (early) integration at the model level (**1**) is also not feasible due to the significant additional time (**2**) spend for these integrations in order to shorten the (late) integration time (**3**). Furthermore, the heterogeneous nature of mechatronics involves the usage of multiple modeling paradigms, models of computation (MoC) and tools which complicates this early model-level integration.

While it is not impossible to couple (co-simulate [GDB08]) or integrate the heterogeneous models (using Ptolemy II [EJL+03] for example), industry often wants to use existing and the best (available within the company) tool/method/MoC for each part of the system and keeping a 'separation of concerns', while still requiring an integrated and predictable result at the end. It is desired to have the best of both worlds: efficient concurrent design combined with seamless (short) and predictable integration. This requires a systematic and flexible (preferably tool independent) methodology for multi-paradigm model-driven design of (the ECS software for) mechatronic systems.

This paper presents a case study on the concurrent design of the Embedded Control System software (ECS) for a real-world mechatronic setup using our multi-disciplinary and model-driven ECS design methodology.

The paper is organized as follows: Section 2 gives more information about ECS software. Section 3 explains the used ECS design methodology. Section 4 presents the case study and the usage of our methodology, followed by an evaluation on concurrent design and integration efficiency in Section 5 and conclusions and ongoing work in Section 6.

2 Embedded Control System Software

A typical mechatronic system consists of a combination of a mechanical (physical) system, mixed-signal and power electronics, and an embedded (motion) control system (ECS). The combination of a mechanical setup and its ECS software requires a multi-disciplinary and synergistic approach for its design, because the dynamic behavior of the mechanics influences the behavior of the software and vice-versa (also known as *cyber physical systems*). Therefore the physical system and its software should be designed together (co-design approach) to find an optimal and dependable realization. The purpose of an embedded control system is to control physical processes like mechatronic setups and more specific the coordinated mechanical movements (like position, velocity and acceleration) to get a smooth and precise movement. A typical ECS software design contains the layered structure [Ben94] shown in Figure 2. The ECS software is a

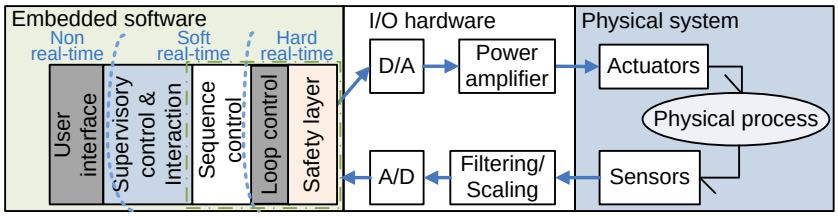

Figure 2: Embedded Control System and its software structure

combination of an event-driven part (sequence of motions, supervisory coordination and safety) and a time-triggered part with motion profiles (the trajectory to follow) and loop controllers (control law). Embedded software for feedback control has challenging hard real-time and low latency requirements which makes it a special subclass of embedded systems software. Hard real-time behavior is required for the low-level (near the hardware) layers. The control laws for the loop control layer require a periodic time schedule with hard deadlines in which jitter and high latency are undesired.

The usage of a model-driven design flow for mechatronic systems and their software implies the usage of multiple models of computation (MoC), modeling formalisms and heterogeneous modeling. *Continuous time, ODE* (ordinary differential equations) are the typical MoC for dynamic system behavior and loop controllers. For the high-level layers of the ECS software various types of *discrete event* MoCs are in use.

3 Methodology

This section presents an overview of the design strategy. Section 4 presents the details with respect to the case study. Our methodology aims at a *concurrent design* process with minimal but sufficient information-exchange between the concurrent tracks (i.e. maximizing concurrency efficiency), while still relying on a predictable and short integration phase by using a systematic *model-driven design* approach supported by a toolchain that supports automatic *code synthesis*.

3.1 Model-driven Design

One key point in our methodology is to use a model-driven design approach. Model-driven design is essential for design automation and the extensive use of (executable) models can give valuable early system feedback. In this way, design errors can be detected long before an (prototype) implementation is realized, so design iterations will be much shorter and less costly. Also, model-driven design allows fast and thorough design space exploration, resulting in well-founded design choices in the earliest stages of design.

3.2 Concurrent Design

First, the system is partitioned into a set of concurrent parts (called actors), based on the physical layout of the system. The top-level interactions of these actors are specified in an abstract model. The actors themselves can be further partitioned into a continuous-time and a discrete-event part, since these parts have completely different models of computation. After jointly specifying the global interfaces and communication strategy between the different parts, they can be designed fully concurrently.

Based on the abstract system model and the partitioning into discrete-event and continuous-time parts, a stepwise refinement process is started for each of these parts, where each track

works towards the joint interfaces from a different direction (top-down vs. bottom-up). The refinement process is done in a modular, property-preserving way by adding more detail to the models.

3.2.1 Discrete Event

The discrete event control software (consisting of the supervisory control & interaction and sequence control layers in Figure 2) is designed with a top-down approach. With each refinement step, the externally observable behavior should stay the same, in order to get a local and predictable stepwise refinement process, which will ensure a seamless coupling of the parts in the integration phase. The refinement consists of the following three steps [HVG+07]:

1. *Abstract modeling:* After partitioning the system into several autonomous concurrent subsystems called actors, an abstract model representing the high-level (supervisory) *interactions* between these actors is made.
2. *Model refinement:* The initial abstract model is refined locally, i.e. the actors are internally repartitioned into a high-level part and a low-level discrete-event part, while the externally observable behavior is unchanged. The refined model adds the *internal interactions* between the internal high-level control and low-level sequence control of the actors.
3. *Synthesis:* In the last step, (real-)time behavior and the low-level interaction with continuous-time behavior are added to the actor behaviors. After this step, the model is ready for automatic property-preserving [HVC07] code synthesis and integration with other software layers (like loop-control or a Human-Machine Interface).

3.2.2 Continuous Time

The low-level ECS software layers (loop control and safety in Figure 2) are designed bottom-up from the mechanical system behavior towards the top-level discrete event software framework interfaces using the following stepwise refinement procedure ([BGVO07]):

1. *Physical Systems modeling:* Create a mechanical system model with relevant setup behavior; verify behavior by simulation; use the model to derive the required control laws.
2. *Control Law Design:* Design the required control algorithms and motion profiles (movement trajectories) using control theory and the model from the previous step; verify stable and correct controller behavior via simulations.
3. *Embedded Control System Implementation:* The loop controllers from the previous step are still based on continuous time assumptions and ideal sensor and actuator behavior. Include the relevant target behavior (electronics, execution platform) like discretization (AD/DA conversion), a transformation from continuous time control to digital control in the model and verify the behavior by simulation.
4. *Target Realization:* Prepare the loop controllers and motion profiles for inclusion in the ECS software by adding the event control interfaces. The loop controllers and motion profiles are synthesized from the model via (partial) code synthesis. The correct working is validated on the real setup on a per unit base (when possible).

3.3 Fast Integration using Code Synthesis

When the discrete-event and the continuous-time models both contain just enough detail to specify all properties of interest (competent model), they can be merged into an executable implementation by synthesizing building blocks sharing a common interface (in for instance C-code).

In this way, integration problems caused by design inconsistencies between the concurrent design tracks can be resolved quickly, since the software integration itself can be fully automated (i.e. short design iterations). It is required that the synthesis tools are sufficiently complete in order to generate code for any model element of interest. Furthermore, the code generation must preserve model behavior (e.g. preserving timing properties).

4 Case-study

With this case study, we demonstrate the design of the embedded control software for a real-world mechatronic system using our methodology and its supporting toolchain, and analyze the efficiency of the integration phase after applying our methodology of fully concurrent design with minimal information exchange. In the experiment, the model-driven and concurrent design process was executed at two physically separated locations with one common design session at the beginning for a top-level abstract model and one common session at the end for the integration and testing process.

4.1 System Description

The Production Cell setup [vdB06] that we have chosen for this experiment is a scaled down lab version of an industrial plastics molding machine, a typical example of a real-world mechatronic production line system. Our setup (see Figure 3) consists of 6 *Production Cell Units* (abbreviated: *PCU*). All PCUs operate simultaneously and need to synchronize with its neighbor PCUs to pass along metal blocks. Each PCU executes a single (pseudo)action in the production process like *feeding* (of raw material), *molding*, *extraction* from the machine, *transportation* (belts) and *storage*. The storage part is simulated by a rotation PCU that picks up a block at the end of the production process and transfers it again to the beginning of the setup, resulting in a loop. Sensors (located before and after the PCUs) are used to detect blocks and are external events for the PCUs to do their job. The loop in combination with the sensor-events-triggered PCUs can result in a deadlock when the setup contains 7 or more blocks. The setup needs at least one free buffer position (next to the sensors) in order to be able to move blocks to the next PCU. When all sensor guarded buffer positions are occupied, the system cannot move anymore resulting in a deadlock. The blocks are picked up using electromagnets mounted on the extraction unit and the rotation unit, transported by the belts and pushed forward by the feeder. The mechanical setup is connected via power and interface electronics to an embedded PC with an FPGA I/O card, that runs the embedded control software under real-time (RTAI) Linux.

4.2 Used Modeling Tools and Languages

The 20-sim tool [Con09] is used for the dynamic systems modeling and feedback controller design. 20-sim supports multi-disciplinary modeling with library components for many engineering disciplines and also supports the domain independent bond-graph notation, which we used for this case, because its energy-based modeling paradigm support our goals better than the conventional block diagram approach used by, for example, Simulink. 20-sim supports model checking and has an extensive control toolbox. It has a customizable template-based C-code synthesis facility for automatic code synthesis of whole models or submodels (e.g. only the controller) with a strict separation between model dependent and target dependent code. The code

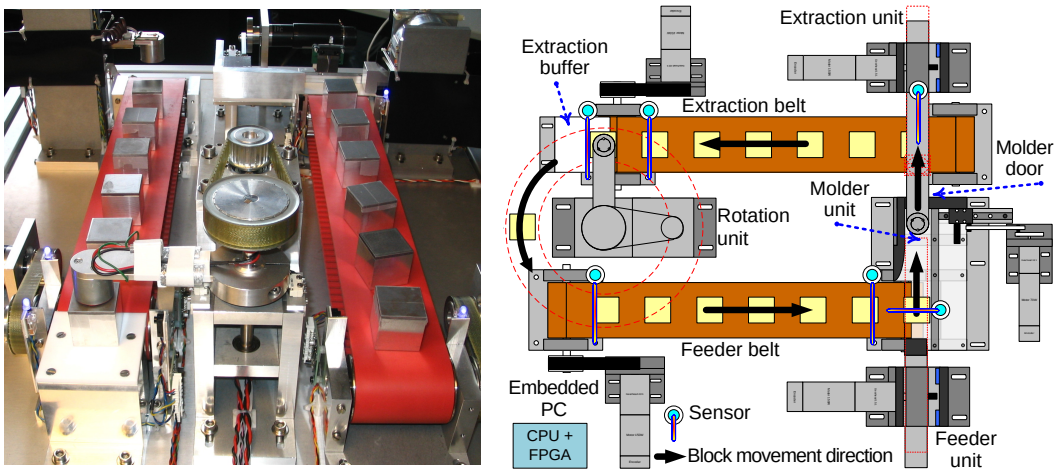

Figure 3: The Production Cell setup

synthesis facility is used for this case to synthesize the continuous time ECS software part.

Discrete event control software is designed and modeled in POOSL (Parallel Object Oriented Specification Language) [PJ97], a specification language with a formal semantics (based on real-time CCS), suitable for modeling combined software and hardware systems. The language consists of three conceptual layers; a data, a process and an architecture layer. The data layer consists of an object-oriented programming model for execution of complex calculations. The process layer models process behavior and inter-process communication (by synchronous message passing). The architecture layer clusters processes together to form a hierarchy. The graphical tool SHESim [GVP+00] is used for simulation and verification of the POOSL model.

For real-time code synthesis from these models, Rotalumis [vB02] is used. This tool contains several optimizations for predictable real-time execution of POOSL-models and can interface with external hardware and/or software through the use of primitive data classes. These primitive data classes link external C++ code to the data layer of POOSL and are used to interface with the Production Cell sensors and actuators and the continuous time ECS software part.

4.3 Discrete Event Control Software Design

For the design of the discrete event control of the production cell, a top-down approach is used, where initial abstract models are extended with more details in a systematic stepwise way. This refinement process is done in a modular fashion, while preserving the externally observable behavior of the higher levels of abstraction. In this way, the impact of local changes on the system as a whole is much clearer. The following sections give a detailed overview of these systematic model refinements as used for the production cell case.

4.3.1 Abstract Model

First, the system is partitioned into a set of independent concurrently working units called actors, in this case the PCUs of figure 3. Even though the PCUs themselves operate autonomously, their mutual interactions need to be synchronized and coordinated by handshaking protocols. The handshake protocols are expressed in the handshake diagram [HVG+07] shown in the left of

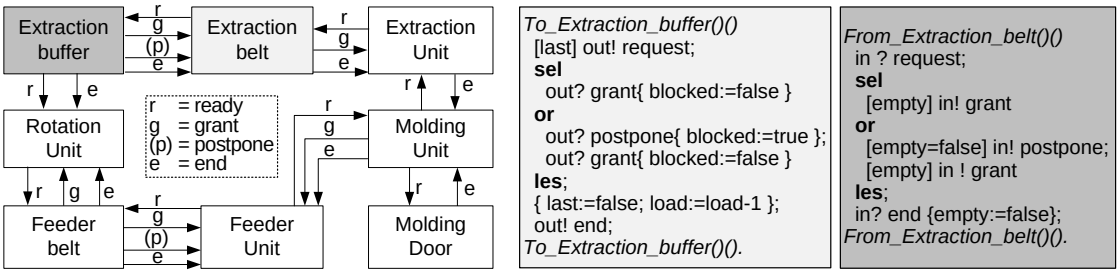

Figure 4: Handshake diagram of the production cell (left), and POOSL-code of the handshake protocol between the extraction belt and extraction buffer actors (right).

Figure 4. The interactions themselves are not specified yet, only an abstract untimed negotiation sequence between actors is modeled. The handshake model is then formalized in an executable POOSL model, specifying the synchronization protocol between the extraction belt and the extraction buffer, shown at the right side of Figure 4.

When the extraction belt has a block ready at its end (indicated by Boolean variable *last*), it needs to synchronize with the extraction buffer before transferring it, and acts depending on the state of this buffer. When the buffer is full, the belt should stop moving until the buffer is ready to accept another block. When the buffer is empty, the belt can continue moving the block onto the buffer. Therefore, the synchronization starts with the belt sending a *request* message (out ! request) to the buffer. When the buffer is available (indicated by Boolean variable *empty*), it immediately sends back a *grant* message ([empty] in ! grant), otherwise it will first send a *postpone* message. When the belt receives this *postpone* message, it is blocked by setting the Boolean variable *blocked* to true. When the buffer is empty again, it sends a *grant* message to the belt, which can then continue moving the block by setting *blocked* to false. After receiving the block, the buffer sends an *end* message to the belt, finishing the handshake.

Even though this model is abstract, and only describes the high-level synchronization interactions between some abstract actors, it provides already useful feedback to the designers, like the possibility of deadlocks in the system with 7 blocks or more.

4.3.2 Model Refinement

The first refinement step adds the untimed *interactions* between high-level discrete event control and the low-level continuous-time behavior of the actors. Effectively, this means adding the sequence control layer of Figure 2, still without specifying the actual low-level behavior itself. The left side of Figure 5a shows the refined model of the extraction belt. The belt actor is internally expanded into a high-level control part (*High_ctl*), and several low-level parts (*SensorF_low_ctl*, *SensorL_low_ctl* and *motor_low_ctl*). The outer interface (ports *in* and *out*) is exactly the same as in the abstract model, but the addition of the (yet undefined) low-level parts allows the specification of internal interactions between the high-level and low-level parts (sequence control). These additional interactions do not change the observable behavior; in fact the belt actor and its refinement are observationally equivalent. The right side of Figure 5a shows the POOSL-code of the refined model. The leftmost block contains the POOSL-code of the *High_ctrl* part, which are the handshakes of the abstract model extended with the interface to the low-level components, such as starting or stopping the belt motor (*motor ! start*, *motor ! stop*) and reading a sensor (*sen-*

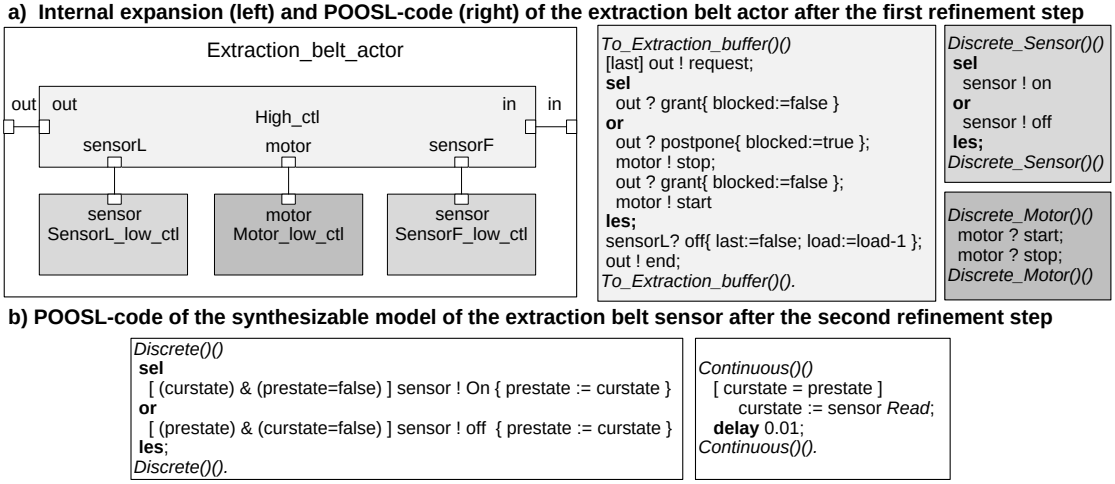

Figure 5: Second and third refinement steps of the discrete-event POOSL model

sorL ? off, *sensorL ? on*). The rightmost blocks, which model the low-level behavior, contain only the interface messages with the high-level control part.

4.3.3 Synthesizable Model

In the last refinement step, the model is extended with low-level continuous-time behavior and timing information. Figure 5b shows the POOSL-code of the low-level behavior of the extraction belt sensor. The low-level *sensorL* (Figure 5a, left), which previously only contained the local discrete event behavior is now split up into two concurrent processes: a discrete process, containing the interface to the high-level control, and a continuous process for updating the system state by reading the sensor. After this step, the model is ready for property-preserving synthesis and integration with the loop-control software.

4.4 Loop Controller Design

The feedback loop controllers and motion profiles for the Production Cell setup are designed and modeled in 20-sim, following the 4 step approach from Section 3.2.2. Because all 6 PCUs need only local loop controllers the loop controller designs are separately made for each PCU, resulting in 6 single-PCU (physical system) models containing the required digital (discrete-time) PD and PID loop controllers and motion profiles, designed to run at a sample frequency of 1 kHz (see for more info [vdB06]). Furthermore, a safety layer was added to limit output signals and to shape input signals in order to get robust and safe PCU loop controllers.

4.5 Integration

In order to combine the controllers and motion profiles with the event triggered sequence control layers within all 6 PCUs (based on the interface agreements), they are extended (in the 20-sim model) with start/stop and finished signals (data-flow), mapped to rendezvous channel communication events in POOSL. A special POOSL data class code synthesis template was used in 20-sim to translate the controllers and motion profile as Rotalumis building blocks. FPGA I/O drivers are written, and separately tested, to access the sensors and actuators on the setup. A

periodic time event loop at 1 kHz is used in POOSL to provide the controllers with the required 1 kHz sample time. A correct working discrete-event discrete-time interaction, the interfaces and the code synthesis templates were tested independently and concurrently using a few small test examples.

5 Results and Discussion

The case experiment was set-up with minimal design interaction during the concurrent design stage. The *total* ECS software design time from specification to integration and testing was less than 10 days. The final ECS software integration up to the last on-target tests were performed in less than 2 days without any serious integration issues, resulting in a properly working production cell setup. The 12 days in total are for this setup a major improvement compared to an earlier bottom-up ECS design test case on the same setup which took more than 6 weeks [GB09]. Only a few minor timing details needed to be addressed, like tuning the required delays between sensor events and the execution of the corresponding actions (for example, the electromagnet on/off response was slower than modeled and dependent on the exact block orientation, which was not modeled at all). No other software integration issues were seen.

Essential for the concurrent way of working with minimal interaction is that the designers have the attitude to focus on their own partition during the concurrent design phase, but, at the same time think across the boundaries of their discipline to foresee potential integration issues resulting from local design choices and to ensure that they are covered by inter-disciplinary information exchange and small early integration tests. Models (ranging from abstract drawings towards executable models) are essential for this information exchange. The usage of natural languages can result in misunderstandings, because different disciplines often share the same terminology, but with a different meaning.

After the joint work on the abstract system level model and the partitioning, the only information exchanged for this case during the concurrent design stage are abstract interfaces between the partitions, their meaning, the required behavior (e.g. events, signals) and their stepwise refinements (e.g. data type, units and timing requirements). Besides this information exchange is it also essential to synchronize the expected behavior of the (other) partitions. The usage of local refinement ensures that the externally observable behavior remains the same and thus limits the frequency of information exchange.

The current checks on the consistency of interfaces between the partitions were done manually, but we are working on automated consistency checking (i.e. for interfaces and shared parameters) at the model level, using the abstract top-level model as basis. We want to use the information in this model also to facilitate a further early integration between models using co-simulation facilities. For this case, we could have re-used the physical system models to test our final (timed) ECS software against a *virtual prototype* of the Production Cell setup using co-simulation (see our work in [GDB08]). Advantage is for example that we can introduce faults and test safely without damaging our real setup. The disadvantage is that this additional integration step requires extra time resulting again in the trade-off between fully integrated design and design time. However, additional (automated) tool support can shorten the additional spend time. For this setup we did not need this step because the behavior of all ECS software parts was already verified using simulations and small on-target tests.

6 Conclusions

The presented case study, demonstrating the concurrent model-driven design of ECS software for a real-world mechatronic system, shows that application of our methodology has resulted in a predictable realization with a short integration time, with a good balance between integrated design and separation of concerns during the concurrent design stage. We have shown that with a more concurrent approach we can still design reliable ECS software in a quick and efficient way. Depending on the total design time, the time between two integration tests and the project size, extra model-level integration steps can become beneficial, essential or a waste of time. Despite the promising developments in the area of integrated heterogeneous multi-paradigm modeling, the trade-off between integrated design and efficient concurrent design should be made separately for every design project. The presented methodology is not limited to an ECS software implementation. The same setup is for example also running entirely from an FPGA, created using the same approach [GB09].

Bibliography

[Ben94] S. Bennett. *Real-time Computer Control : An Introduction*. Prentice Hall, New York, 2nd edition, 1994.

[vdB06] L. van den Berg. Design of a Production Cell Setup. MSc thesis 016CE2006, University of Twente, Control Engineering, 2006.

[BGVO07] J. Broenink, M. Groothuis, P. Visser, B. Orlic. A Model-Driven Approach to Embedded Control System Implementation. In *Western Multiconference on Computer Simulation WMC 2007, San Diego*. Pp. 137–144. SCS, San Diego, January 2007.

[vB02] L. van Bokhoven. *Constructive Tool Design for Formal Languages from semantics to executing models*. PhD thesis, TU Eindhoven, The Netherlands, 2002.

[Con09] Controllab Products. 20-sim website. 2009. http://www.20sim.com

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, L. Jie, X. Liu, S. Ludvig, S. Neuendorffer, S. Sachs, Y. Xiong. Taming Heterogeneity—the Ptolemy Approach. In *Proceedings of the IEEE*. Volume 91(1), pp. 127 – 144. Jan. 2003. doi:10.1109/JPROC.2002.805829

[GB09] M. Groothuis, J. Broenink. HW/SW Design Space Exploration on the Production Cell Setup. In *Communication Process Architectures 2009, Eindhoven, The Netherlands*. IOS Press, Amsterdam, Nov. 2009. To be published.

[GDB08] M. Groothuis, A. Damstra, J. Broenink. Virtual Prototyping through Co-simulation of a Cartesian Plotter. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation, 2008*. Pp. 697–700. IEEE Industrial Electronics Society, Sept. 2008. doi:10.1109/etfa.2008.4638472

[GVP+00] M. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven, M. Stevens. Object-Oriented Modelling and Specification using SHE. 2000.

[HVC07] J. Huang, J. Voeten, H. Corporaal. Predictable real-time software synthesis. *Real-Time Syst.* 36(3):159–198, 2007. doi:10.1007/s11241-007-9013-6

[HVG+07] J. Huang, J. Voeten, M. Groothuis, J. Broenink, H. Corporaal. A model-driven design approach for mechatronic systems. *Application of Concurrency to System Design, International Conference on* 0:127–136, 2007. doi:10.1109/ACSD.2007.40

[PJ97] P. van der Putten, J.Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.