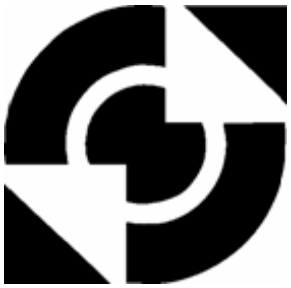
University of Twente

EEMCS / Electrical Engineering
Control Engineering



An integrated embedded control software design case study using Ptolemy II

Kees Verhaar

MSc report

Supervisors:
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
ir. M.A. Groothuis

May 2008

Report nr. 011CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

The heterogeneous nature, together with the increasing complexity of embedded systems raises the need for design tools that support integrated functional verification. Current design tools focus on one specific step in the design process (e.g. system dynamics modeling, control law design or software design), making integrated verification difficult. Also, manual model transformations need to be performed when moving to the next design phase, which takes a lot of time and can introduce errors. This makes iterative design difficult and error prone.

Instead of using different tools for each design step, an integrated approach, using a single modeling framework, can be used. This approach solves the problems of integrated verification and iterative design. In this project a case study on this integrated approach is performed using Ptolemy II as an integrated development platform and the Production Cell setup as a practical test case.

First, a feasibility study is conducted to explore the possibilities and limitations of Ptolemy II. A simple version of embedded control software for the Production Cell setup is developed. This shows that Ptolemy II has limited facilities for system dynamics modeling and control law design. Also, automatic code generation is still experimental and cannot be used for all models. However, when taking these limitations into account, correctly working embedded control software for the Production Cell can be created.

Next, a well-structured model of the Production Cell setup is created, in order to further explore the Ptolemy approach. This model uses the communication structure proposed in van Zuijlen (2008). Everything essential for integrated functional verification of the behavior of the system is included in this model: a plant dynamics model, a controller model, a kinematic model of the aluminum blocks in the system and a 3D graphical animation. Loop controller performance is evaluated by means of simulation plots. The 3D graphical animation is used to verify correct sequence control and controller synchronization. The final result is a complete integrated model of the Production Cell setup, showing correct behavior. Automatic code generation is used to produce C code which is compiled and run on the target PC/104 platform, resulting in a completely functional real setup.

Using an integrated approach for embedded control software solves the problems of integrated testing and iterative design, but requires a generic tool which cannot offer all specific features required for each design step. Therefore, the Ptolemy method (the integrated approach) is compared to four other methodologies used in embedded control software design: the co-simulation approach, the CE-method, the Matlab/Simulink approach and the POOSL approach. Focussing on embedded control software development for mechatronic systems, the CE-method is a good choice, although it still needs improvement. These improvements can be made by using techniques found in Ptolemy II. 20-Sim should be extended to support more models of computation, starting with Finite State Machines (FSM) and Discrete Event (DE), to support the modeling of a broader range of systems. Ptolemy II techniques can be used to ensure formal correctness when combining multiple models of computation in a single model. Incorporating the Ptolemy II code generation framework in 20-Sim and gCSP will allow automatic code generation for a wide range of target languages and language variants. Finally, including object-oriented techniques in modeling and improving the extendability of the CE-toolchain will enhance the usability.

Models of computation for closely related tasks should be integrated in a single tool. Integrated verification between tools can then be performed by using co-simulation. Ultimately, tool integration should be transparent to the user. This can be achieved by creating well-defined tool interfaces and a graphical user interface combining these tools. Well-defined tool interfaces also facilitate in-the-loop simulation, further reducing the gap between model and realization.

Samenvatting

De heterogene aard, in combinatie met de toenemende complexiteit van embedded systemen leidt tot de behoefte aan ontwerpsoftware die geïntegreerd functioneel testen ondersteunt. De huidige pakketten concentreren zich op een specifieke stap in het ontwerp proces (b.v. modelleren van dynamica, ontwerp van de regelaar of software ontwerp). Dit maakt geïntegreerd testen moeilijk. De overgang tussen ontwerpfasen vereist handmatige model transformaties, wat veel tijd kost en fouten introduceert. Dit bemoeilijkt iteratief ontwerpen.

In plaats van verschillende ontwerppakketten voor de diverse ontwerpfasen te gebruiken, kan een geïntegreerde aanpak, ondersteund door een geïntegreerd software *framework* gebruikt worden. Deze aanpak lost de problemen van geïntegreerd testen en iteratief ontwerpen op. Dit rapport presenteert een *case study* naar deze geïntegreerde aanpak, gebruik makend van Ptolemy II als ontwerp software en van de Production Cell opstelling als praktische *test case*.

Er is een haalbaarheidsstudie gedaan naar de mogelijkheden en beperkingen van Ptolemy II. Hiervoor is een eenvoudige versie van de software voor de Production Cell ontwikkeld. Hieruit worden de beperkingen van Ptolemy II op het gebied van modelleren van dynamica en voor het ontwerpen van regelalgoritmen duidelijk. Automatische codegeneratie is nog experimenteel en is niet bruikbaar voor alle modellen. Echter, wanneer rekening wordt gehouden met deze beperkingen kan werkende embedded software voor de Production Cell gemaakt worden.

Vervolgens wordt een gestructureerd model van de Production Cell gemaakt om de mogelijkheden van de Ptolemy aanpak verder te onderzoeken. Dit model maakt gebruik van de communicatiestructuur zoals gepresenteerd in (van Zuijlen, 2008). Relevante aspecten, nodig voor verificatie van het systeem zijn opgenomen. De prestaties van de PID regelaars worden beoordeeld aan de hand van simulatieplots. Een 3D animatie wordt gebruikt om correct gedrag van de *sequence controllers* en de synchronisatie te beoordelen. Het eindresultaat is een compleet geïntegreerd model van de Production Cell opstelling dat correct gedrag vertoond. Automatische codegeneratie wordt gebruikt om C code te genereren die gecompileerd en vervolgens op de opstelling gedraaid wordt. Dit resulteert in een correct werkende opstelling.

Een geïntegreerde aanpak voor de ontwikkeling van software voor embedded systemen lost de problemen van geïntegreerd testen en iteratief ontwerpen op. Echter, een dergelijke aanpak vereist generieke ontwerpsoftware die niet alle specifieke opties nodig voor elke ontwerpstap kan bieden. Daarom wordt de Ptolemy methode (de geïntegreerde aanpak) vergeleken met vier andere ontwerpmethoden: de co-simulatie aanpak, de CE-methode, de Matlab/Simulink aanpak en de POOSL aanpak. Voor mechatronische systemen is de CE-methode een goede keuze, hoewel deze wel verbeterd moet worden. Verbeteringen kunnen worden gemaakt door gebruik te maken van technieken uit Ptolemy II. 20-Sim moet uitgebreid worden met ondersteuning voor meer rekenmodellen, beginnend met *Finite State Machines (FSM)* en *Discrete Event (DE)*, zodat een breder scala aan systemen gemodelleerd kan worden. Technieken uit Ptolemy II kunnen gebruikt worden om formele correctheid van modellen die meerdere rekenmodellen combineren te garanderen. Automatische codegeneratie voor een breed scala aan programmeertalen wordt mogelijk door het gebruik van het Ptolemy II codegeneratie systeem in 20-Sim en gCSP. Tenslotte kan de gebruikersvriendelijkheid van de CE-methode verbeterd worden door object oriëntatie in modellen en door de het uitbreiden van de software eenvoudig te maken.

Rekenmodellen voor sterk gerelateerde taken moeten in één pakket geïntegreerd worden. Voor geïntegreerde verificatie tussen pakketten kan gebruik gemaakt worden van co-simulatie. Uiteindelijk moet de integratie van ontwerpsoftware transparant voor de gebruiker zijn. Dit kan bereikt worden door goede interfaces tussen ontwerppakketten te definiëren en een overkoepelende gebruikersinterface te maken. Goede interfaces tussen ontwerppakketten maakt ook 'in-the-loop' simulatie mogelijk, wat het gat tussen model en realisatie verder verkleint.

Preface

This report marks the final step in my life as an Electrical Engineering student at the University of Twente. It has been a great learning experience for me, both on an academic as well as a personal level. Being able to take part in projects outside of the Electrical Engineering curriculum, such as the board of E.T.S.V. Scintilla, the media project of the 'Introductie Kommissie' and running my own business together with a good friend have certainly contributed to this.

I am thankful for all the opportunities I had, and the people who supported me. First of all, I would like to thank Marcel Groothuis for his dedicated support during my MSc project as well as his useful comments. Special thanks also to Jan Broenink who provided me with this assignment and allowed me to develop my own research ideas. Also, I would also like to thank the rest of the people at the Control Engineering group, especially my fellow MSc students, for providing a pleasant working atmosphere and lively discussions that helped me to improve my work.

I would also like to thank my family. Without their continuing support I would not have made it to where I am now. Last, but certainly not least, my thanks go out to all my friends, especially those that started their studies at the 'Vestiging Friesland' in 2001. You have certainly made my time here more enjoyable.

Now that my time as a student has come to an end, new challenges lie ahead. Having successfully completed my MSc assignment, I feel ready and am looking forward to facing them.

Kees Verhaar
Enschede, May 2008

Contents

1 Introduction

This chapter starts by presenting the problem dealt with in this project. Next, the project context is given. The chapter concludes by presenting the approach followed in this project and an outline of the rest of this report.

1.1 Problem statement

In general, the process of system design can be described by the pyramid in figure 1.1. By exploring alternatives and making design choices at decreasing abstraction level (and increasing detail level) an idea is translated into a final product.

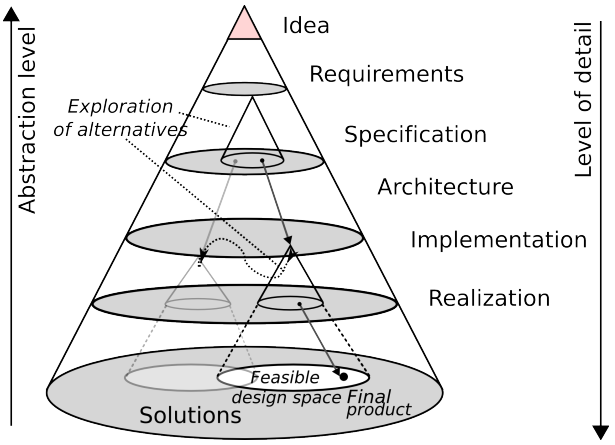


FIGURE 1.1 - Generic system design process (Corporaal, 2006)

For embedded control system design the generic design process translates into the design process in figure 1.2. The process of developing embedded control software can be divided into four steps: Physical System Modeling, Control Law Design, Embedded System Implementation and Realization. In general, there is no one-to-one relation between the design steps and the tools used in the design process.

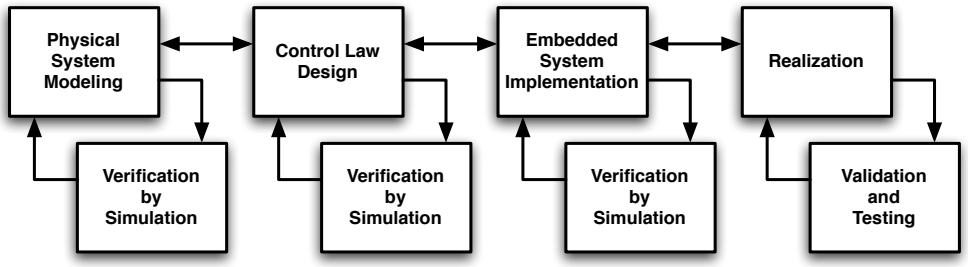


FIGURE 1.2 - Embedded Control Software design process (Broenink and Hilderink, 2001)

Ideally, the result of each design step is verified by simulation, which requires the results of previous design steps. For example, in order to verify the control law design a model of the physical system (the plant) is required. The usage of different tools for each design step introduces the problem of integrated testing. For example: how can we verify the control law designed using one tool (e.g. Matlab) using the physical system model designed using another tool (e.g. 20-Sim)? As the design process progresses more tools are involved and this problem gets worse. A solution should be found, so that integrated testing is possible at all stages of the design process.

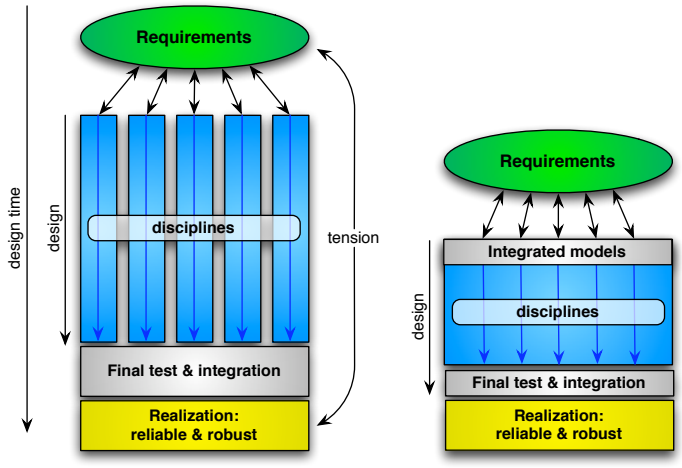
A second issue arises when an iterative design cycle is used in the design process. In iterative design, short design cycles are made where the design is refined at each cycle. In order to evaluate the consequences of changes in one design step in the complete design, tight integration of the four design steps (and thus the corresponding tools) is required. Current tools, designed for a specific part of the design trajectory, cannot offer such integration. The transition of one design step to another will now require a lot of effort, as well as introduce errors, because of the manual model transformations that need to be performed. These transformations should be taken care of by the design toolchain to overcome these issues.

Several approaches can be taken to overcome the problems identified here. One approach is to couple the design tools used by a co-simulation interface (Damstra, 2008). Another approach is to integrate the models used at the various design stages into a single modeling framework. This last approach is explored in this project, using the Ptolemy II modeling framework.

1.2 Project context

1.2.1 ViewCorrect

This project is part of the ViewCorrect research project. The purpose of the ViewCorrect research project is to provide methodological support, including (prototype) tools, for the predictable design of distributed hard real-time embedded control systems for mechatronic products. The methodology consists of three major components: views, multidisciplinary core models and correctness-preserving code generation. The views allow designers from different engineering disciplines to interactively and concurrently work on the design of the complete system, while each team member can observe and understand the impact of design decisions of others through his own, well-known view. Views representing software components are mapped onto the target hardware platform automatically, in a correctness-preserving way. This methodology aims to relax the tension between design cost and design time on the one hand and quality (in particular reliability and robustness) on the other hand. Figure 1.3 illustrates the ViewCorrect approach, as opposed to the traditional approach (Broenink et al., 2005).



(a) Traditional approach (b) ViewCorrect approach

FIGURE 1.3 - ECS design approaches

1.2.2 Ptolemy II

Ptolemy II is a software framework developed as part of the Ptolemy Project (Eker et al., 2008) at the University of California, Berkeley. The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The key underlying principle in the project is the use of well-defined models of computation that govern the interactions between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation.

Ptolemy II is a Java-based component assembly framework with a graphical user interface called Vergil. It includes a growing suite of domains, each of which realizes a model of computation. Figure 1.4 gives an impression of the Ptolemy II design environment.

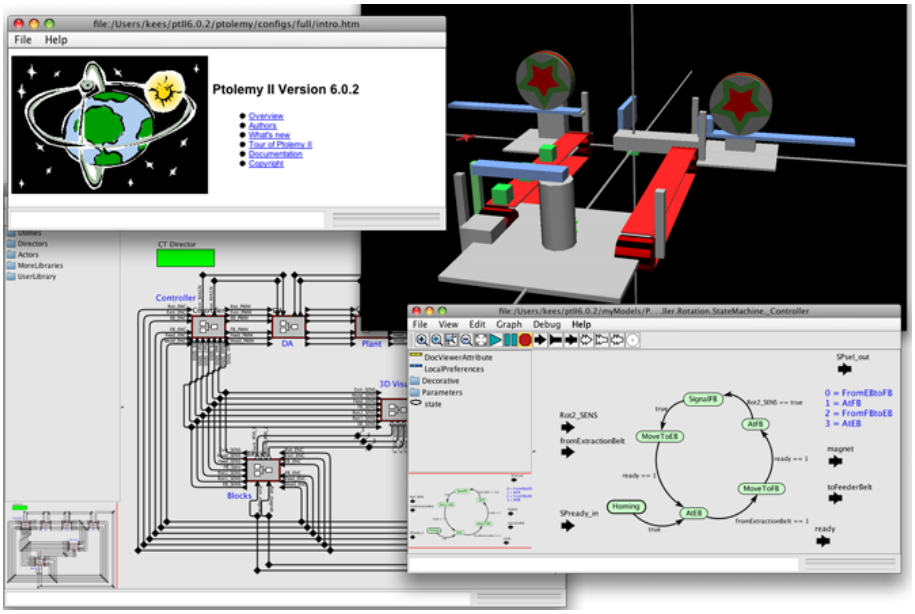


FIGURE 1.4 - An impression of the Ptolemy II design environment

1.2.3 The production cell setup

As a practical case study the production cell setup is used. This setup was developed by Bert van den Berg to serve as a test platform for embedded control software development and as a demonstration setup (van den Berg, 2006). Figure 1.5 shows an image of the production cell setup. The six units in the setup can operate independently, but do need synchronization to achieve desirable behavior of the complete system.

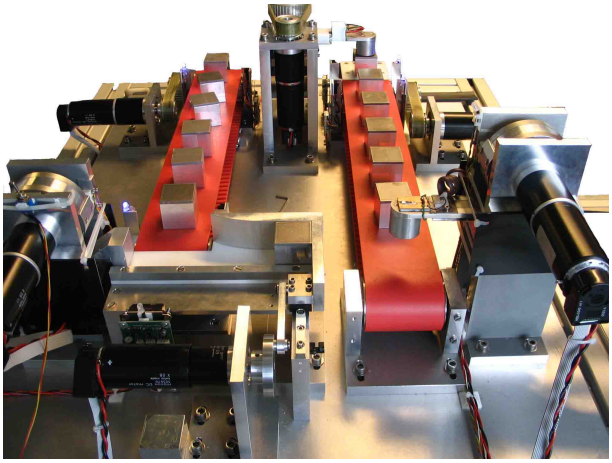


FIGURE 1.5 - The production cell setup

1.3 Assignment approach

As mentioned in section 1.1, this project aims to explore a design approach where all design steps are supported by a single modeling framework. The modeling framework used is Ptolemy II. The goal is to see whether or not such an integrated approach is useful for embedded control software development.

The assignment is structured as follows:

- The research starts with a feasibility analysis of Ptolemy II as an Embedded Control Software development platform for mechatronic systems. Simple controller software for the production cell setup is developed to evaluate required features for embedded control software design, such as continuous time modeling, discrete time modeling and code generation;
- The second phase of the project focuses on developing a model of the production cell system. The knowledge gained in the feasibility analysis will be used to create a well-structured hierarchical model which employs the right model of computation at each hierarchical level. From this practical test-case it should become clear whether the Ptolemy approach is useful for embedded control software development for mechatronic systems;
- This project is concluded with a discussion on the Ptolemy method and its corresponding tool (Ptolemy II). This includes a comparison of the Ptolemy method to other methods and their corresponding tools. The methods and tools used for this comparison and the reasons for choosing them are elucidated in section 1.3.1.

1.3.1 Methods & tools to compare

The four methods that will be used in the comparison with the Ptolemy method are:

- Co-simulation: as mentioned in paragraph 1.1, using co-simulation can be used to overcome the problems of integrated testing and model transformation in iterative design. Therefore, the co-simulation approach is included in the comparison. A co-simulation backplane approach using CosiMate as a co-simulation tool, as described in (Damstra, 2008) is used;
- CE-toolchain: the CE-toolchain consists of 20-Sim (Controllab Products B.V., 2008), a tool for physical system modeling and controller design, gCSP (Jovanovic et al., 2004), a tool for the graphical design of embedded software using the CSP process algebra (Hoare, 1985) and the CTC++ library that supports multithreaded real-time programming in C++. This toolchain is developed at the Control Engineering laboratory at the University of Twente and is an alternative to Ptolemy II;
- POOSL: a high-level approach of embedded software development is supported by the POOSL language (Huang et al., 2007). POOSL is a system-level description language that is used for complex systems. It is developed at the University of Eindhoven and is also used in the View-Correct project, which is why it is included in this comparison;
- Matlab/Simulink: The industry standard for modeling and control software development is Matlab/Simulink by The MathWorks (The Mathworks, 2008). It is included in the comparison to give an idea how the aforementioned (experimental) methods and tools relate to the main approach used in the industry today.

1.4 Report outline

Chapter 2 presents background information on Ptolemy II and the Production Cell setup. In chapter 3 the results from the feasibility analysis are shown. Next, chapter 4 shows the design of the integrated Production Cell model. Results of loop controller verification, functional verification and realization are presented in chapter 5. Chapter 6 discusses the integrated design approach which leads to an overview of other commonly used methodologies and tools and suggestions for improving the CE-method and toolchain. Finally, chapter 7 presents the conclusions from this project and some recommendations for future work.

Additional information is given in three appendices. Appendix A describes how to create a custom actor for use in the Ptolemy II modeling environment. Also, the process of creating a so-called *helper* for code generation is explained. Appendix B shows some simulation results, in addition to those found in chapter 5. Appendix C describes how to create an AVI file from the 3D graphics animation in Ptolemy II.

2 Background

This chapter provides more background information on the modeling framework used in this project, Ptolemy II, and on the production cell setup, which is used as a practical test case.

2.1 Ptolemy II

In this section the essential features of Ptolemy II are described in more detail. First, the structure of a Ptolemy II model is described. Next, the mechanism which makes it possible to use multiple models of computation in a single model is explained. Then the framework for automatic code generation is explained and finally some examples of applications using Ptolemy II are shown.

2.1.1 Model structure

A model in Ptolemy II is built up through a hierarchical ordering of *actors*. A Ptolemy II actor is comparable with a 20-Sim submodel. Actor-oriented design contrasts with (and complements) object-oriented design by emphasizing concurrency and communication between components (actors). Like objects, actors have well defined interfaces that consist of *ports* and *parameters*. Ports represent a point of communication with other actors, while a parameter is part of the configuration of an actor. Figure 2.1 gives an example of a Ptolemy II model generating a sine wave.

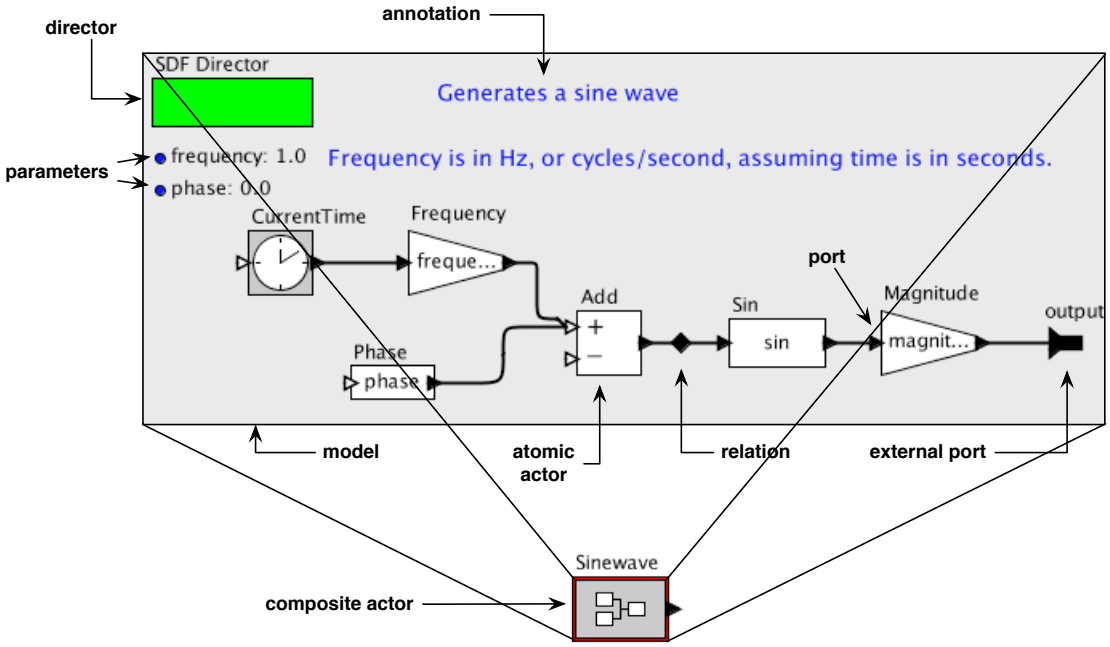


FIGURE 2.1 - Illustration of an actor-oriented model in Ptolemy II

In an actor-oriented design components (actors) communicate by sending messages through *channels*, connected to their ports. This is in contrast with object-oriented design, where components interact primarily through method calls. Channels can also be bi-directional. A channel can be split by using a *relation*. The use of channels for communication implies that actors interact only with the channels they are connected to and not directly with other actors.

Similar to an actor a model may also define an external interface. This interface is made up of external ports and parameters, which are different from the ports and parameters of actors inside the model. This interface is called the model's *hierarchical abstraction*. The hierarchical abstraction can be used in the same way as an actor, on a higher hierarchical level.

Actors can either be *atomic* or *composite*. An atomic actor is at the lowest level, comparable to a 20-Sim equation submodel. A composite actor contains other actors, either atomic or composite and is comparable to a 20-Sim graphical submodel. Model hierarchy is realized through the use of composite actors.

In Ptolemy II actors (both atomic and composite) can be converted to a *model class*. Such a model class provides advantages similar to classes known from object-oriented design; model classes can be instantiated as many times as desired and model subclasses can be defined which inherit from a parent model class (actor). This feature relates only to modeling, not to code generation.

The Ptolemy II modeling framework supports *data and domain polymorphism*. Data polymorphism allows the design of actors that can operate on any of a number of input data types. For example, the AddSubtract actor can accept any numeric type of input. Domain polymorphism allows the design of actors that can operate in several models of computation. The method for communicating across channels is the main issue here. For example, in the CSP domain channel communication is rendezvous based while in the SDF (Synchronous Data Flow) domain sending and receiving data is synchronized (i.e. all actors send a data token simultaneously).

It is important to note that the syntactic structure of a model as discussed here says very little about the semantics, i.e. the meaning, of a model. A channel connecting two actors means that these two actors interact in some way. The exact semantics are determined by the model of computation, which might give operational rules for executing a model, determining when actors perform internal computations, update their state, perform external communication, etc. Which model of computation is used at which hierarchical model level is indicated by the *director*. Each hierarchical level in the model can have a different director.

2.1.2 Heterogeneous modeling

One of the key concepts in Ptolemy II is that of *heterogeneous modeling*, meaning that a single model can contain multiple models of computation. This is implemented by using a hierarchical model structure in which a director determines which model of computation to use at each hierarchical level. Figure 2.2 depicts a typical model structure. In this case, E1, E4 and E3 are atomic actors, while E2 is a composite actor. A composite actor introduces a deeper hierarchical level and can therefore be governed by a different model of computation. This is denoted by the local director D2, which may be different from D1. A composite actor may or may not have its own local director. If it has a local director, then it is defined to be opaque. An opaque composite actor is directed by its local director while a non-opaque composite actor is directed by the director of its parent (its *executive director*). When any action method is called on an opaque composite actor, the composite actor will generally call the corresponding method in its local director. This interaction is crucial, since it is domain independent and allows for communication between different models of computation.

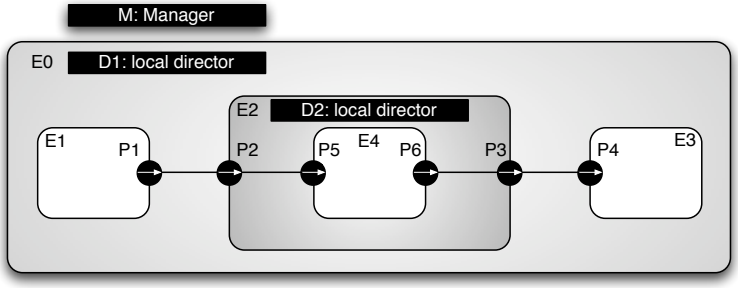


FIGURE 2.2 - Example showing a typical arrangement of actors, directors and managers

Communication among actors can only take place via channels, connected to ports (P1, P2, P3, etc. in figure 2.2). Each port uses a *receiver*, according to the domain it is in (determined by the director). This receiver determines the communication protocol to be used. Examples of receivers are a *mailbox* receiver, which has capacity for a single data token, a *FIFOQueue* which implements a first-in, first-out queue or a *CSPReceiver* which implements rendezvous behavior.

One execution of an actor consists of the execution of three actor methods: `prefire()`, `fire()` and `postfire()`. These methods are called by the director according to a schedule appropriate for its model of computation. The execution of a complete model will now be illustrated by using the model in figure 2.2 as an example. A visual representation of the execution in the form of a sequence diagram is shown in figure 2.3.

1 D1 invokes `prefire()`, `fire()` and `postfire()` of E1, E2 and E3;
2 The `fire()` method of E2 transfers a token from P2 to P5 by delegating to the local director D2 and invoking its `transferInputs()` method;
3 E2 then invokes `fire()` of D2, which in turn invokes `prefire()`, `fire()` and `postfire()` on E4;
4 E4 sends its resulting token to P3 (not to P4, because E2 is an opaque composite actor);
5 Finally, the `fire()` method of E2 delegates to its executive director (which is D1) and invokes its `transferOutputs()` method which transfers the token from P3 to P4.

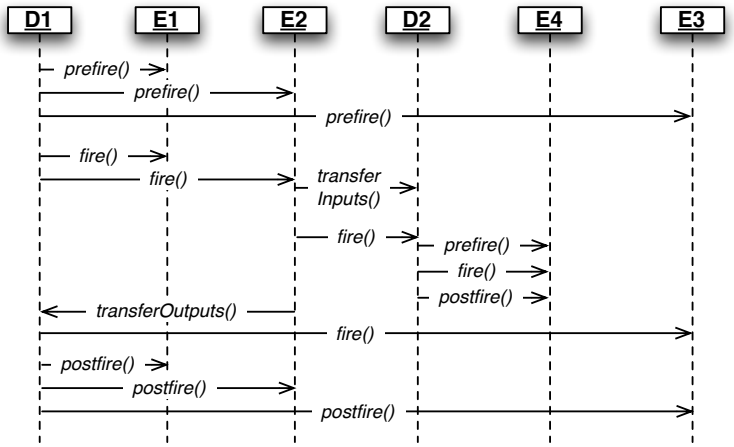


FIGURE 2.3 - Sequence diagram of the execution of the model in figure 2.2

A composite actor delegates transfer of its inputs to its local director, and transfer of its outputs to its executive director. This is the correct organization, because in each case, the director appropriate to the model of computation of the destination port is the one handling the transfer. It can therefore handle it in a manner appropriate to the receiver in that port.

It should be noted that no automatic data conversion takes place at ports on the boundary between two domains. As an example assume that D1 is a continuous time director while D2 is a discrete time director. At P2 this would require a sampler while at P3 a reconstructor is required. These should be inserted manually, which leaves these design choices open for the user, who may want to investigate the effects of using a first order hold instead of a zero order hold as a reconstructor. Failure to take appropriate measures at domain transitions may yield unexpected results.

The current version of Ptolemy II (6.0.2) supports a broad range of domains and their corresponding models of computation. Domains that are reasonably mature:

- CT: continuous-time modeling;
- DDF: dynamic dataflow;
- DE: discrete-event modeling;

- FSM: finite state machines and modal model;
- PN: Kahn process networks;
- Rendezvous: synchronous message passing;
- SDF: synchronous dataflow;
- SR: synchronous reactive;
- Wireless: wireless modeling.

Domains that are still experimental:

- CI: component interaction (push/pull);
- Continuous: continuous-time modeling (improved version);
- CSP: Communicating Sequential Processes;
- DDE: distributed discrete events;
- DT: discrete time;
- Giotto: periodic time-driven;
- GR: 3D graphics;
- HDF: heterochronous dataflow;
- PetriNet: Petri Net modeling;
- PSDF: parameterized synchronous dataflow;
- TM: timed multitasking.

Due to the experimental nature of Ptolemy II, multiple implementations for a single model of computation may exist (e.g. CT and Continuous for continuous-time modeling). For an in-depth description of these domains, refer to (Brooks et al., 2007).

2.1.3 Code generation

Automatic code generation is an essential part of the ViewCorrect methodology. The code generation framework in Ptolemy II uses a *helper* based mechanism. A helper is responsible for generating target code for a Ptolemy II actor. Each actor has one helper for each target language that is supported (C, VHDL, etc.). A helper consists of two files: a Java class file which determines which code blocks to insert based on actor instance-specific information (e.g. port datatype, parameter value) and a code template file which includes the target code blocks for the various actor methods.

The code generator kernel uses the directors in the model to compute an order of execution of the actors in the model and then uses the helper Java classes to harvest code blocks from the code template files. A macro language is used to allow the usage of instance-specific information in the code template files.

The Ptolemy II code generation framework is based on *partial evaluation* (Jones et al., 1993). Partial evaluation is used as a code generation technique for transforming an actor-oriented model into target code while preserving the models semantics.

A schematic overview of the code generation process is shown in figure 2.4.

In the current Ptolemy II version (6.0.2) the code generation framework supports the Synchronous Dataflow (SDF), Finite State Machine (FSM) and the Heterogeneous Dataflow (HDF) domains. An extensive set of actors have corresponding helpers for the C target language and some even for VHDL. However, a large set of actors remains for which no helpers exist.

2.1.4 Current usage

Ptolemy II has been under development since 1996 and has been used in many applications. In (Baldwin et al., 2004) Ptolemy II has been used as a basis for a modeling and simulation framework for wireless sensor networks, called *VisualSense*. Here, Ptolemy II has been extended with wireless channel models, sensor node models and a wireless director based on the Discrete Event model of computation. In (Kienhuis et al., 2000) a tool is developed to transform a Matlab program into a process network specification in order to reveal parallelism and

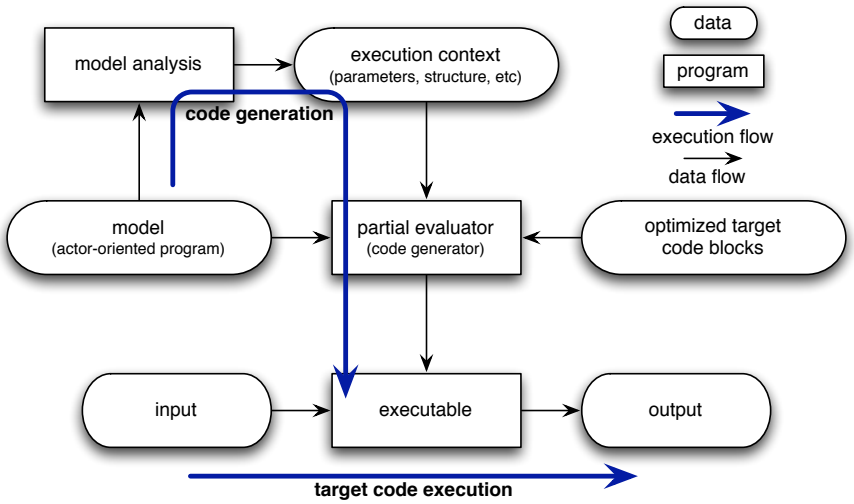


FIGURE 2.4 - Schematic overview of the code generation process in Ptolemy II

facilitate the mapping onto a hardware architecture. Ptolemy II is used as a verification tool, as it allows the simulation of process network specifications.

In (Dumont and Boulet, 2005) a multi-dimensional version of SDF, called Array-OL, is proposed. In order to have a simple, yet efficient simulation tool Ptolemy II is extended with specific actors and a director for Array-OL.

As a final example the work in (Martin et al., 2003) is mentioned. Here a design framework for wearable electronic textiles is proposed. Ptolemy II is used as a simulation framework to integrate models of the physical environment, human locomotion, sensor behavior, network communication, power consumption and software execution.

2.2 The production cell setup

The production cell setup resembles a Stork plastics molding machine, together with units that feed raw materials into it and units that extract the finished products, forming a total of six actuated units. A schematic overview of the production cell setup is depicted in figure 2.5.

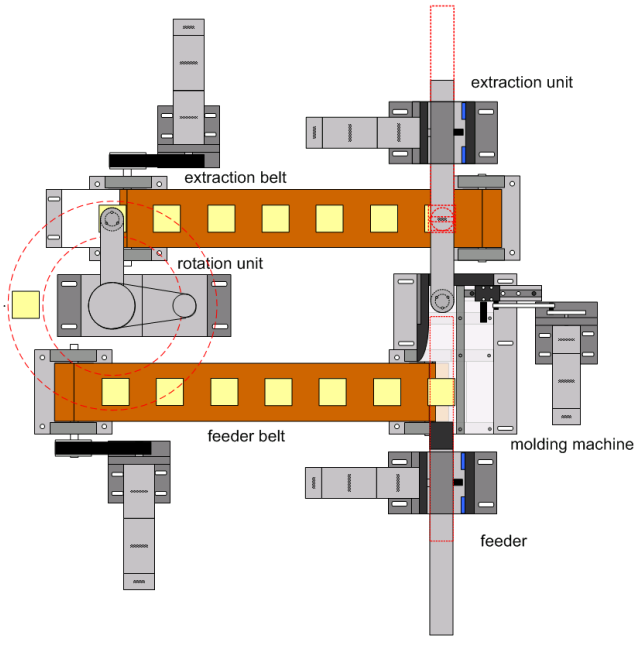


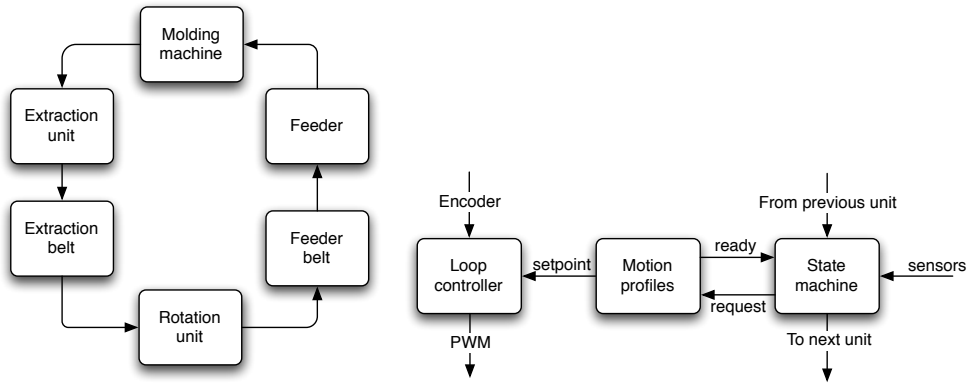
FIGURE 2.5 - Schematic overview of the production cell setup (van den Berg, 2006)

When in operation, the system feeds raw materials (represented by aluminum blocks) via the feeder belt and the feeder into the molding machine. When the door of the molding machine opens, the extraction unit will extract the finished product (represented by the same aluminum blocks) onto the extraction belt. In order to create a useful demonstrator the rotation unit is introduced, which moves the aluminum blocks from the extraction belt to the feeder belt so that the setup can run for an infinite amount of time.

The six units in the system all have individual controllers. However, in order to achieve desirable behavior, these controllers do need to synchronize their actions with each other. This means that some form of communication will need to take place among the six controllers.

Previous efforts for controlling the production cell have been undertaken. In (Maljaars, 2006) the CE-toolchain was used for this purpose. First, a dynamic model of each individual unit in the system is created and validated using 20-Sim. Next, 20-Sim is used to design PID control laws and finally the embedded control software design is done using gCSP. C code can then be generated that is compiled and run on the target machine, a PC/104 embedded PC running Linux with the RTAI real-time extensions.

A different approach is taken in (van Zuijlen, 2008). Here, the goal is to implement the embedded control software on an FPGA, instead of on a PC. This poses new challenges, such as the lack of a floating point unit on an FPGA and true parallel execution. Furthermore this research proposes a controller structure inspired by the structure of the real setup, depicted in figure 2.6.



(a) Top level controller structure (b) Structure of a single controller

FIGURE 2.6 - Production cell controller structure as proposed in (van Zuijlen, 2008)

In this project the models of plant dynamics developed in (Maljaars, 2006), as well as a simplified version of the controller structure proposed in (van Zuijlen, 2008) will be reused.

2.3 Conclusions

In this chapter background information on both Ptolemy II and the production cell setup has been given. The Ptolemy II model structure is explained, as well as the mechanisms by which heterogeneous modeling and code generation are implemented. Choosing the right model of computation at each hierarchical model level is crucial, as this choice determines the semantics (i.e. the meaning) of the model.

Previous work done on the production cell setup will be reused in this project. This includes the models of the plant dynamics developed by (Maljaars, 2006) and the controller structure proposed by (van Zuijlen, 2008).

The next chapter presents a feasibility analysis in which the theory on Ptolemy II explained in this chapter is tested in practice.

3 Feasibility analysis of Ptolemy II as an ECS development platform

In this chapter a feasibility analysis of Ptolemy II as an embedded control software development platform is presented. This feasibility analysis is performed by developing controller software for one unit of the production cell setup: the rotation robot. A simple controller is used, which can be adapted to control the other units in the setup. Each of the following sections deals with one step in the design process from figure 1.2.

3.1 Dynamic plant modeling

Dynamic plant models for the production cell setup were developed in (Maljaars, 2006). 20-Sim was used as a modeling tool. These models are used as a basis for the models of the dynamic plant behavior used in this project.

The Ptolemy II continuous time (CT) model of computation is used for the modeling of the plant dynamics. Models in the CT domain are described in the form of ordinary differential equations (ODEs):

$$\frac{d\mathbf{x}}{dt} = f(\mathbf{x}, \mathbf{u}, t) \tag{3.1}$$

$$\mathbf{y} = g(\mathbf{x}, \mathbf{u}, t) \tag{3.2}$$

The simulation of a continuous time system boils down to numerically solving the ODEs of the system. Ptolemy II implements some of the known methods for doing this, such as the forward Euler method, the backward Euler method and the 2(3) order Runge-Kutta method.

By default, Ptolemy II allows the modeling of a dynamic system through the following mechanisms:

- Building a block diagram using integrators, multipliers, adders, etc;
- Entering a transfer function of the form $H(s) = \frac{b(s)}{a(s)}$;
- Using a linear state space description of the form $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}, \mathbf{y} = C\mathbf{x} + D\mathbf{u}$;
- Any combination of the above methods.

This list can be extended by implementing custom actors and directors to include for example Ideal Physical Model (IPM) iconic diagrams and even bond graphs.

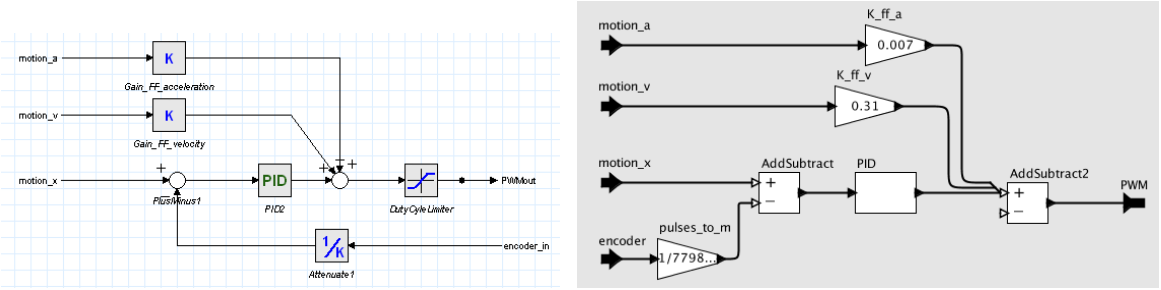
The models presented in (Maljaars, 2006) are in the form of IPM iconic diagrams and therefore cannot be used in a Ptolemy II model directly. Using 20-Sim, a model linearization was performed to obtain a transfer function ($H(s)$) that can be used in Ptolemy II. As long as the system operates in its normal working area, no significant non-linear effects are present and this model linearization is allowed.

3.2 Control law design

The next step in control software development is control law design. In (Maljaars, 2006, page 4) a standard PID control law is used in conjunction with feed-forward of acceleration and velocity signals. A similar controller structure with the same settings will be used in this project. Figure 3.1 shows this controller in 20-Sim and Ptolemy II.

In the actual setup, the controller will run on a standard PC/104 computer, which means it will run in the digital domain. The model of computation corresponding to this situation, Synchronous Dataflow (SDF), is used for the controller model. Another reason for choosing this model of computation at this stage is that the SDF domain is best supported by the Ptolemy II code generator.

No standard PID controller actor is available in Ptolemy II. A custom actor was developed which

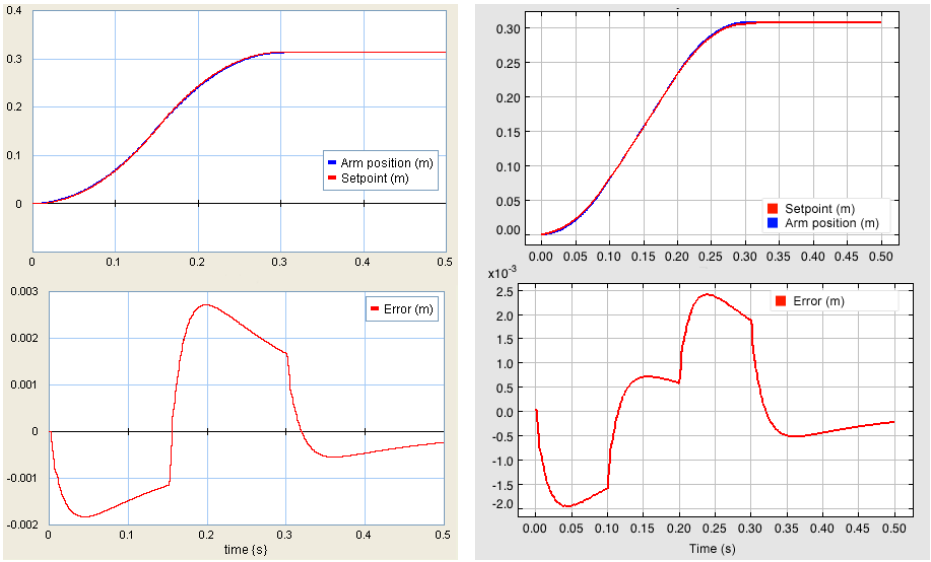


(a) 20-Sim (Maljaars, 2006) (b) Ptolemy II

FIGURE 3.1 - Loop controller design in 20-Sim and Ptolemy II

implements a PID control law. Developing a custom actor involves writing Java code that interacts with the Ptolemy II framework. More on developing custom actors can be found in appendix A.

This custom PID actor is used together with feed-forward signals for acceleration and velocity (figure 3.1(b)). In order to verify the control law design a simulation was performed. The resulting graphs, showing the generated controller setpoint, the actual arm position and the position error can be found in figure 3.2(b). Loop controller verification results from 20-Sim, obtained from the model by Maljaars (2006), are shown in figure 3.2(a). The difference in the results obtained from 20-Sim and Ptolemy II can be explained by the usage of a different motion profile generator (the 20-Sim motion profile generator is not available in Ptolemy II). In both cases the error does not exceed the maximum allowed error.



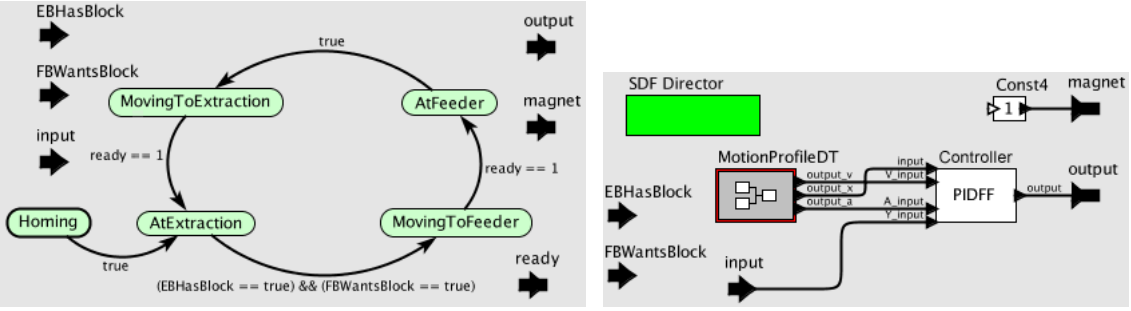
(a) 20-Sim (b) Ptolemy II

FIGURE 3.2 - Rotation robot control law verification in 20-Sim and Ptolemy II

3.3 Embedded system implementation

The control law for the loop controller, developed in the previous section, is used in conjunction with a sequence controller. Sequence control can best be described using a finite state machine (FSM). Ptolemy II includes a FSM director for modeling finite state machines, which is used here. Sequence control for the rotation robot is implemented according to the state machine in figure 3.3. The initial state is the 'Homing' state, which brings the rotation arm to the position at the extraction belt. The arm initiates its motion when there is an aluminum block at the extraction platform and the drop off position at the feeder belt is empty.

Each state in the finite state machine in figure 3.3(a) contains a *refinement*. An example of a refinement, for the 'MovingToFeeder' state is shown in figure 3.3(b). This refinement is an SDF model that determines the relation between input (shown on the left) and output (shown on the right) ports. In this case each refinement consists of the controller described in the previous section and a motion profile generator appropriate for each state.



(a) Finite state machine (b) Refinement for 'MovingToFeeder' state

FIGURE 3.3 - Rotation robot sequence controller

3.4 Realization

The first step in preparing the model for code generation is to connect model variables to I/O pins on the target hardware. Ptolemy II *LinkDrivers* were written for this purpose, comparable to the LinkDrivers found in gCSP. A LinkDriver is an actor that is inserted in the model in each channel that connects the controller and the plant. When the model is simulated the LinkDriver plays no role. However, when code is generated the LinkDriver code generation helper inserts the C code for performing I/O actions. Settings for LinkDriver type (PWM, encoder, etc.) and I/O channel can be made in the Ptolemy II graphical user interface.

The controller will run on a standard PC/104 stack running Linux with the RTAI real-time extensions enabled. The generated C code should include RTAI system calls to initialize real-time behavior, set up timers and to ensure a fixed control frequency. For this purpose the standard Ptolemy II C code generator was extended with a version that includes these RTAI specific system calls in the generated code.

The final step is to create a new model which contains only the controller. This is necessary because Ptolemy II can only generate code for an entire model, not for a selected submodel only. The generated code can then be compiled and run on the PC/104 target platform.

3.5 Results

First tests, which include only the rotation robot, look promising. The generated code runs on the PC/104 stack connected to the production cell setup and the system behaves as expected. The procedure for developing a controller for the rotation robot is repeated for the other five units in the system. The end result is a model that contains six controllers, for all six units. The code generated from this model can be compiled and run on the target platform, which results in a completely running Production Cell setup. There is no explicit synchronization between the six units. Essentially, synchronization of the controllers is now performed implicitly by the aluminum blocks triggering the sensor signals. The sensor signals are read using a polling mechanism running at the sampling frequency: 1.0 kHz. State transitions in the sequence controllers are triggered by these sensor signals.

3.6 Conclusions

The following conclusions can be drawn from this feasibility analysis. First of all: it is possible to use Ptolemy II as a embedded control software development platform. For the production cell setup all stages in the design process are completed successfully, resulting in controller software that behaves as desired. However, some remarks should be made. Dynamic plant mod-

eling in Ptolemy II is a difficult task. Transfer functions (either in $H(s)$ or in state-space form) have to be calculated by hand, which can be a lot of work and introduces errors. Furthermore, it is not a flexible method, as small changes in the system can require the transfer functions to be re-calculated. A graphical modeling method, like Ideal Physical Modeling iconic diagrams or bond graphs are more user friendly.

The controller structure used here uses only SDF and FSM models of computation, mainly because of limitations in code generation. Also, the controller structure is kept very simple. No explicit synchronization is performed, no safety features are included, all six controllers run in a single timed loop, etc. In order to further explore the Ptolemy method, particularly with respect to combining multiple models of computation, a better Ptolemy II model, including controllers, of the Production Cell system should be developed. This is done in the next chapter, where the controller structure proposed in (van Zuijlen, 2008) is used as a basis.

The code generation framework is found to still be experimental. It requires a separate model for code generation that contains only the parts for which code should be generated. At this stage, LinkDrivers are used for hardware I/O. This causes the model to become dependent on the target platform, which limits model reuse and this can therefore be considered to be an undesired way of working. A solution is to separate the generic model from the target specific code by creating stubs in the generic model which are filled in later, for example by using the 4C toolchain (Visser et al., 2007).

4 Design of the Production Cell model in Ptolemy II

In this chapter the design of a complete production cell model, based on the results obtained from the feasibility analysis in chapter 3 is presented. Section 4.1 discusses some design considerations that are the basis for the design. Next the actual design is presented, starting with the top-level model structure in section 4.2. The controller, plant and aluminum block models are discussed in the following sections. Finally, some conclusions on the design are drawn.

4.1 Design considerations

The model design discussed in this chapter incorporates the dynamic behavior of the plant and the discrete time controller in a single heterogeneous model. The design of the model is created with the possibilities and limitations of the used method and tooling in mind. For example, the model is strongly hierarchical which enables the use of multiple models of computation and the CSP domain is not used, as its Ptolemy II implementation is insufficient.

The goal of creating a model of the complete system is to support the design of the embedded control software. The design of the model should allow the functional verification of the controller design. With this goal in mind, certain assumptions have been made:

- Linear models of plant dynamics: when operating in its normal working area the plant behaves like a linear system. Therefore, non-linear effects like limitations on the motor current have been left out of the model;
- Aluminum block insertion only at extraction belt: for reasons of simplicity, the aluminum blocks are assumed to be positioned on the extraction belt when execution of the model starts;
- Kinematic model of aluminum blocks: for the aluminum blocks in the setup it is decided to only model their kinematics and to neglect their dynamic behavior. Most of the dynamic behavior is irrelevant for the verification of the functional behavior of the embedded controller;
- Safety: safety features have not been included in this model, as they are not essential for the correct behavior of the system;
- Elasticity of belts: the elasticity of the extraction belt and the feeder belt can cause undesired effects, such as bouncing of the aluminum blocks. However, as for the dynamics of the aluminum blocks, the elasticity of the belts is irrelevant for the controller design and therefore neglected.

4.2 Top-level model structure

The top-level structure of the production cell model in Ptolemy II is depicted in figure 4.1. At top-level, the model is governed by a Continuous Time director. This corresponds to the physical world, where the system operates in a continuous time environment.

The main control loop consists of four composite actors: a controller, which includes setpoint generators (top left), digital to analog conversion (DA), the plant and analog to digital conversion (DA). Because there are six units in the system, the main control loop is realized six times, which results in six channels connecting the composite actors in the main control loop. The CT director is capable of handling the mixed signal model (i.e. continuous and discrete time signals).

Besides the main control loop there are three more composite actors present. The 3D visualization (middle right) uses encoder values from the plant together with sensor status and position information from the aluminum block model to create a 3D animation of the system. This animation is used as an intuitive way of verifying the kinematic behavior. The Ptolemy II graphics actor library, in conjunction with a GR director, is used for the 3D animation.

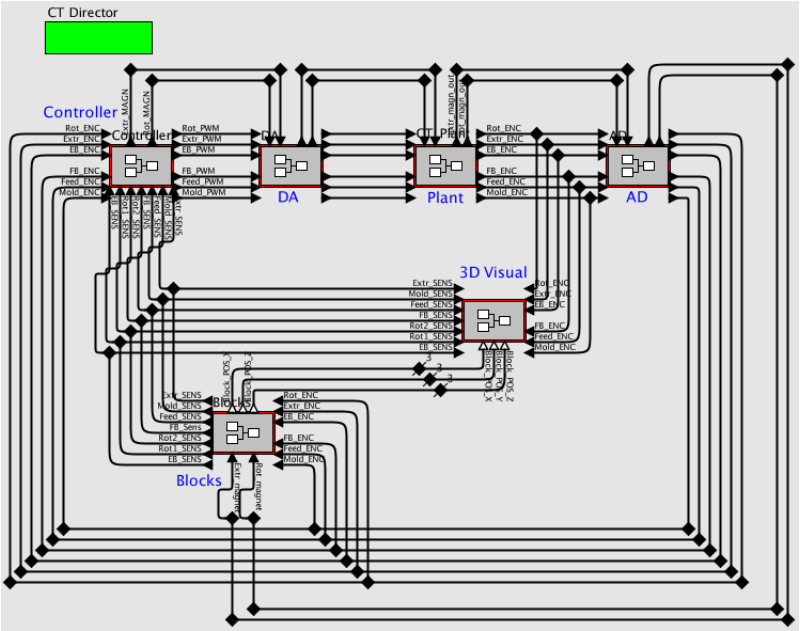


FIGURE 4.1 - Top-level structure of the production cell model

The block model (middle left) describes the kinematic behavior of the aluminum blocks in the production cell setup and will be explained in section 4.5.

4.3 Controller model

4.3.1 Hierarchical controller

The controller in the Ptolemy II model is built using the controller structure proposed by (van Zuijlen, 2008), see figure 2.6, as a basis. This structure is translated to a hierarchical Ptolemy II model. At each hierarchical level the correct model of computation is chosen. The controller hierarchy and the models of computation used at each level are visualized in figure 4.2.

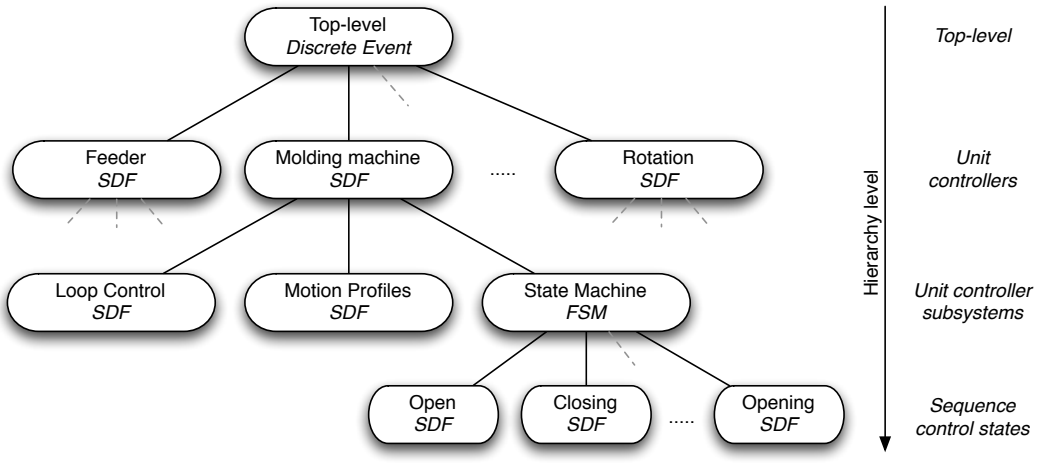


FIGURE 4.2 - Controller hierarchy and used models of computation (for the molding machine)

4.3.2 Controller top-level

The Ptolemy II implementation of the top-level controller is depicted in figure 4.3. Communication among the six unit controllers is directed by a Discrete Event (DE) director. When a unit controller finishes its task, it sends a message (an event) to the next unit controller indicating

that it can start its task. This is a more robust structure than the one used in chapter 3 because the system will now only run if all six unit controllers operate as expected. In the configuration used in chapter 3, the sensors are not used for synchronization, only for verification of correct behavior by the individual unit controllers. The Starter block is used to initiate the motion of

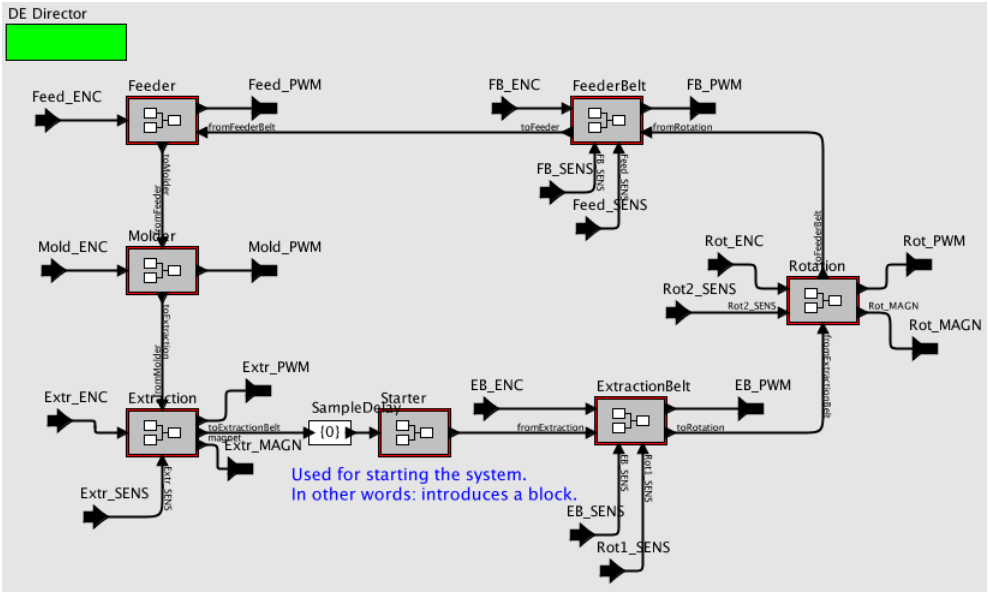


FIGURE 4.3 - Controller structure

the system. It models the signal that the first aluminum block inserted into the system would generate. The SampleDelay block provides a starting point for calculation of the execution schedule by the DE director, thus avoiding deadlock.

4.3.3 Unit controller

The controller for a single unit (the unit controller), as proposed by (van Zuijlen, 2008) (see figure 2.6(b)) is implemented in the Ptolemy II model as shown in figure 4.4. The state ma-

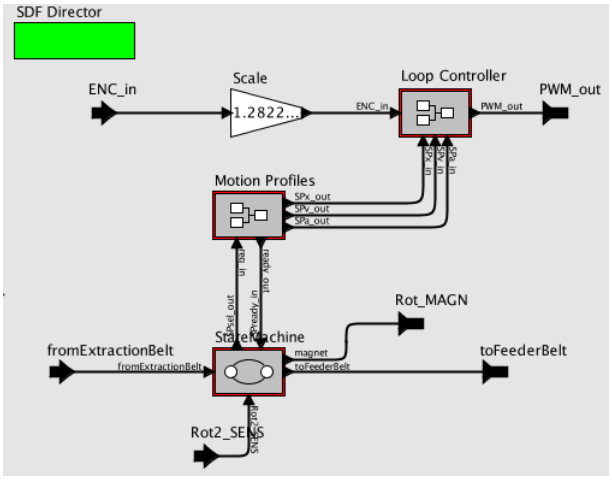


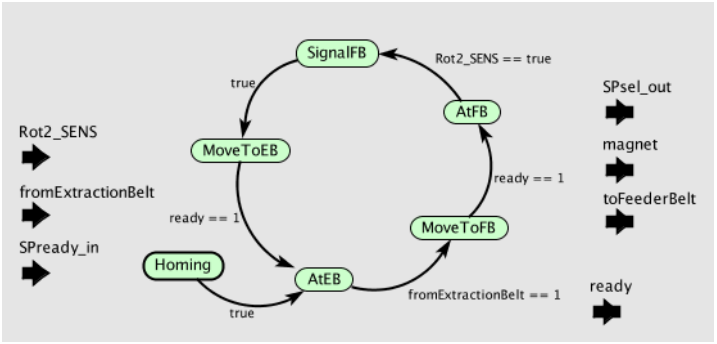
FIGURE 4.4 - Unit controller structure

chine (shown at the bottom) determines which motion profile to execute, based on the events received from the controller of the previous unit and sensor statuses. The number of the selected motion profile is sent to the Motion Profiles composite actor (shown in the middle) which generates appropriate signals for position, velocity and acceleration setpoints. The Loop

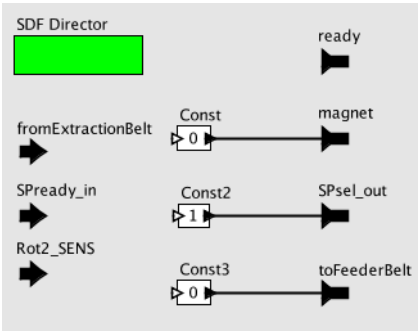
Controller then generates the PWM value which ensures a minimal tracking error, using the setpoint values and the encoder value received from the plant. As soon as the selected motion profile is finished, the State Machine is signaled. The State Machine can then perform the appropriate actions, such as selecting the next motion profile and sending an event to the next unit controller.

4.3.4 State machine

Sequence control for a single unit is modeled in the form of a state machine. The state machine for the rotation robot is shown in figure 4.5(a). State transitions are triggered either by an event from the previous unit controller, a sensor signal or a message indicating that a motion profile is finished. Each state implements a *refinement*, which determines the input/output relation. The refinement for the 'AtFB' state (an SDF model) is shown in figure 4.5(b).



(a) State machine for sequence control of one unit

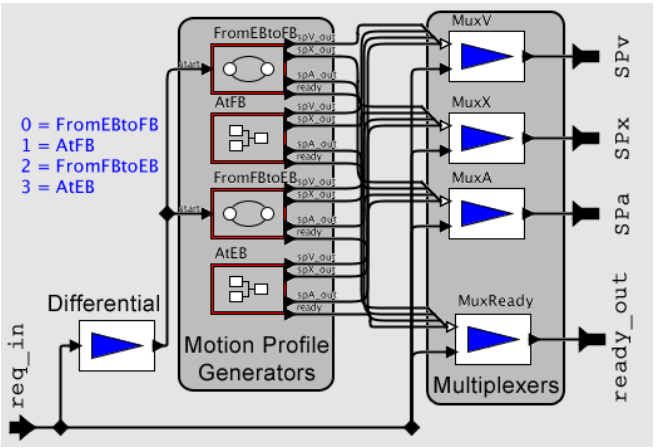


(b) State refinement

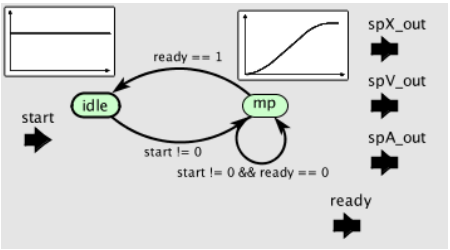
FIGURE 4.5 - State machine and refinement for one state

4.3.5 Motion profiles

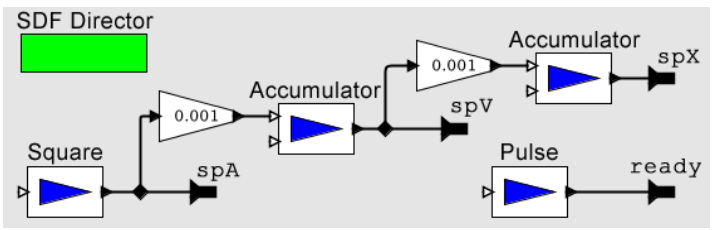
The implementation of the Motion Profiles block is depicted in figure 4.6.



(a) Motion profile generator structure



(b) State machine for 'FromEBtoFB'



(c) State refinement for 'mp' state

FIGURE 4.6 - Motion profile generator

The `req_in` port (bottom left in figure 4.6(a)) receives the number of the desired motion profile (0 through 3 in this case). The selection is made by four multiplexers at the right. Whenever the `req_in` signal changes (detected by the Differential actor) the motion profiles are reset. Actual motion profile generation is performed by the four composite actors in the center. The transitional motion profiles use a state machine with two states for this, see figure 4.6(b). In the 'idle' state a constant setpoint value is used. In the 'mp' state a constant acceleration profile is used which is integrated twice (see figure 4.6(c) and 4.7). The signal indicating that a motion profile is finished is sent through the `ready_out` port.

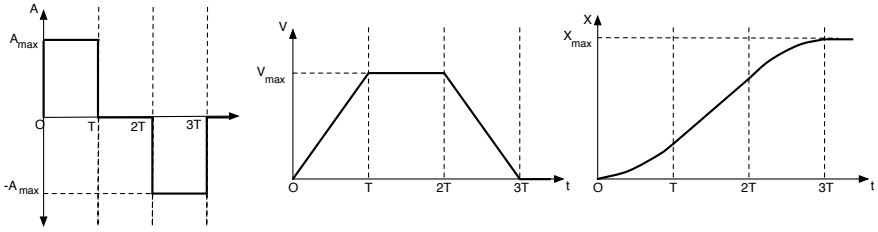


FIGURE 4.7 - Motion profiles

4.3.6 Loop controller

The loop controller here is exactly the same as the one used in chapter 3. As the functionality of the PID controller block proved to be sufficient, there is no need to change it for this model.

4.4 Plant model

For the modeling of the plant dynamics the same method as in the feasibility analysis (refer to section 3.1) is used. For each unit a model linearization in 20-Sim is performed, resulting in six transfer functions. Each transfer function describes the relation between a PWM signal input and its corresponding encoder output value. The plant models are not coupled, therefore six uncoupled actors are used, see figure 4.8.

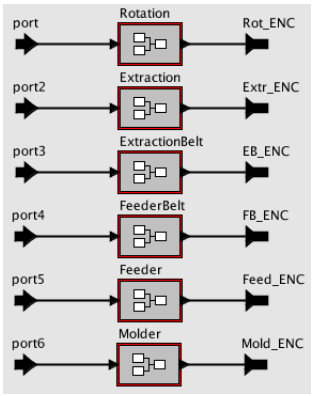


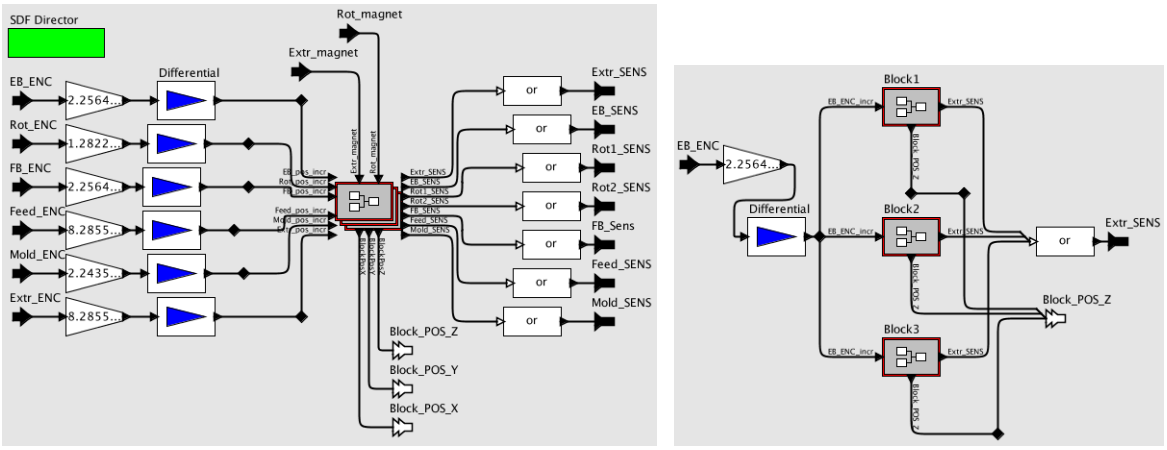
FIGURE 4.8 - Plant models for all six units

4.5 Block model

The aluminum blocks in the production cell setup are modeled by the composite actor shown in the middle left of figure 4.1. The implementation of the aluminum block model is shown in figure 4.9.

As mentioned in section 4.1 only the kinematic behavior of the blocks is taken into account. Dynamic behavior is ignored, as it is mostly irrelevant for the design of the embedded control software.

The actor in the middle of figure 4.9(a) is a MultiInstanceCompositeActor. It is an example of the object-oriented features available in Ptolemy II modeling. The model contained in the



(a) Structure as drawn in Ptolemy II (b) Automatic expansion at runtime

FIGURE 4.9 - Structure of the aluminum block model

MultiInstanceCompositeActor is instantiated nBlocks (a parameter at top-level) times when running the model, as illustrated in figure 4.9(b) (for three ports). This means that the actual model for the aluminum blocks is designed only once. By setting the value of the nBlocks parameter, the actual number of blocks used in the simulation is determined. In the current version of Ptolemy II (6.0.2) the MultiInstanceCompositeActor does not work properly, which causes unexpected behavior if the model is run more than once. This bug has been fixed in Ptolemy II 7.0 beta.

The block model uses the angular velocity (hence the differential actors) to update its own position, measured along the trajectory traveled by each block. This position information is used to trigger the sensors in the setup. A boolean `OR` is used to trigger the sensors for all blocks. The same position information is converted into x, y and z coordinates that are used in the 3D graphical representation of the block.

4.6 Conclusions

A complete model of the production cell setup, including both the plant dynamics and the discrete time controller, has been designed using Ptolemy II as a design environment. Aspects relevant to the design of the embedded control software have been incorporated into the model, while less relevant aspects (such as the dynamics of the aluminum blocks) have been disregarded. In future work, the model can be refined to include these aspects to predict effects like the bouncing of an aluminum block on a belt or against the molder door, which can affect functional behavior in extreme cases. A safety layer can be implemented by checking whether hardware control signals (PWM and encoder signals) stay within limits for a certain state of the sequence controller. A finite state machine can be used for this.

Strong use was made of the Ptolemy method: the model contains many hierarchical levels, each using the proper model of computation (realized by means of the directors). The model allows for full functional testing. Correct behavior can be verified by using the 3D graphics animation and classical graphing techniques. Changes to the controller parameters, PID settings, sequence control, etc. can be made in a straightforward manner because of the clear controller structure. Also, each loop controller can run at a different frequency by changing the appropriate parameter in the AD block in figure 4.1.

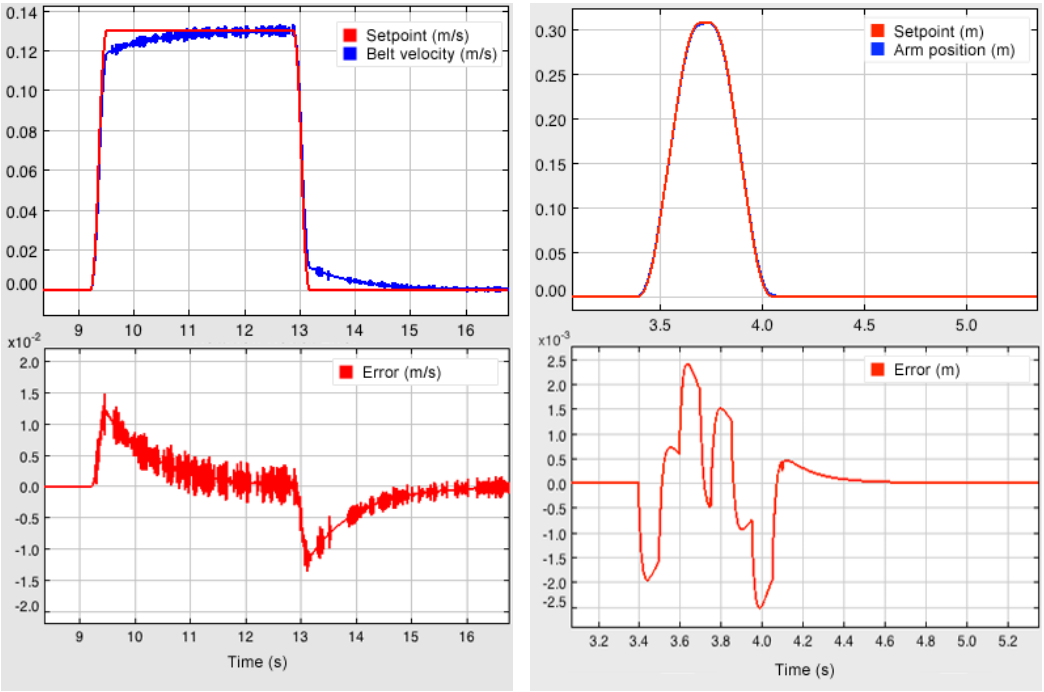
The next chapter presents more on the results obtained using this model, with regard to controller design and realization. Chapter 6 discusses the Ptolemy approach used in the design of this model.

5 Results

This chapter presents the results obtained from simulations of the Production Cell model in Ptolemy II. After loop controller verification (section 5.1), functional verification is performed using a 3D animation (section 5.2). Finally, deployment of the generated code to the target platform is discussed in section 5.3.

5.1 Loop controller verification

Simulations were performed to verify loop controller behavior. Simulation results for two units in the system are presented in figure 5.1. Simulation results for the other four units are in appendix B.



(a) Extraction belt (b) Rotation robot

FIGURE 5.1 - Controller verification results

According to Maljaars (2006), the steady-state error of the position controlled units (rotation robot, feeder, molder and extraction robot) should not exceed 0.5 mm = 5×10^{-4} m. These demands are satisfied, although the maximum error is exceeded when moving from one position to the other. For the velocity controlled units (extraction belt and feeder belt), no error constraints are mentioned in (Maljaars, 2006). A maximum error of 1% of the maximum belt velocity is chosen, which corresponds to 0.13×10^{-2} m/s. This maximum error is not exceeded. However, there is a lot of noise on the belt velocity signal. This is due to the differentiation of the encoder signals, which is used in velocity control. To reduce the amount of noise, a state variable filter can be used to obtain the velocity signal, instead of a true differentiator. Another option is to remove the differentiator on the encoder signal, use an extra integrator on the velocity setpoint and then use position control.

5.2 Functional verification

One of the main goals of developing an integrated model of the Production Cell setup is to allow for integrated functional verification. This is facilitated by a 3D graphical representation of the complete system, as shown in figure 5.2. The 3D animation provides an intuitive way of verifying correct behavior. It includes a basic representations of all six actuated units, the

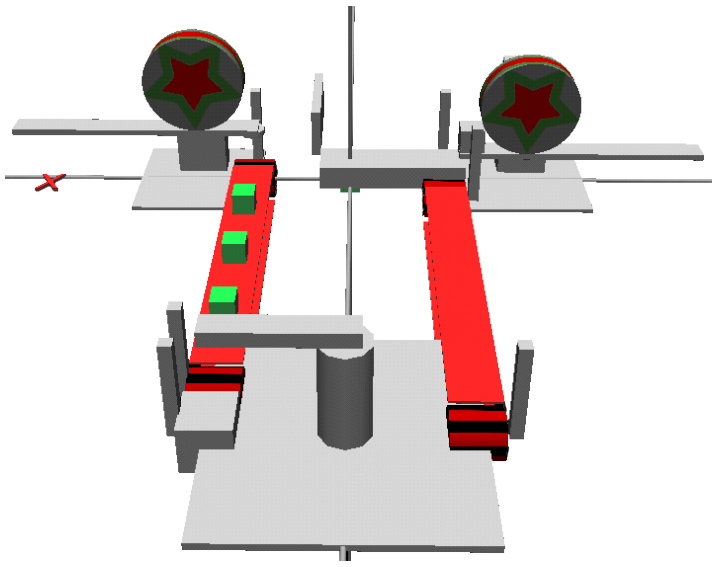


FIGURE 5.2 - Ptolemy II 3D animation of the Production Cell model

sensors in the system and the aluminum blocks. The 3D animation was used extensively while designing the sequence controllers, resulting in a design showing functionally correct behavior.

5.3 Realization

The final step in the embedded control software design trajectory is the realization of the system. In the Ptolemy approach, this step is implemented by means of automatic code generation. Currently, the Discrete Event (DE) domain, used at the controller top-level (figure 4.3), is not supported in code-generation. Implementing code generation for the DE domain is not feasible in the timeframe available for this project.

In order to allow code generation the model is adapted to eliminate the Discrete Event model of computation. The DE director at the controller top-level is replaced by a Synchronous Dataflow (SDF) director. This implies that communication between controllers will now have to take place at each timestep. Therefore, the unit controllers are adapted to produce a token at each timestep, according to the scheme of table 5.1.

TABLE 5.1 - Tokens sent at controller top-level in DE and SDF domains

Unit controller finished	**Discrete Event**	**Synchronous Dataflow**
Task not finished	none	0
Task finished	arbitrary value	1

The controller will now put a higher computational load on the target PC/104 computer, because communication takes place at each timestep, instead of only at those timesteps where communication is required. Furthermore, the current code generator does not allow multiple timed loops. Therefore, the entire controller will run in a single timed loop at 1.0 kHz on the target PC/104 computer.

Similar to the steps taken in section 3.4, a separate model is created which includes only the controller model and LinkDrivers for I/O. Code is generated, compiled and run on the target PC/104 computer. Qualitative analysis (by visual inspection) on the running Production Cell system shows that it behaves as expected.

6 Discussion

This chapter discusses the Ptolemy method, based on the results obtained from the practical case study using the Production Cell setup. Evaluation of the integrated approach leads to an overview of other methodologies and their tools. Next, some suggestions on improving the CE-toolchain are given and finally conclusions are drawn.

6.1 Integrated approach

In this project an integrated approach was used to create embedded control software for the Production Cell setup. A single modeling framework (Ptolemy II) was used for all aspects of system modeling and embedded control software design. Results show that the problems identified in section 1.1 can be overcome by using an integrated approach:

- Integrated testing: because all relevant aspects of the system have been modeled in a single modeling framework, integrated verification proved to be a relatively easy task. Control law designs were verified, followed by the verification of correct functional behavior (sequence control).
- Iterative design: as all steps of the embedded control software design process are supported by one software package, no manual model transformations need to be performed. This reduces the probability of introducing errors, as well as the time required for each iteration.

Code can be generated directly from the integrated model. This code can be compiled and run on the actual target platform, without any additional intermediate steps.

However, using an integrated approach does introduce a problem. All design steps are performed using a single design tool, which means that this tool should support all involved disciplines. This requires a very generic tool, which (by definition) means that domain-specific requirements can not always be met. The modeling of the Production Cell system in this project illustrates this. A generic tool (Ptolemy II) was used, in which system dynamics modeling is very limited and no specific features for control law design are implemented (e.g. a pole-zero analysis wizard). Domain-specific tools can accommodate these needs much better.

To provide some more context, the next section gives an overview of other commonly used methodologies and tools for embedded control software development.

6.2 Other methods and tools

Focusing on embedded control software development for mechatronic systems, several important points can be identified:

- Integration: the gap between each phase in the development process should be as small as possible, allowing for fast design iterations. Also, integrated verification should be possible;
- Trajectory coverage: a methodology and its supporting tools should cover all phases in the design trajectory of figure 1.2. Important practical issues are modeling (for system dynamics, control law design and software design), simulation and code generation;
- Economics: the cost of using a certain toolchain, its maturity and the time-to-market that can be realized are important economic factors in choosing a certain methodology / toolchain.

With these points in mind, a comparison of the Ptolemy method with four other methods (Co-simulation, CE-method, Matlab/Simulink and POOSL, refer to section 1.3.1) is made. This comparison is split into three parts:

- Methodology properties: the ways of thinking and working in each methodology, as well as typical application areas;
- Methodology implementations: the implementation of each methodology is described, using the Boderc methodology description, which will be explained in section 6.2.2 (Heemels and Muller, 2006);
- Operational features: some important issues in the tooling that supports each methodology.

6.2.1 Methodology properties

These properties describe the methodologies at the highest abstraction level. The first issue here is the *Way of thinking*, i.e. the basic idea of a methodology. For Ptolemy II, this is the concept of an integrated approach; use a generic modeling language to model all relevant aspects of a system in one integrated model, enabling integrated verification. Choosing the right model of computation is very important. On the other hand, the co-simulation approach tries to use the 'best' tool for modeling each part of the system and couples these tools to perform integrated verification. The CE-method uses specific tools for each design step in a process of stepwise refinement. At each transition to a next design step, automatic model transformations are used to transform the model to a new view. In Matlab/Simulink rapid prototyping is used. The POOSL method focuses on concurrent engineering (software is developed in parallel with the design in other disciplines), local refinement (each design step should focus on partial information, which reduces complexity) and predictable refinement (properties of parts should be preserved when integration is performed) with a focus on scheduling and timing.

The second issue concerns the *Way of working*, i.e. the workflow for creating the system design. Designers using the CE-method tend to use a bottom-up approach. An example is the Production Cell setup, for which a motor model is created first, then a plant model for one unit, next a loop controller for one unit, then sequence control for one unit and finally the synchronization and concurrency between all units is taken into account. In the POOSL method this order is reversed: a C-model describes the concurrency in the system, the M-model is used to describe the interactions between high-level and low-level control and finally the R-model incorporates timing and continuous time behavior. Ptolemy and co-simulation combine these two methods. In Ptolemy, the low-level behavior is analyzed first (e.g. motor control). Next, the top-level design is created and refined until all relevant aspects have been modeled. A single design tool (Ptolemy II) is used for this. In co-simulation one starts with the top-level model design. Implementation is done bottom-up, starting in seperate design tools and finally combining these submodels in the top-level model in the co-simulation package. Matlab/Simulink uses a simulation based approach, making extensive use of Hardware-, Software- and Processor-in-the-loop simulations.

Finally, each method targets specific domains. Although it is possible to use all methods for almost any design application, specific methods address specific domains (e.g. it is possible to create a wireless network model in 20-Sim, but Ptolemy II would be a better choice).

Table 6.1 summarizes the results from this section.

TABLE 6.1 - Methodology properties

	Ptolemy	**Co-simulation**	**CE-method**	**Matlab/Simulink**	**POOSL**
Way of thinking	Integrated approach. Choose right Model of Computation.	Use best tool for each view. Use co-simulation only when loop among submodels exists (feedback).	Specific tools for each design step. Use stepwise refinement.	Rapid prototyping.	Concurrent engineering, local refinement, predictable refinement. Focus on scheduling and timing.
Way of working	'Middle out': verify low-level functionality, then top-level design. All in one software package.	Top-down design: top-level model and interfaces. Bottom-up implementation: create submodels in external tools, connect in co-simulation package.	Bottom-up. Model based: plant dynamics (20-Sim), controller (20-Sim), ECS design (gCSP), realization (CTC++).	Heavy use of Hardware-, Processor- and Software-In-the-Loop simulations.	Top-down. Software (model) approach using C, M and R models.
Typical domains	Real-time embedded software design and analysis.	Hardware/software co-design, mechatronics.	Mechatronics.	Signal processing, controller design.	Embedded software design and analysis. No modeling of plant dynamics and controller design.

6.2.2 Methodology implementations

From the basic idea of each methodology follows an implementation. This implementation is described using four terms, also used in the Boderc project (Heemels and Muller, 2006):

- *Formalisms*: languages / syntax used for system modeling. Formalisms exist for modeling behavior, but also to formalize system requirements. Instances of formalisms are called *Models*. Examples of formalisms are differential equations, finite state machines, temporal logic and queuing formalisms;
- *Techniques*: used to retrieve information from models or to transform models. Examples of analysis techniques are model checking, performance analysis and program analysis techniques. Examples of transformation techniques are high-level synthesis and software compilation;
- *Methods*: in other words: a reasoning framework. Provides guidelines and can be seen as a 'recipe book' how and in which order to apply certain Formalisms, Techniques, Submethods and Tools to solve the design problem at hand;
- *Tools*: Software Tools support the efficient application of Formalisms, Techniques and Submethods.

Table 6.2 summarizes the results of the comparison on methodology implementations. All methodologies use graphical formalisms to describe a model at top-level. In Ptolemy the semantics ('meaning') of the model depends on the model of computation (i.e. formalism) used. In co-simulation, each simulation package uses its own formalism, coupled via a co-simulation package. The CE-method uses various formalisms for plant modeling, including Ideal Physical Modeling and bondgraph theory. CSP is used for embedded software design. Matlab/Simulink is completely actor-oriented. POOSL is a timed and probabilistic extension of CCS (Calculus of Communicating Systems) which uses a diagram of processes and channels at top-level and POOSL code at lower levels. Each formalism is supported by an underlying language. Ptolemy uses Java, various languages are used in co-simulation, the CE-method uses port-based SIDOPS code in conjunction with C++ and CSP, Matlab/Simulink uses Matlab code and in the POOSL method POOSL code is used.

All methodologies use executable models to allow simulation to analyze the model. In Ptolemy, co-simulation, the CE-method and Matlab/Simulink the traditional method of 2D graphs is used. Additionally, Ptolemy and the CE-method (more specifically, 20-Sim) support 3D animations. Further analysis methods include wizards (e.g. pole-zero analysis in 20-Sim) and diagram animation (gCSP and POOSL). Supported transformation techniques include code generation (Ptolemy, CE-method, Matlab/Simulink and POOSL) and automated compilation (CE-method).

The methods and tools used in these five methodologies can be found in table 6.2.

TABLE 6.2 - Methodology implementations

	Ptolemy	**Co-simulation**	**CE-method**	**Matlab/Simulink**	**POOSL**
Formalisms	Actor-oriented models, semantics of actor based on Model of Computation. Language: Java.	Anything, as long as external tooling uses executable models and supports external communication. Synchronization via co-simulation tool. Languages: various.	Actor-oriented block-diagrams, IPM, bondgraph, CSP. Languages: port-based SIDOPS, C++, CSP.	Actor-oriented block-diagrams for signal flow. Language: Matlab code.	Timed and probabilistic extension of CCS; Top-level: diagram of processes and channels; lower levels: POOSL code. Language: POOSL.
Techniques	Simulation (graphs and 3D animation), code generation (C, VHDL).	Co-simulation tool to analyze inter-simulator signals. External simulators and analysis tools to check behavior of specific model parts.	20-Sim: Simulation, wizards, code-generation (C++, Handel-C); gCSP and CTC++: animation and tracing, code generation; 4C: compilation.	Simulation, code generation (C, C++, VHDL) through real-time toolbox.	Simulation through diagram animation, interaction diagram (sequence diagram).
Methods	Hierarchical block-diagrams, finite state machines.	Co-simulation connection diagram and views used in external tools.	20-Sim: hierarchical block-diagrams (signal flow), IPM, bondgraph; gCSP: GML; CTC++: C++ code.	Hierarchical block-diagrams.	Top-level: block-diagram; lower levels: POOSL code; C, M and R models.
Tools	Ptolemy II.	CosiMate and supported external simulators.	20-Sim, gCSP, CTC++, 4C, RTAI.	Matlab/Simulink, proprietary RTOS.	SHESim, Rotalumis, 20-Sim.

6.2.3 Operational features

A key factor for success of a methodology is the usability of the tooling that supports the methodology. This section identifies some important operational features of the tools supporting each methodology. Table 6.3 shows the results, which are explained in the remainder of this section.

Model structuring is important. A tool should allow the creation of a clear and well structured model. Essential are the use of hierarchical modeling and easy drawing and routing of signals.

Next, *object orientation*, known from object oriented programming, can be useful in modeling. Ptolemy II allows the creation of model classes. These model classes can be instantiated as many times as desired, or they can be used as a base class so that model classes can be derived from them. POOSL uses object oriented techniques through its Data, Process and Cluster classes. No object orientation is used in the other tools, although 20-Sim allows parameter coupling between submodel copies, mimicking object-oriented behavior.

Another key feature in a modeling tool is *extendability*. A user should be able to create custom submodels or even additions to the tool, to accommodate modeling needs not natively covered by it. All tools allow some form of extensions, but Ptolemy II takes this the furthest by allowing custom Java classes to be written that can extend both the actor library as well as the modeling environment itself. However, this is not a user friendly method, as knowledge of the Ptolemy II architecture and the Java programming language is required.

Code generation is an essential feature in all five methodologies and thus is supported in all tools. However, different mechanisms are used: Ptolemy II uses a helper based mechanism, template based code generation is used in the CE toolchain, Simulink uses the Real-Time Workshop (helper based) and POOSL generates code based on Process Execution Trees. All these mechanisms produce executable code. However, the Ptolemy II code generation framework is still experimental and therefore only usable in certain situations (only static scheduling, not all actors have an associated helper).

Model reuse should allow a designer to reuse previous work. Next to model libraries, present in all tools, Ptolemy II adds the possibility of writing domain and data polymorphic actors. This ensures that an actor can be used in multiple domains, using multiple data types.

The use of *signal busses* allows a designer to group similar signals into a single bus. This can greatly enhance the readability of a model. Only Simulink natively supports signal busses in a user friendly way. In co-simulation and POOSL signal busses can be created by using a 'vector' data type. Ptolemy II and the CE-toolchain support signal busses, but the order of the signals inside the bus depends on the order of connecting these signals, which can become confusing.

Next, the *mathematical features* of the tools are examined, in particular with respect to matrix calculations and complex numbers, as these are widely used in embedded control applications.

Design speed, including simulation time and time required for model construction, is another key factor. Graphical modeling allows for fast model construction, while code-based model design takes more time. For simulations, Ptolemy II is relatively slow due to its Java background. Co-simulation introduces only a small overhead on top of the used simulators, but can cause high computational load and thus long simulations due to running multiple simulators on a single computer. Simulations in the CE-toolchain (20-Sim) are very fast due to heavy optimization. Simulink is in between Ptolemy and the CE-toolchain; it has a Java back-end but it has also been heavily optimized. Finally, simulations in POOSL are also relatively slow, but can be sped up (up to 100×) by using Rotalumis, at the cost of possibilities for analysis.

As the variety of target *platforms* grows, it becomes more important for tools to run on various platforms. In this sense choosing Java, though slow, is a good choice (Ptolemy II, gCSP). Most tools are available for various platforms. Currently, only CosiMate (co-simulation) and 20-Sim (CE-toolchain) run only on Windows, although Linux versions are under development.

TABLE 6.3 - Operational features

	Ptolemy II	**Co-simulation**	**CE-toolchain**	**Simulink**	**POOSL**
Model structuring	Strongly hierarchical, poor channel routing	Top-level: co-simulation connection diagram. Lower levels: depends on specific tooling.	20-Sim: allows hierarchy. Easy signal routing. gCSP: hierarchy in processes.	Hierarchical modeling supported.	Top-level: block diagram showing processes and data channels. Lower levels: POOSL code.
Object orientation	Supports actor classes which are instantiated or subclassed.	Not applicable.	No object oriented techniques in modeling. Parameter coupling in submodel copies.	No object oriented techniques in modeling.	Uses Data, Process and Cluster classes.
Extendability	Open source, extensions (both extra actors and tool extensions) through extra Java classes.	Support additional external simulators by writing new interfaces to co-simulation bus.	20-Sim: custom submodels, DLLs; gCSP: code blocks; CTC++: not extendable.	Custom submodels, Matlab functions, external DLLs.	Extendable through custom data, process and cluster classes.
Code generation	Helper based, for many languages. Currently C and VHDL.	Code generation through external design packages.	20-Sim: template based C code; gCSP: C++ for CTC++, Handel-C.	Generate C, C++ or VHDL code using Real-Time Workshop.	Rotalumis: Process Execution Trees in C++.
Model reuse	Domain and data polymorphic actors.	Reuse existing models in supported tools.	Reuse of submodels.	Model libraries.	Reuse of classes.
Signal busses	Supported, but can result in loss of overview.	Supported through use of vectors.	Supported in 20-Sim, not in gCSP.	Supported.	Supported through use of custom datatypes.
Mathematical features	Ptolemy expression language supports complex numbers and matrix calculations.	Only vectors (no matrices), no native complex numbers.	Full matrix support. No complex numbers.	Math support through Matlab engine, includes matrices and complex numbers.	Data classes implement data types; supports matrices, custom datatypes possible.
Design speed	Relatively slow simulations (Java).	Small overhead on top of used simulators.	Fast model construction and simulation.	Fast model construction, good simulation speed.	SHESim: slow simulation and model construction; Rotalumis: 100× faster simulation.
Platforms	Runs on all platforms for which a JVM is available.	Windows only.	20-Sim: Windows; gCSP: all platforms for which a JVM is available; CTC++: Linux and Windows; 4C: Windows only.	Windows, Linux, Mac OS X, Solaris.	SHESim: Windows, Linux, Power Mac, HP UX, Sun Solaris; Rotalumis: Windows, Linux; 20-Sim: Windows.

6.2.4 Conclusions

From the previous section it has become clear that a wide range of methodologies for embedded control software development exist, each with its own characteristics. Which methodology and tool to choose depends heavily on the application at hand. For example, if the system under development involves a lot of signal processing, Matlab/Simulink seems a good choice. If a lot of mechatronics is involved, Matlab/Simulink is less suited and the CE-method is a better choice because of its more extensive dynamics modeling features (e.g. bond graphs). Also, methodology choice may depend heavily on user preference and experience with certain tooling. If a designer is used to using the CE-method with 20-Sim, gCSP and CTC++, it will require significant effort to switch to the Ptolemy method using Ptolemy II. In multi-disciplinary design projects, where each discipline is supported by its own tool(s), this will be an even greater issue. In such a situation co-simulation becomes an attractive option.

With regard to integration, Ptolemy II, co-simulation and Matlab/Simulink show the best results. Ptolemy II and Matlab/Simulink allow the designer to perform the entire embedded control software trajectory in a single tool (although using Matlab/Simulink might require purchasing a lot of toolboxes). In co-simulation different tools are integrated by means of a co-simulation package. The CE-toolchain and Matlab/Simulink provide the best coverage of the complete design trajectory. In the CE-toolchain the focus in modeling is towards mechatronic systems, while in Matlab/Simulink signal processing is more important. Ptolemy II also covers the entire design trajectory, but lacks valuable features for system dynamics modeling and control law design. POOSL focuses on embedded software analysis and design and does not include features for system dynamics modeling and control law design. Therefore, it is not complete for software design for mechatronic systems. All methods use executable models for simulation and allow automatic code generation. When taking economic factors into account, the CE-toolchain seems an attractive option as it is low-cost, relatively mature (especially when compared to Ptolemy and POOSL) and results in a short time-to-market. Matlab/Simulink is the most mature method, but it is also expensive. Ptolemy II and POOSL are still very experimental, which limits their use in an industrial environment. These methods should be viewed as research projects; experiences gained there can be used to improve other embedded control software development methods. Success of the co-simulation approach relies heavily on the external design tools used and their interface to the co-simulation environment.

In conclusion, it can be said that there is no single methodology or tool that can be considered superior to other methodologies and tools. Methodology and tool selection depends on many factors including, among others, application area, user preference, user experience, maturity and cost. However, when focusing on embedded control software development for mechatronic systems, the CE-method with its corresponding toolchain is a good choice. It is a relatively low-cost solution, it covers all steps in the design process and it allows a short time-to-market. However, the CE-toolchain can still be improved, as will be discussed in the next section. Integration between the tools in the CE-toolchain remains an issue, for which co-simulation can provide a solution. Ultimately, tool integration should be transparent to the user. This can be achieved by creating well-defined interfaces between tools and a graphical user interface that combines these seperate tools. Well-defined tool interfaces also allow for in-the-loop simulations, which can reduce the gap between model and realization.

6.3 Improving the CE-toolchain

In theory, Ptolemy II includes the essential features needed for successful embedded control software development. However, as discussed in the previous section, using a generic tool causes practical problems. A better design can be created when using tooling specific to the design task at hand. However, Ptolemy II techniques can be used to improve these specific tools. For the CE-toolchain, the following points are identified:

- Multiple Models of Computation: currently, 20-Sim supports continuous time and discrete

time modeling. However, for efficient modeling of dynamics and controller design, more models of computation are desirable, starting with the Finite State Machine (FSM) and Discrete Event (DE) domains. In order to ensure the formal correctness of the resulting heterogeneous models, the Ptolemy method for combining multiple models of computation in a single model can be used (Eker et al., 2003). Models of computation that require specialistic tools (e.g. gCSP for CSP or Modelsim for VHDL) should not be integrated. Here, co-simulation is a better choice.

- Code generation: the code generation framework in Ptolemy II is very generic and flexible. Its helper-based mechanism allows easy implementation of code generation for new target languages or language variants. For example, integer-based code generation can be implemented by creating an integer-based director and its corresponding code generation helper. Currently, 20-Sim supports only floating point code generation. This is a limitation if, for example, an FPGA-based controller is designed, as an FPGA does not support floating point calculations natively. Using the Ptolemy II code generation framework in 20-Sim will solve this issue and allow for future extensions.

  The code generation system in gCSP can also be improved by using Ptolemy II techniques. A gCSP model consists of an hierarchical ordering of processes, similar to the hierarchical ordering of actors in a Ptolemy II model. For gCSP models it is also desirable to have the capability of generating code for multiple target languages (C++ for use with the CTC++ library, CSPm for formal checking, Handel-C for FPGAs, etc.). These two properties lead to the conclusion that the Ptolemy II code generation framework can be used in gCSP, and would improve its code generation capabilities.

  The current code generation framework in Ptolemy II supports only statically scheduled domains (execution schedule determined at time of code generation). For successful code generation from all heterogeneous models the code generation framework should be extended to include non-statically scheduled domains (domains for which the execution schedule is determined at runtime, like CSP and DE). Helpers for all directors, as well as for remaining actors, should be developed. This is not a straightforward task and will require extensive testing, especially if real-time behavior is desired.
- Object orientation: Ptolemy II uses object oriented techniques in modeling. Actors can be converted to a model class. These model classes can then be instantiated as many times as desired or used as a base class from which other model classes are derived. This allows a designer to create well-structured models in which the amount of repetitive work is minimized. Generic model classes can also be reused easily. Incorporating object oriented modeling techniques in the CE-toolchain will allow for a more efficient workflow.
- Extendability: Ptolemy II can easily be extended with additional functionality, because of its object oriented software design and open source nature. Users can extend the framework to allow modeling of systems that are not supported natively, while still gaining from the benefits offered by a general modeling framework. An example of an extension to Ptolemy II for modeling sensor networks is *VisualSense* (Baldwin et al., 2004). The CE-toolchain, especially 20-Sim, can also benefit from supporting these kind of extensions, as it generates a larger userbase. 20-Sim currently supports extensions through the use of external DLLs. This feature should be extended and the documentation should be improved, so that it is easier to use. Furthermore, the sharing of extensions for both Ptolemy II and 20-Sim should be promoted by creating an online repository where users can share their work.

6.4 Conclusions

An integrated approach, tested by using Ptolemy II, solves the problems of integrated testing and iterative design. However, a generic tool can never fully support all disciplines, which can be a problem for specialized engineers. While there are various other methodologies and accompanying tool chains available, none of them can be considered superior. Optimal method-

ology and tool choice depends heavily on many factors, including the application domain, user preference, user experience, maturity and cost.

A compromise between the integrated approach and the approach using a seperate tool for each design task should be found: integrate domains used in closely related design tasks in a single tool and use co-simulation when simulation in conjunction with other domains is required. When focusing on embedded control software development for mechatronics, the CE-method, together with co-simulation, is a good choice. Ultimately, the integration of the tools in the CE-toolchain should be transparent to the user. This requires well-defined tool interfaces, which also allow for in-the-loop simulations to further decrease the gap between model and realization.

The CE-toolchain can be improved by using Ptolemy II techniques. More models of computation should be integrated into 20-Sim (especially FSM and DE). The Ptolemy II mechanism for combining multiple models of computation can be used to ensure formal correctness. The generic and flexible Ptolemy II code generation framework can be used as a basis for the 20-Sim and gCSP code generation systems. Object oriented techniques can be incorporated into the CE-toolchain to facilitate more efficient modeling. Finally, the Ptolemy philosophy of allowing extensions made by users should be adopted by the CE-toolchain. All this will increase the range of application areas for the CE-toolchain.

7 Conclusions and recommendations

7.1 Conclusions

Embedded systems are inherently heterogeneous. Their design involves various disciplines, e.g. mechanics design, dynamics modeling, control engineering and software design. Increasing complexity of these systems requires design tools that support these heterogeneous systems. The main issues here are integrated verification and enabling short design iterations without manual model transformations. Both of these issues are addressed by the Ptolemy approach.

In this project Ptolemy II was used to create embedded control software for the Production Cell setup. Previous work on modeling system dynamics (Maljaars, 2006) and controller structure design (van Zuijlen, 2008) was reused. The Ptolemy II model of the Production Cell setup allows for integrated functional verification. Short design iterations can be made, as the use of a single design tool requires no manual model transformations. The automatically generated C code from the model can be compiled and run on the target PC/104 computer, resulting in a correctly working Production Cell setup.

Although it is shown that correctly working embedded control software can be developed by using a single design tool, this approach does show practical problems. A general design tool can not satisfy all specialistic needs for each involved discipline. For example, while designing the control software for the Production Cell setup, this became clear by the limited possibilities for system dynamics modeling and control law design. For this reason, a compromise between using a single tool for the entire design process and using separate tools for each design task should be found. Integrating closely related models of computation for a design step in a single tool (e.g. the continuous time, discrete time, finite state machine and discrete event models of computation for control law design in 20-Sim) and using co-simulation for integrated verification between tools is a good compromise. Ultimately, tool integration should be transparent to the model designer. This requires well-defined interfaces between these tools and a single graphical user interface to combine the separate tools. Well-defined tool interfaces also facilitate in-the-loop simulations, further decreasing the gap between model and realization.

No methodology or tool for embedded control software development can be considered superior. Methodology and tool selection depends on many factors including, among others, application area, user preference, user experience, maturity and cost. When focusing on mechatronics the CE-method with its toolchain is a good choice, although it still needs improvement.

The CE-toolchain can be improved by using Ptolemy II key features. Extending 20-Sim with support for the Finite State Machine and Discrete Event models of computation allows the modeling of a wider range of systems. Ptolemy techniques can be used to ensure formal correctness of the resulting heterogeneous models. Both 20-Sim and gCSP can support a wide range of target languages and language variants by incorporating the Ptolemy II code generation framework. Object orientation principles and user extensions can further improve the usability of the CE-toolchain.

7.2 Recommendations

At the end of this project there still remains work to be done.

First of all, the Ptolemy II Production Cell model should be extended to include the aspects ignored in this project (e.g. dynamics of the aluminum blocks, non-linearity's in the models of the plant dynamics) Also, safety features, error handling and a user interface should be included. This will result in the first truly complete model of the Production Cell system, allowing for full integrated verification.

The Ptolemy II code generation framework should be extended to allow code generation for non-statically scheduled domains. Helpers should be defined for all actors and code generation for a specific submodel should be made possible, eliminating the need for a separate model for code generation. These extensions will create the possibility to generate code directly from the model used for verification, making the step from model to realization as small as possible.

Currently, physical I/O in Ptolemy II is implemented through LinkDriver actors. This makes the model dependent on the target platform, which is undesired. This hardware dependency can be removed by using a target connector, for example the 4C toolchain.

Finally, the suggested improvements to the CE-toolchain should be implemented. This includes implementing the Finite State Machine and Discrete Event models of computation in 20-Sim, using a Ptolemy II style code generation framework in 20-Sim and gCSP, using object oriented techniques in modeling in 20-Sim and improving the extendability of 20-Sim and gCSP.

A Creating a custom actor

One of the key features of Ptolemy II is its extendability. This appendix shows an example of how to create a custom actor, both for simulation and code generation.

The instructions below assume that you have installed the Java Development Kit (JDK), which includes the javac binary, that you have make and other tools installed, that Ptolemy II has been installed, and that the environment variable `$PTII` has been set to the Ptolemy II root directory.

A.1 A custom actor for simulation

For this example, we are going to take the Scale actor and change the default factor from one to two. Note that this example commits a cardinal sin of software design: it copies code and makes small changes. It would be far better to subclass the actor and make the necessary changes using object-oriented techniques such as overriding.

1 Create a directory in which to put the new actor. It is most convenient if that directory is in the classpath, which is most easily accomplished by putting it somewhere inside the `$PTII` directory. For this example, we will assume you do:

```
cd $PTII
mkdir myActors
```

2 Create the new .java file that implements the new actor. For this example, just copy Scale.java:

```
cd myActors
cp $PTII/ptolemy/actor/lib/Scale.java ./Scale.java
```

3 Edit myActors/Scale.java and change:

```
package ptolemy.actor.lib;
```

to

```
package myActors;
```

Finally, change the default factor from one to two. This is accomplished by changing:

```
factor.setExpression("1");
```

to

```
factor.setExpression("2");
```

4 Compile the actor:

```
cd $PTII/myActors
javac -classpath $PTII Scale.java
```

5 The new actor is now ready for use in Vergil (The graphical user interface for Ptolemy II). Start Vergil:

```
"$PTII/bin/vergil"
```

6 In Vergil, click on File ⇒ New ⇒ Graph Editor

7 In the graph editor window, select from the Graph menu 'Instantiate Entity'. In the dialog that pops up, enter the classname for the new actor, which is 'myActors.Scale'. An instance of the actor will be created in the graph editor and can be used in a model.

More information on designing Ptolemy II actors can be found in (Bhattacharyya et al., 2007, chapter 5).

A.2 A code generation helper for a custom actor

In order to allow code generation for the custom Scale actor created in the previous section, a helper needs to be created. In this case a helper for the C target language is created. This helper consists of two files: a Java file `Scale.java` and a C file `Scale.c`. Both files are located in:

```
$PTII/ptolemy/codegen/c/myActors/
```

First, create the `Scale.java` file:

```
1  package ptolemy.codegen.c.myActors;
2
3  import ptolemy.codegen.c.kernel.CCodeGeneratorHelper;
4  import ptolemy.kernel.util.IllegalActionException;
5
6  public class Scale extends CCodeGeneratorHelper {
7    public Scale(ptolemy.actor.lib.Scale actor) {
8      super(actor);
9    }
10
11   public String generateFireCode() throws IllegalActionException {
12     super.generateFireCode();
13     ptolemy.actor.lib.Scale actor =
14       (ptolemy.actor.lib.Scale) getComponent();
15     String type = isPrimitive(actor.input.getType()) ? "" : "Token";
16     _codeStream.appendCodeBlock(type + "FireBlock", false);
17     return processCode(_codeStream.toString());
18   }
19 }
```

In lines 13 and 14 the specific instance of the Scale actor in the Ptolemy II model is retrieved. This instance is used in line 15 to determine whether the operands are primitives (i.e. scalar instead of matrix). This information is then used in line 16 to retrieve the appropriate C code for the Scale actor. In line 17 the generated code is processed (macro's are evaluated) and returned to the code generation kernel.

The `Scale.c` file implements C code for primitive and matrix multiplication. For primitives:

```
/***FireBlock***/
// primitive is commutative.
$ref(output) = $val(factor) * $ref(input);
/**/
```

For matrices:

```
/***TokenFireBlock***/
if ($ref(scaleOnLeft)) {
  $ref(output) = Scale_scaleOnLeft($ref(input), (double) $val(factor));
} else {
  $ref(output) = Scale_scaleOnRight($ref(input), (double) $val(factor));
}
/**/
```

The `$ref` macro refers to the value of a port or a parameter in the model. The `$val` macro does essentially the same, but is used for ports or parameters that do not change during model execution, and thus requires less memory. These macro's are processed by the Ptolemy II code generation kernel. The `Scale_scaleOnLeft` and `Scale_scaleOnRight` functions take care of matrix multiplications:

```
/***sharedScaleOnLeftBlock***/
Token Scale_scaleOnLeft(Token input, double factor) {
  int i;
  Token result;

  if (input.type == TYPE_Array) {
    result = $new(Array(input.payload.Array->size, 0));
    for (i = 0; i < input.payload.Array->size; i++) {
      result.payload.Array->elements[i] =
        Scale_scaleOnLeft(Array_get(input, i), factor);
    }
    return result;
  } else {
    return $tokenFunc($new(Double(factor))::multiply(input));
  }
}
/**/

/***sharedScaleOnRightBlock***/
Token Scale_scaleOnRight(Token input, double factor) {
  int i;
  Token result;

  if (input.type == TYPE_Array) {
    result = $new(Array(input.payload.Array->size, 0));
    for (i = 0; i < input.payload.Array->size; i++) {
      result.payload.Array->elements[i] =
        Scale_scaleOnRight(Array_get(input, i), factor);
    }
    return result;
  } else {
    return $tokenFunc(input::multiply($new(Double(factor))));
  }
}
/**/
```

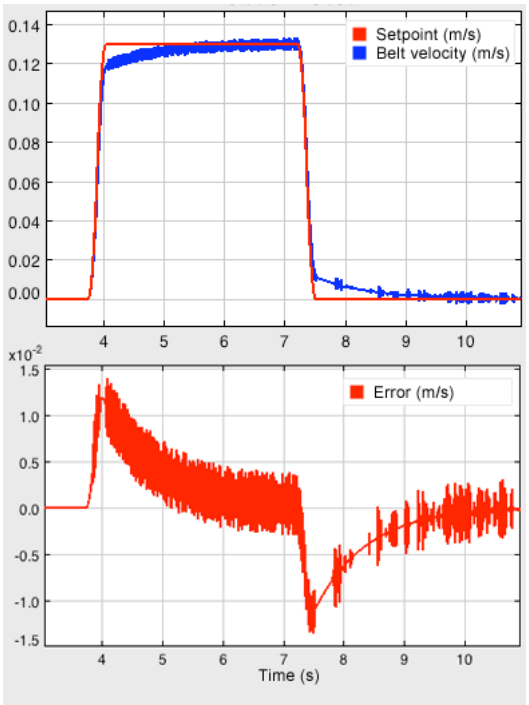
After creation of both the `Scale.java` and `Scale.c` files the helper can be compiled:

```
cd $PTII/ptolemy/codegen/c/myActors/
javac -classpath $PTII Scale.java
```

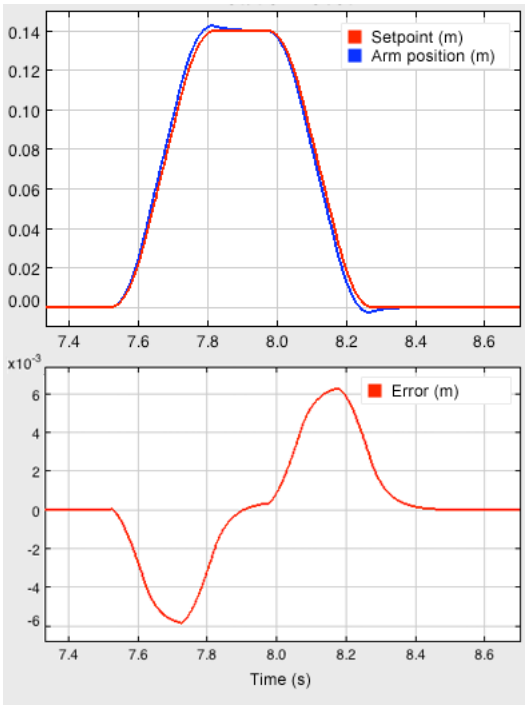
C code can now be generated for the custom Scale actor.

B Simulation results

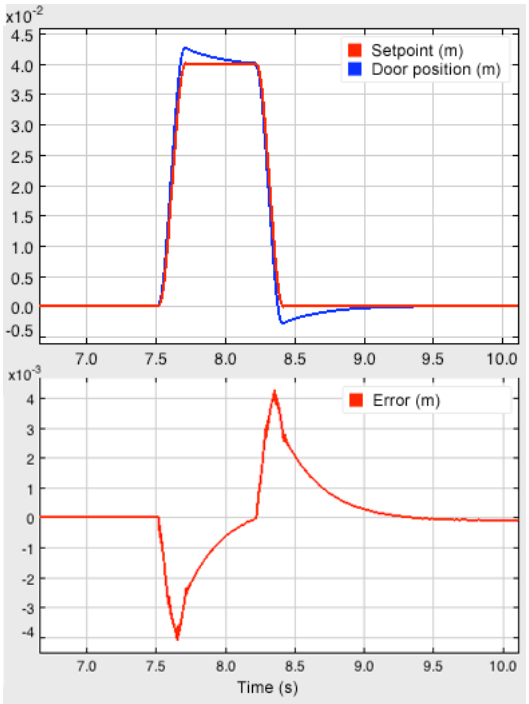
This appendix shows the simulation results obtained in loop controller verification for the feeder belt, the feeder, the molder and the extraction robot.



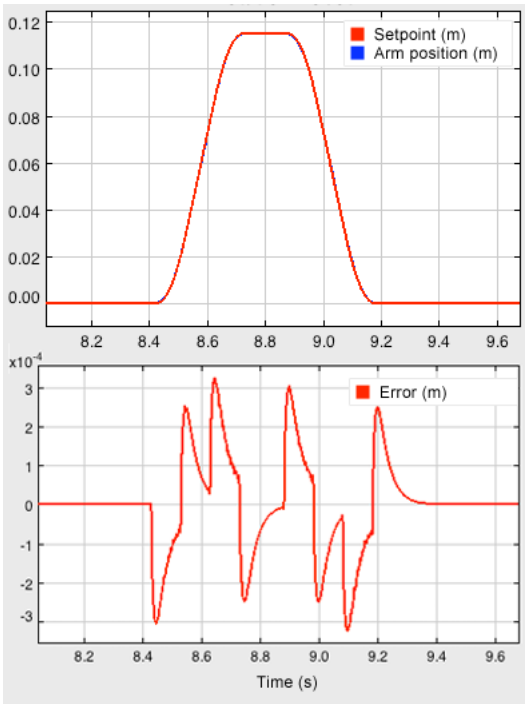
(a) Feeder belt



(b) Feeder



(c) Molder



(d) Extraction robot

C Creating an AVI file from a 3D animation

Ptolemy II does not support the playback of an animation at a speed corresponding to reality. For this reason, and for archiving and presentations, a feature for exporting animations from Ptolemy II is desirable. In this project a simple version of such a feature is implemented. It works by saving each n^{th} frame to a jpeg file. An external utility, mencoder, is then used to combine these jpeg files into an avi file.

In Ptolemy II the ViewScreen3D actor is used to display a window with the 3D animation. This actor was modified to save the jpeg files. The code for the ViewScreen3D actor can be found in:

```
$PTII/ptolemy/domains/gr/lib/ViewScreen3D.java
```

The environment variable `$PTII` should be set to the directory of your Ptolemy II installation.

The following code is added at the end of the `fire()` function of the ViewScreen3D actor:

```
if(_counter % 40 == 0) //export every nth frame
{
  GraphicsContext3D context = _canvas.getGraphicsContext3D();
  Raster raster = new Raster(
            new Point3f(0f,0f,0f),
            javax.media.j3d.Raster.RASTER_COLOR,
            0,
            0,
            _getHorizontalPixels(),
            _getVerticalPixels(),
            new ImageComponent2D(
              javax.media.j3d.ImageComponent2D.FORMAT_RGB,
              getHorizontalPixels(),_getVerticalPixels()) ,
            null);
  context.readRaster(raster);
  BufferedImage image = raster.getImage().getImage();
  try {
    //Create filenames of 8 numbers, for easier use with mencoder
    String number = Integer.toString(_framecounter);
    String zero = new String("0");
    int strlen = number.length();
    int desiredlength = 8;
    for(int length = strlen; length<=desiredlength; length++)
    {
      number = zero.concat(number);
    }
    //Path to save the jpeg files to
    ImageIO.write(
      image,
      "jpg",
      new File("/Users/kees/ProductionCell/jpegs/"+number+".jpg"));
    _framecounter = _framecounter + 1;
  } catch (Exception e) { }
}
_counter = _counter + 1;
```

Two important parameters are hardcoded in this file:

- line 1: this number should be set to the ratio of the sampling frequency and the desired framerate of the AVI file. In this example the sampling frequency is 1000Hz and the desired framerate of the AVI file is 25 frames per second: $\frac{1000}{25} = 40$;
- line 31: the path to where the jpeg files are saved. The jpeg files are sequentially numbered with eight digit numbers, e.g. 00000001.jpg, 00000002.jpg, 00000003.jpg, etc.

After modifying the `ViewScreen3D.java` file, it should be recompiled:

```
cd $PTII/ptolemy/domains/gr/lib/ViewScreen3D.java
javac -classpath $PTII ViewScreen3D.java
```

Now, when running a Ptolemy II model with a ViewScreen3D actor, jpeg files will be saved to the directory specified in the `ViewScreen3D.java` file. These files can then be combined into an AVI file by using mencoder, an open source video utility:

```
mencoder "mf://*.jpg" -mf fps=25 -o animation.avi -ovc lavc \
  -lavcopts vcodec=msmpeg4v2:vbitrate=1600
```

The `fps` parameter indicates the framerate of the target AVI file. This framerate should be set according to the ratio calculated for the `ViewScreen3D.java` file. In this case `animation.avi` is used as an output file.

Bibliography

Baldwin, P., S. Hohli, E. A. Lee, X. Liu and Y. Zhao (2004), Modeling of sensor nets in Ptolemy II, in *Proceedings of the third international symposium on information processing in sensor networks*, ACM, New York, USA, pp. 359–368.

van den Berg, L. (2006), Design of a Production Cell Setup, Technical Report 016CE2006, University of Twente.

Bhattacharyya, S. S., C. Brooks, E. Cheong, J. Davis, M. Goel, B. Kienhuis, E. A. Lee, M.-K. Leung, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, Y. Zhao, H. Zheng and G. Zhou (2007), *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*, Electrical Engineering and Computer Sciences, University of California at Berkeley.

Broenink, J. F. and G. H. Hilderink (2001), A structured approach to embedded control systems implementation, in *IEEE International Conference on Control Applications*, Eds. M. Spong, D. Repperger and J. Zannatha, México City, México, pp. 761–766.

Broenink, J. F., J. Voeten, J. van Amerongen and H. Corporaal (2005), ViewCorrect: Predictable Co-Design for distributed embedded mechatronic control systems, URL `http://www.ce.utwente.nl/ViewCorrect/`.

Brooks, C., E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao and H. Zheng (2007), *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*, Electrical Engineering and Computer Sciences, University of California at Berkeley.

Controllab Products B.V. (2008), 20-Sim, URL `http://www.20sim.com/`.

Corporaal, H. (2006), Embedded system design, in *PROGRESS White Papers 2006*, Ed. F. Karelse, Technologiestichting STW, pp. 7–27.

Damstra, A. (2008), Virtual prototyping through co-simulation in hardware/software and mechatronics co-design, Technical Report 005CE2008, University of Twente.

Dumont, P. and P. Boulet (2005), Another Multidimensional Synchronous Dataflow: Simulating Array-OL in Ptolemy II, Technical report, Université des Sciences et Technologies de Lille.

Eker, J., J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong (2008), The Ptolemy Project, URL `http://ptolemy.eecs.berkeley.edu/index.html`.

Eker, J., J. W. Janneck, E. A. Lee, J. Lui, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs and Y. Xiong (2003), Taming Heterogeneity - The Ptolemy Approach, in *Proceedings of the IEEE*, volume 91, volume 91, pp. 127–144.

Heemels, M. and G. Muller (Eds.) (2006), *Boderc: Model-based design of high-tech systems*, Embedded Systems Institute, Eindhoven, The Netherlands.

Hoare, C. (1985), *Communicating Sequential Processes*, Prentice Hall International.

Huang, J., J. Voeten, M. Groothuis, J. Broenink and H. Corporaal (2007), A model-driven design approach for mechatronic systems, in *Seventh International Conference on Application of Concurrency to System Design*, pp. 127–136.

Jones, N. D., C. K. Gomard and P. Sestoft (1993), *Partial evaluation and automatic program generation*, Prentice Hall International.

Jovanovic, D. S., B. Orlic, G. K. Liet and J. F. Broenink (2004), gCSP: A graphical tool for designing CSP systems, in *Communicating Process Architectures*, Eds. I. East, J. Martin, P. Welch, D. Duce and M. Green.

Kienhuis, B., E. Rijpkema and E. Deprettere (2000), Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures, in *Hardware/Software co-design, Proceedings of the Eighth International Workshop on*, pp. 13–17.

Maljaars, P. (2006), Controllers for the Production Cell Set Up, Technical Report 039CE2006, University of Twente.

Martin, T., M. Jones, J. Edmison and R. Shenoy (2003), Towards a design framework for wearable electronic textiles, in *Wearable computers, Proceedings of the Seventh IEEE International Symposium on*, Virginia Tech, pp. 190–199.

The Mathworks (2008), Matlab/Simulink, URL `http://www.mathworks.com/`.

Visser, P., M. Groothuis and J. Broenink (2007), Multi-purpose toolchain for embedded control system code on a variety of targets, Technical report, University of Twente.

van Zuijlen, J. (2008), Control of the Production cell using Handel-C, Technical Report 008CE2008, University of Twente.