Model-driven design for real-time embedded systems*

Jinfeng Huang[1] and Jeroen Voeten[1,2]

[1] Eindhoven University of Technology, Department. of Electrical Engineering, P.O. Box 513, 5600 MB Eindhoven, The Netherlands;
[2] Embedded Systems Institute P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. The embedded software design is one of the most challenging tasks during the design of a complex real-time system. On one hand, it has to provide solutions to deal with concurrency and timeliness issues of the system. On the other hand, it has to glue different disciplines (such as software, control and mechanical) of the system as a whole. In this paper, we propose a model-driven approach to design the software part of a complex real-time system, which consists of two major parts: systematic modeling and correctness-preserving synthesis. The modeling stage is divided into four steps, which focus on different aspects (such as concurrency, multiple disciplines and timeliness) of the system respectively. In particular, we propose a set of handshake patterns to capture the concurrent aspect of the system. These patterns assist designers to build up an adequate top-level model efficiently. Furthermore, they separate the system into a set of concurrent components, each of which can be further refined independently. Subsequently, the multidisciplinary and real-time aspects of the system are naturally specified and analyzed in a series of refinements. After the important aspects of the system are specified and analyzed in a unified model, a software implementation is automatically synthesized from the model, the correctness of which is ensured by construction. The effectiveness of the proposed approach is illustrated by a complex production cell system.

1 Introduction

Industry has to deal with more and more complex real-time systems which involve multiple disciplines and complex functionalities. The software design of these systems is usually a challenging and problematic task, due to the following two major reasons.

– The software design usually starts after the design in other disciplines (e.g. mechanics and electronics) is finished, which often results in long design iterations. Furthermore, since the design of other disciplines is often fixed during the software design, it is difficult to obtain an optimal solution for the whole system.

* An extended version of this paper appears in the proceeding of IEEE International Conference on Application of Concurrency to System Design 2007. This research is supported by PROGRESS, the embedded systems research program of the Dutch organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

– The software design involves non-trivial interactions between multiple disciplines and between the different functional components. The design decisions made on a single functional component or in one discipline can have impact on the behaviors of other parts of the system. These impacts are often difficult to foresee before system integration in most existing design approaches. Therefore, unexpected software behaviors are only observed in a late design stage, which results in long and costly design iterations.

To overcome these design difficulties and improve the design efficiency, a systematic approach is required to design the embedded real-time software, which has the following characteristics.

– Concurrent engineering: the software design is carried in parallel with the design in other disciplines, which accelerates the design process and facilitates the optimization of design solutions.
– Local refinement: each design step should focus only on partial information (a component or an aspect) of the system, which reduces the design complexity.
– Predictable refinement: when different parts (components or aspects) of the system are integrated together, the original properties of each part should be preserved in the integration. In this way, design efficiency can be largely improved.

In this paper, we introduce a model-driven approach for the embedded real-time software design. The approach starts from a top level abstraction of the system. Then a sequence of refinements are carried out to focus on different aspects (concurrency, multi-disciplines and real-time) of the system. In the end, the final implementation is automatically generated from the model. In the following part of the paper, we illustrate how the concurrent engineering, local refinement and predictable refinement are supported by the proposed approach.

The model-driven approach proposed in the paper consists of two major parts: systematic modeling and correctness-preserving synthesis. The following discussion focuses on the first part of the approach. Details of the second part can be found in [5].

During the modeling stage, different aspects of the system, such as concurrency, multiple disciplines and timeliness need to be investigated based on a series of models. We show that these aspects can be investigated at different abstraction levels of the system. By properly ordering the investigation sequence, these aspects can be analyzed in a step-wise and property-preserving way. The design process starts from an informal description (called handshake diagrams) of the system, which can be easily constructed following the proposed guidelines and gives a natural representation of the system. Based on the handshake diagram, a C-model can be derived to investigate the concurrency aspect of the system. Then the multidisciplinary aspect is analyzed in a refined model (M-model) of the C-model. In the end, the real-time aspect is specified in a refined model (R-model) of the M-model. Furthermore, the consistency between different abstraction levels can be maintained so that properties analyzed at a higher level model are still valid at lower level models.

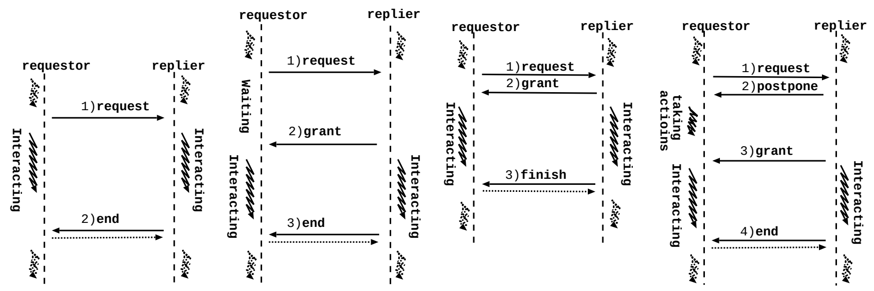
2 Handshake diagram

A natural way to separate a system is to divide it into a set of concurrent components, each of which is an autonomous entity consisting of mechanics, electronics and/or software. We use the player concept in [6] which abstracts each entity from different disciplines. We classify these components into active and passive players. The identification of active players is mainly based on the active physical elements of the system (such as actuators in a system). Sometimes, these active players interact with each other through a buffer instead of in a direct manner. These intermediates between active players are considered as passive players.

These players in the system interact with each other to achieve manufacturing goals. However, the interactions between these players cannot occur unconditionally. For instance, When a robot can put a block into a buffer only when the buffer has free space. To avoid conflicts and operation reliability, additional negotiation is required before these players carry out their interactions. We use the handshake mechanisms to specify these negotiations between players. These handshakes between players not only define the point where interactions between players can occur properly, but also well separate the internal behavior of each player from other players, which facilitates local refinements in later design stages. During the handshake process, two players are involved. The one initializing the handshake process is called requestor and the other is replier. According to different types and states of the requestor, we propose three different handshake patterns.

– *Two-way handshake* This mechanism is used if the requestor is a passive player. As shown in Figure 1-a, when the requestor asks the other player (which must be an active player) to carry out the interactions between them, it only need to wait for the end of the interactions.
– *Three-way handshake* This mechanism is used in case that the requestor is an active player but it is in an inactive state. As shown in Figure 1-b, the requestor first sends the request to the replier. Then the requestor waits for the grant from the replier, which indicates that the replier is also ready for the interaction. After the interaction between them finishes, one informs the other about the end of interaction.
– *Four-way handshake* This mechanism is used in the situation that the requestor is an active player and it is also in an active state. In this case, after the requestor sends the request to the replier, it has to take actions based on the state of the replier. Figure 1-c shows two scenarios of the four-way handshake. The requestor first sends the request to the replier. Then the replier immediately replies (postpone or grant) to the requestor according to its states. If the request is granted, the interaction between them can occur. In case that the request is postponed, the requestor has to take certain actions and wait until the partner grants the request.

It is not difficult to conclude that the two-way handshake between two participants can be seen as a special case of the three-way handshake. The same conclusion also holds for the three-way handshake mechanism and the four-way handshake mechanism. Despite these facts, the two-way and three-way handshake mechanisms are still introduced separately because they have less synchronization overhead.



(a) Two-way handshake (b) Three-way handshake (c) Four-way handshake

Fig. 1. The synchronization mechanisms

3 C-model

In the C-model, the system consists of a set of players derived from the handshake diagram. The C-model is a high level abstraction of the system, where only "abstract" interactions between players are considered. The details of other disciplines are not necessary to construct the C-model. Therefore, the embedded software design can be carried in parallel with the design of the rest system. It is usually a difficult task to obtain an adequate C-model, which often relies on the designers' experience and wisdom. However, by using the handshake diagram, we can derive an adequate C-model in the following steps.

1. Identify the number of concurrent interactions. Each player consists of a set of concurrent activities, each of which performs the handshake with one of other players.
2. Define the condition for the handshake. Each requestor and replier perform the handshakes based on its own states.

In Section 6, we show how to derive the C-model from a handshake diagram in a concrete example. In the C-model, properties (such as deadlock and resource access conflicts) relating with the interactions between the concurrent players can be formally analyzed.

4 M-model

A complex real-time system is built by several disciplines. The embedded software of the system glues these disciplines as a whole. In general, software engineering mainly addresses the high-level discrete event control of the system, which plans actions for the physical elements. Control engineering deals with the low-level continuous control of the system, which derives stable and optimal control algorithms for the physical elements. Mechanical engineering applies principles of physics to implement the physical elements. When a high-level control unit generates an action for a physical element, the action is interpreted by a corresponding control algorithm in a low-level control unit.

For instance, when a high-level control unit issues an action “motor A: start to move”, this action is actually connected to a low-level control loop which ensures that the physical motor starts to move steadily according to a predefined motion profile. On the other hand, the low-level control keeps on monitoring the states of the physical elements and provides events to the high-level control, which trigger the high-level control to plan next actions for the physical elements. For instance, a low-level control unit keeps on monitoring the position of a physical element, and when the element reaches a crucial position, the low-level control unit generates an event to its high-level control unit. Consequently, the high-level control unit plans the next action for the physical element.

We use an M-model to specify the interactions between different disciplines. To maintain design consistency, the M-model is obtained by enhancing the C-model with more details but keeping the same observable behavior at the same time. In this way, the properties verified/tested in the C-model can be preserved in the M-model. Roughly speaking, each player in the C-model is further split into two kinds of processes in the M-model, which correspond to high-level control and low-level control. To simplify the analysis of the interactions between different disciplines, the behaviors of the low-level control units are specified at a discrete event level of abstraction in the M-model. On one hand, this allows that the major interactions between the different disciplines are analyzed in a relatively simple model. On the other hand, this model also provides a framework for later integration of the continuous time behavior in a straightforward way.

5 R-model

In previous models, the system behavior is analyzed qualitatively. However, a complex real-time system often involves continuous time behaviors (such as the continuous movement of its mechanical part) where its states change over time. To address the quantitative aspects of the system, we need to incorporate the timing information and the continuous time behavior into the model. The timing information for the high-level control can be specified by the quantitative timing relations between adjacent discrete events (such as action_a; delay t; action_b). The timing information for the low-level control can be added to the model by incorporating control loops. By defining the switch conditions for discrete events based on the state change in a control loop, the timing relations between the events are specified indirectly. After different aspects of a system are incorporated into a unified model, the gap between requirements and implementations has been largely filled. The remaining work is to generate an implementation which has the expected behavior as specified in the model. We use the synthesis approach proposed in [5] to generate the implementation from the model correctly by construction. In the next section, we demonstrate our approach by designing a rather complex industrial system.

6 Case study

To demonstrate how the proposed approach can be applied to design the embedded real-time software, we use a production cell system as the case study, which simplifies a realistic industrial system [3]. There are six actuators and more than a dozen sensors in the system, which cooperate with each other to manufacture a product.

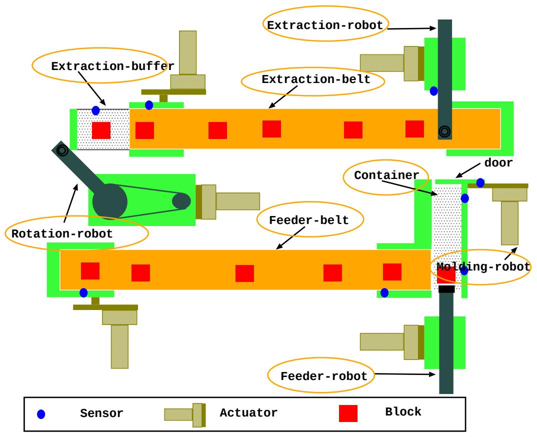


Fig. 2. The production cell system

6.1 System description

As shown in Figure 2, the primary products (in this case the metal blocks) are put into the system at the left-end of the Feeder_belt. The Feeder_belt transports these primary products to the Feeder_robot, which pushes the primary product into the Container. In the industrial version of the production cell system, the blocks are pieces of plastics, which are molded inside a molding machine. In our demo system, the molding machine is represented by a simple device (called the Molding_robot), which opens or closes the door of the Container to emulate the behavior of the molding machine. Before the blocks are pushed into the Container, the door has to be closed. After the block is processed in the Container, the door opens and the Extraction_robot picks up the block using an electromagnet. Then the block is moved to the Extraction_belt, which delivers it to a fixed table (called the Extraction_buffer). For the purpose of demonstration, the Rotation_robot is installed at the end of the Extraction_buffer. Using an electromagnet, the Rotation_robot transfers the block from the table to the Feeder_belt, such that the block can enter the production cycle again.

6.2 System modeling

It is easy to identify that the production system consists of 8 players, among which 6 are active players corresponding to six motors and two others are passive players. Using the introduced handshake patterns, we can easily draw the handshake diagram for the system as shown in Figure 3.

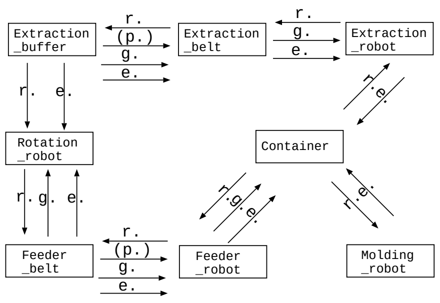
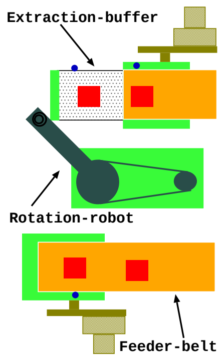


Fig. 3. The handshake diagram of the production cell system



(a) Physical interactions

```
Input()()
   in? request;
   available:=true;
   [empty] skip;
   [empty=false] skip;
   in? end{available:=false};
Input()().
```

(b) Synchronization with Extraction_buffer

```
Output()()
   [available] out! request;
   out? grant
   /* pick up the block*/
   empty:=false;
   /* move the block to Feeder_belt */
   [available=false]out! end;
   /* move back to Extraction_buffer*/
   empty:=true;
Output()().
```

(c) Synchronization with Feeder_belt

Fig. 4. The behavior of Rotation_robot

In the following, we use one example to demonstrate how to make an adequate abstraction for each player in the C-model based on the handshake diagram.

Example 1. **The Rotation_robot player**
The Rotation_robot interacts with the Extraction_buffer and the Feeder_belt in the system (as shown in Figure 4-(a)). We use two POOSL [3] process methods (Input()() and Output()()) to specify the handshakes between them. More specifically, Input()() defines the handshake between the Extraction_buffer and the Rotation_robot and the Output()() specifies the other. Both process methods are combined as two concurrent activities.

```
par
   Input()()
and
   Output()()
rap.
```

[3] Due to the limited size of the paper, more information of the POOSL language can be found at the website http://www.es.ele.tue.nl/poosl/.

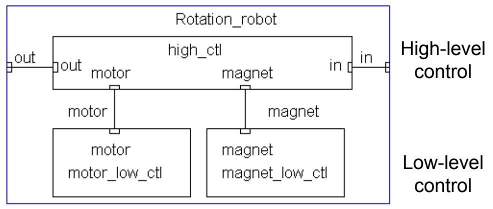


Fig. 5. The refinement of Rotation_robot in the M-model

We use a two-way handshake mechanism to negotiate the interaction between the Extraction_buffer and the Rotation_robot. Figure 4-(b) illustrates the handshakes between the Rotation_robot and the Extraction_buffer, where the Rotation_robot is always willing to receive the request (no guard is put in front of “in? request”.). After it receives a request, a flag “available” is set, which indicates that the delivering block is in the Extraction_buffer area. The consequent behavior is guarded by a flag “empty” representing that the Rotation_robot is ready to pick up a block. After the block is picked up (“empty” becomes false), the Rotation_robot receives an end message from the Extraction_buffer, which indicates the block has left the extraction buffer area.

When the Extraction_buffer interacts with the Rotation_robot, it also interacts with the Feeder_belt at the same time, which is specified by another parallel activity in Figure 4-(c). The Extraction_buffer first requests to the Feeder_belt, when a block is ready for the delivering. It waits until the Feeder_belt is ready to receive the block (out? grant). After that, the block is delivered to the Feeder_belt. A guard ([available=false]) is put in front of out? end to ensure the logic correctness at this abstraction level. Namely, the interaction with the Feeder_belt is always finished later than that with the Extraction_buffer. The comments in Figure 4-(c) guides the refinement in later design stages.

The above example illustrates how to abstract a system in the C-model. By identifying the number of concurrent interactions, choosing handshake mechanisms for interactions and defining handshake conditions, an adequate abstraction of the system can be easily specified. Although this abstraction is at a very high level, many crucial system properties related with resource access conflicts can already be analyzed. For instance, in the C-model of the production cell system, the following properties can be verified.

– The system is always deadlock free, when there are less than 8 blocks in the system.
– There is at most one block in the Extraction_Buffer.
– There is at most one block in the Container.
– The Feeder_robot can only push a block into Container when the door is closed and the Container is empty.

As we have mentioned previously, players negotiate with each other for reliable “physical” interactions between them (as shown in Figure 1). However, these interactions are not explicitly specified in the C-model but in the M-model.

Example 2. Reconsider the Rotation_robot player in Example 1, where its handshake mechanisms for interactions with the other two players have been specified. But the actual interactions between these players are left out. In the M-model, these behaviors are incorporated in a systematic way into two layers (high-level and low-level controls).

```
Input()()
   in? request;
   available:=true;
   [empty] skip;
   [empty=false] skip;
   in? end{available:=false};
Input()().
```

(a)

```
Output()()
   [available] out! request;
   out? grant;
   magnet! on;
   empty:=false;
   motor! start;
   motor? stop;
   magnet! off;
   [available=false] out! end;
   motor! reverse;
   motor! start;
   motor? stop;
   motor! reverse;
   empty:=true;
Output()().
```

(b)

Fig. 6. The high-level control in the Rotation_robot

As shown in Figure 5, the Rotation_robot is further refined into 3 processes: high_ctl, motor_low_ctl and magnet_low_ctl. The high_ctl process performs the high-level control of the player. Its behavior can be extended naturally by adding "interactions" to the behavior of the Rotation_robot in the C-model. Recall the handshake mechanisms in Figure 1. The interacting behavior between players are preformed during their handshakes. For instance, after the Rotation_robot receives the grant message from the Feeder_belt, both of them are ready to carry out the actual "physical" interaction. As shown in Figure 6-(b), the Rotation_robot turns on the magnet to pick up the block and then starts its motor to move the block to the Feeder_belt. When the robot reaches the feeder belt ("motor? stop"), it releases the block by turning off the magnet. After the block is released, the Rotation_robot reverses its direction ("motor! reverse") and then moves back to the table to pick up the next block.

```
Discrete()()
   motor? start;
   motor! stop;
   motor ? reverse;
   motor? start;
   motor! stop;
   motor? reverse;
   Discrete()().
```

Fig. 7. The behavior of the motor_low_ctl process

Since the low-level control processes (e.g. the motor_low_ctl and the magnet_low_ctl process in Figure 5) act as an interpreter between the high-level control and physical elements, their discrete-event behavior can be specified in a rather straightforward way. For instance, Figure 7 illustrates the discrete event behavior of the low-level control for the rotation motor. It either carries out the actions from the high-level control on the physical element (e.g. motor? start;), or triggers discrete events based on the state of the physical element (e.g. motor! stop;).

```
Discrete()()
   motor? start {rot Start; running:=true};
   [rot Active=false] motor! stop{rot Stop; running:=false};
   motor ? reverse{rot SetBackwards};
   motor? start {rot Start; running:=true};
   [rot Active=false] motor! stop{rot Stop; running:=false};
   motor ? reverse{rot SetForwards};
Discrete()().
```

```
Continuous()()
   [running] position:= rot Read;
   nextpos:= rot Calculate (position);
   rot Write (nextpos);
   delay 0.001;
Continuous()().
```

Fig. 8. The behavior of the motor_low_ctl process

During system design, control algorithms in the low-level control are usually analyzed and generated by other commercial tools such as simulink [2] or 20-sim [1][4]. To provide a natural integration of them in the R-model, they are first represented by data methods, which are replaced by actual algorithms during system synthesis. A similar idea can also be used to represent physical communications such as reading a sensor value in the R-model. The details about implementing an interface data for POOSL model can be found in [5]. The following example briefly illustrates how to add the continuous time behavior into the model and prepare the final blueprint for the system synthesis.

Example 3. The motor_low_ctl process in Figure 7 incorporates the continuous time behavior by specifying a separate activity (Continuous()()), which is in parallel with the discrete one (Discrete()()) as shown in Figure 8. As we have mentioned perviously, the control algorithm is encapsulated into a data class ("rot" in this case), which also charges physical communications such as reading a value from the position sensor and writing a value to the actuator. Therefore, a typical control loop with the frequency 1000 times/s can be specified by the Continuous()() activity as shown in Figure 8. The data object "rot" only defines communication interfaces in the R-model, which has no direct communication capability with the physical world. During software synthesis, it will be replaced by its counterpart in the C++ implementation, which provides the actual physical communication with the physical world.

The continuous time behavior does not interact directly with the high-level control (the high_ctl process in Figure 5). It only interacts with the discrete event behavior inside the low-level control by imposing switch conditions on events. For instance, the condition "[rot Active=false]" indicates that the rotation arm reaches the feeder belt or the extraction buffer. Consequently, events should be triggered in the low-level control to stop the physical motor completely ("rot Stop") and to inform the high-level control ("motor ! stop"). Therefore, the interactions between the low-level control (the motor_low_ctl process) and the high-level control (the high_ctl process) remain the same as those in the M-model.

6.3 System synthesis

When a model has sufficiently described the system behavior, it is desirable during the system synthesis to generate an implementation which has the same desired properties as that of the model. By using the synthesis approach in [5], the R-model in the previous step can be transformed into an implementation automatically. Figure 9 gives a snapshot of the running system, which is controlled by the synthesized software.

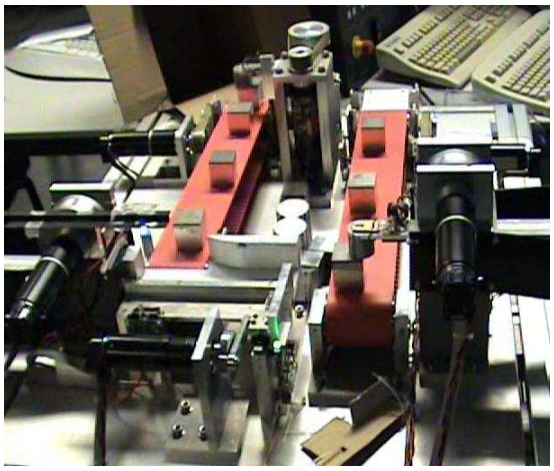


Fig. 9. A snapshot of the implementation

– The execution of the implementation is based on its virtual time semantics. In this sense, the implementation has exactly the same behavior as that of the model.
– The virtual time and the physical time are synchronized. Therefore, the behavior of the implementation is almost the same as that of the model. The synchronization errors between two time domains can be used to predict the property deviation of the implementation in the physical time domain from the model in the virtual time domain.

7 Conclusions

Dynamics of the market and fast time to market demand an efficient design process for complex real-time systems. Software design is usually most problematic and inefficient part in the whole design process. In this paper, we proposed a model-driven approach tailored for embedded real-time software design to reduce design complexity and to improve design quality. The proposed approach embeds three distinguishing characteristics into its design steps: concurrent engineering, local refinement and predictable refinement. The software design process starts from the top-level view of a system, which requires little details of other disciplines. Furthermore, by using the proposed handshake mechanisms, an adequate top-level model (C-model) can be constructed naturally. Inside the C-model, the system is divided into a set of concurrent players, each of which can be further refined locally. Although the C-model is at a high abstraction level, many important safety properties can already be investigated. In the first refinement (M-model) of the C-model, each player is independently refined into high-level and low-level controls, which facilitates the local refinement in the R-model. The high-level control plans actions for the physical elements of the system and the low-level control operates the physical elements in a stable and optimal way. The M-model focuses on the interactions between two control levels. To ensure the predictable refinement, each player has the same observable behavior as that in the C-model. Consequently, properties of the C-model can be preserved. In the refinement (R-model) of the M-model, the continuous time behavior is incorporated inside the low-level control. Consequently, this refinement can be carried out locally inside each low-level control. The interactions between high-level and low-level controls remain the same as those in the M-model, which ensures the R-model is a predictable refinement of the M-model. The effectiveness of the proposed approach is illustrated by the design of an industrial-size production cell system.

References

1. 20-sim. http://www.20sim.com/, Jan. 2007.
2. Simulink. http://www.mathworks.com/products/simulink/, Jan. 2007.
3. L. v. d. Berg. Design of a production cell setup. Technical Report MSc-Report 016CE2006, The Netherlands, July 2006.
4. M. Groothuis and J. Broenink. Multi-view methodology for the design of embedded mechatronic control systems. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, pages 416–421, October 2006.
5. J. Huang, J. Voeten, and H. Corporaal. Predictable real-time software synthesis. *To be published in journal of real-time systems*.
6. P. van der Putten and J. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.