A model-driven design approach for mechatronic systems*

Jinfeng Huang[1], Jeroen Voeten[1,2], Marcel Groothuis[3], Jan Broenink[3] and Henk Corporaal[1]
[1] Eindhoven University of Technology, Department. of Electrical Engineering;
[2] Embedded Systems Institute
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[3] University of Twente, Faculty of EE-Math-CS,
P.O.Box 217, 7500 AE Enschede, The Netherlands
J.Huang@tue.nl

Abstract

The software design is one of the most challenging tasks during the design of a mechatronic 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 mechatronic 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

Nowadays, the industry has to deal with more and more complex mechatronic systems which involve multiple disciplines and complex functionalities. The software design of these systems is usually a most 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.

- 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 software part of a mechatronic system, 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 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

Characteristics	Models	Abstraction levels
Concurrency	C-model	High
Multi-disciplines	M-model	↓
Real-time	R-model	Low

Figure 1. The key models in the approach

in the integration. In this way, design efficiency can be largely improved.

In this paper, we introduce a model-driven approach for the software design of mechatronic systems. 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 remaining part of the paper is organized as follows. Section 2 gives an overview of the proposed approach. Section 3 illustrates a production system which is used as an example to illustrate the approach in later sections. Section 4 presents a systematic modeling approach for the software design of mechatronic systems based on the POOSL language [4]. The approach consists of three major steps, which address different aspects of the system respectively. Section 5 briefly reviews the synthesis approach in [6] used to generate an implementation from a model in a correctness-preserving way. Section 6 concludes the paper.

2 Overview of the approach

In this paper, we present a model-driven approach, which 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 [6].

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 (see Figure 1). 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.

- **Handshake diagram** In the handshake diagram, the system is considered as a set of concurrent components (called players), which operate independently and communicate with each other. The identification of the players in a system is mainly based on its physical elements. These players interact with each other to achieve certain functional goals. A set of handshake patterns are proposed, which ensure the proper interactions between players.

- **C-model** In the C-model, the system consists of a set of players derived from the handshake diagram. The C-model focuses on the untimed interactions between different players, which requires little information from other disciplines. Different from the handshake diagram, the conditions of the handshakes between players are explicitly defined. In the C-model, properties (such as deadlock and resource access conflicts) relating with the interactions between the concurrent players can be formally analyzed. It is usually a difficult task to obtain an adequate C-model, which often relies on the designers' experience and wisdom. In this paper, we show that an adequate C-model can be obtained from the handshake diagram smoothly.

- **M-model** A mechatronic system typically is built by different disciplines including software, control and mechanical engineering. Correspondingly, the interactions between these disciplines are reflected by the interactions between the high-level discrete control, the low-level continuous control. In the M-model, these interactions are first specified and analyzed in a discrete-event model. We illustrate that the refinement from the C-model to the M-model can be carried out locally in a predictable way.

- **R-model** In previous models, the system behavior is analyzed qualitatively. However, a mechatronic system also 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. We show that a consistent refinement from the M-model to the R-model can be carried out locally.
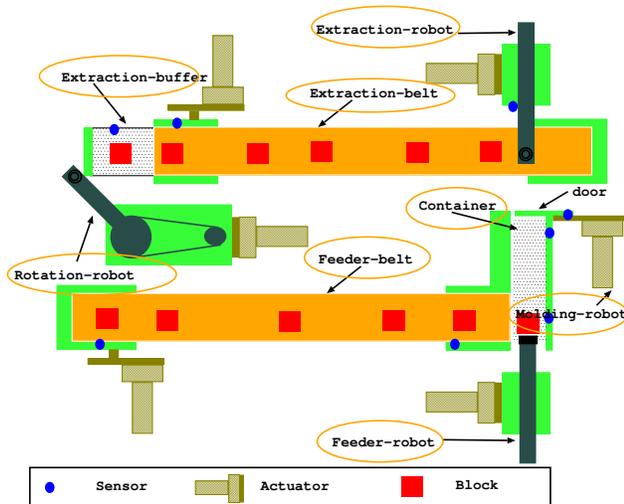
Figure 2. The production cell system

3 System description

To demonstrate how the proposed approach is applied to design the software of a complex mechatronic system, 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.

As shown in Figure 2, the primary products (in this case the metal blocks) are put into the system at the left-end of Feeder_belt. Feeder_belt transports these primary products to Feeder_robot, which pushes the primary product into 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 Molding_robot), which opens or closes the door of Container to emulate the behavior of the molding machine. Before the blocks are pushed into Container, the door has to be closed. After the block is processed in Container, the door opens and Extraction_robot picks up the block using an electromagnet. Then the block is moved to Extraction_belt, which delivers it to a fixed table (called Extraction_buffer). For

the purpose of demonstration, Rotation_robot is installed at the end of Extraction_buffer. Using an electromagnet, Rotation_robot transfers the block from the table to Feeder_belt, such that the block can enter the production cycle again.

4 System modeling

In this section, we first give a short review of the modeling language used in the paper. Afterwards, each major step in the modeling stage is introduced in details. The consistency between these steps are also demonstrated by using the production cell system as a case study.

4.1 Modeling language: POOSL

The POOSL (Parallel Object-Oriented Specification Language) is a general purpose modeling language, which integrates a process part based on a timed and probabilistic extension of CCS [7] and a data part based on the concepts of traditional object-oriented languages. Therefore, in addition to the control flow statements in traditional languages such as loop and choice statements, POOSL provides a set of primitives to specify e.g. parallelism, non-determinism, interruption, synchronous communication and time. Here we give a brief explanation of the language to help in understanding the examples in the paper. More detailed information about the language can be found in [4] [9].

A POOSL model consists of a set of concurrent processes connected by static channels. Each process has its own data space. It can only share its information with other processes through synchronous communication. Communications between processes are accomplished through ports connected by static channels. For instance, statement "out! request" indicates the willingness to send a request message through port "out" and "in? request" indicates the willingness to receive a request message from port "in". When the "in" and "out" ports are connected by a channel, both parts are synchronized and the communication is performed.

In addition to the parallelism between processes, a finer grain of parallelism (concurrent activities) can be also specified inside a process using the **par** statement ("par S_1 and...and S_n rap"). Each activity can share a data space with other activities, and exchange its information with others through shared data. One way for concurrent activities to interact with each other is through the use of guard statements. For instance, consider an n-size buffer in a producer-consumer example. The behavior of the buffer can be specified by two concurrent activities as in Figure 3. Both activities interact with each other through a shared variable "size". The guard "[size<n]" in the first activity specifies that the buffer can receive a product from the producer only when it is not full. After a product is inserted into the buffer, "size:=size+1" is executed atomically. The second activity

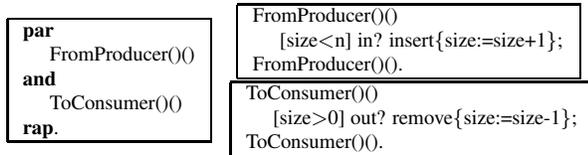|	

par
    FromProducer()()
and
    ToConsumer()()
rap.
	FromProducer()()
	    [size<n] in? insert{size:=size+1};
	    FromProducer()().
	ToConsumer()()
	    [size>0] out? remove{size:=size-1};
	    ToConsumer()().

Figure 3. The behavior of the n-size buffer

is enabled when the buffer is not empty. We can also see that the infinite behavior of both activities can be easily expressed in a tail-recursive fashion. The POOSL language provides the "delay" primitive to specify timing information in the model. Similar to many other formal languages, the timing semantics of POOSL relies on the two-phase execution model in [8]. The state of the model can change either by asynchronously executing actions (Phase 1) or by synchronously consuming time (Phase 2). Time advances only when no action can be performed. This timing semantics assumes that actions are instantaneous in the model, which largely simplifies the analysis of the timing behavior.

Different from the traditional "if" statement, non-deterministic selection ("sel S_1 or...or S_n les") does not specify conditions for its branches. This facilitates designers to abstract system behaviors. This is because there are not always enough details available to determine the conditions when making an abstraction of a system behavior.

4.2   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 [10] 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 the belts and the robots in the production cell system). Sometimes, these active players interact with each other through a buffer instead of in a direct manner. The intermediate between active players are considered as passive players (such as the container and the extraction buffer in the production cell system). For instance, the production cell system we mentioned earlier can be partitioned into 8 players as shown in Figure 2.

These players in the system interact with each other to achieve manufacturing goals. However, the interactions between these players cannot occur unconditionally. For instance, Rotation_robot is not allowed to put a block at the left-end of Feeder_belt, if there is already a block occupying the position. 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. 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. As shown in Figure 4-a, the requestor first sends a request to the replier. Then the replier carries out the actual interaction when it is ready. In the end, one player informs the other about the end of the interaction. For instance, we can use this handshake mechanism when Container asks Extraction_robot to fetch a block inside it.

- *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 4-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. For instance, when a block arrives at Feeder_robot, Feeder_robot starts to request Container to insert the block. Container may have another block at the moment. In this case, Container does not react to Feeder_robot until it is empty.

- *Four-way handshake* This mechanism is used in the situation that the requestor is an active player and it is 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 4-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. Consider the scenario when Feeder_belt wants to put a block to Feeder_robot. Feeder_robot can be either occupied or not at that moment. Feeder_robot has to inform Rotation_robot immediately so that the Feeder_belt can either stop or continue moving without intruding the safety constraint[1].

---

[1]To avoid breaking the system physically, at most one block can be in

COMPUTER
SOCIETY

(a) Two-way handshake   (b) Three-way handshake   (c) Four-way handshake
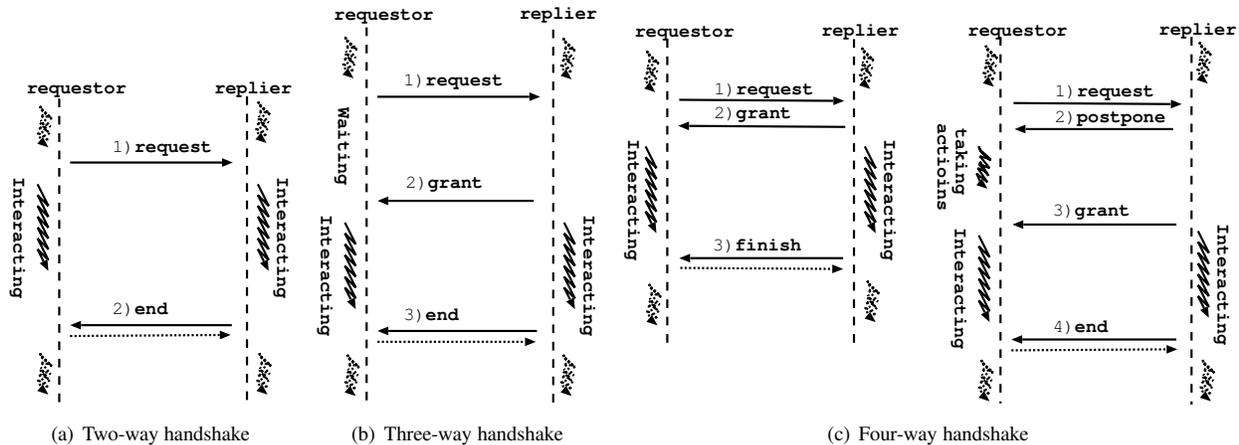
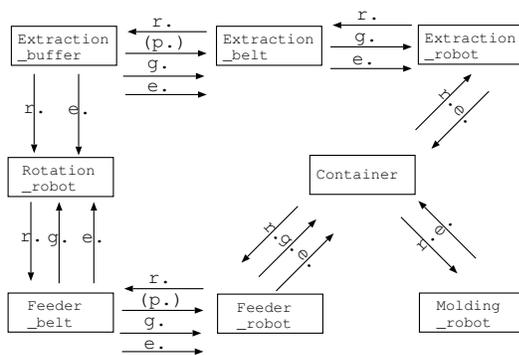**Figure 4. The synchronization mechanisms**

Figure 5. The handshake diagram of the production cell system

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.

Using the introduced handshake patterns, we can easily draw the handshake diagram for a system. For instance, the handshake diagram of the production cell system can be constructed as shown in Figure 5.

4.3 C-model

The C-model is the first "formal" model in the design process in the sense that some properties of the system can

---

Feeder_robot.

be analyzed by using simulation or verification techniques. To support the concurrent engineering, the C-model is a high level abstraction of the system, where only "abstract" interactions between players are considered. The details of the mechatronic devices such as motor speed and belt length are not required in the model. To ensure the local refinement of each player in later design stages, each player should be always ready to receive the request from other players and it can also request other players without knowing their states.

Having the handshake diagram, we can derive an adequate C-model from it in two 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. For instance, Rotation_robot interacts with two other players in the system. It has two concurrent activities to perform the handshakes with them respectively (see also Example 1).

2. Define the condition for the handshake. Each requestor and replier perform the handshakes based on its own states. For instance, Extraction_buffer requests to Rotation_robot when it is occupied by a block.

In the following, we use several examples to demonstrate how to make an adequate abstraction for each player in the C-model.

Example 1 *The Rotation_robot player*

Rotation_robot interacts with Extraction_buffer and Feeder_belt in the system (as shown in Figure 6-(a)). We use two POOSL process methods (Input()() [2] *and Output()())*

---

[2]In the POOSL syntax, the two pairs of braces in the definition of a process method are used to contain an input and a output parameter lists. In this example, both parameter lists are empty.
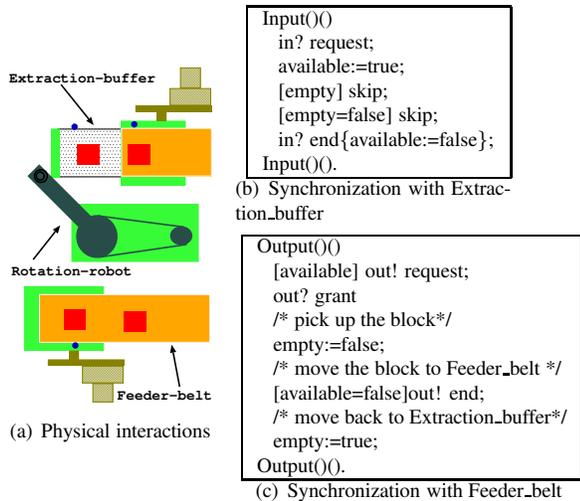
(a) Physical interactions

```
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

Figure 6. The behavior of Rotation_robot

to specify the handshakes between them. More specifically, Input()() defines the handshake between Extraction_buffer and Rotation_robot and Output()() specifies the other. Both process methods are combined as two concurrent activities.

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

We use a two-way handshake mechanism to negotiate the interaction between Extraction_buffer and Rotation_robot. Figure 6-(b) illustrates the handshakes between Rotation_robot and Extraction_buffer, where 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 Rotation_robot is ready to pick up a block. After the block is picked up ("empty" becomes false), Rotation_robot receives an end message from Extraction_buffer, which indicates the block has left the extraction buffer area.

When Extraction_buffer interacts with Rotation_robot, it also interacts with Feeder_belt at the same time, which is specified by another parallel activity in Figure 6-(c). Extraction_buffer first requests to Feeder_belt, when a block is ready for the delivering. It waits until Feeder_belt is ready to receive the block (out? grant). After that, the block is delivered to 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 Feeder_belt is always finished later than that with Extraction_buffer. The

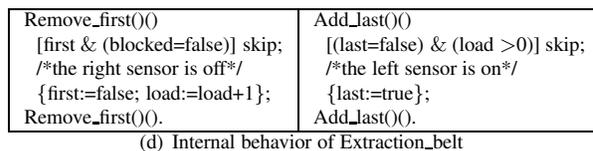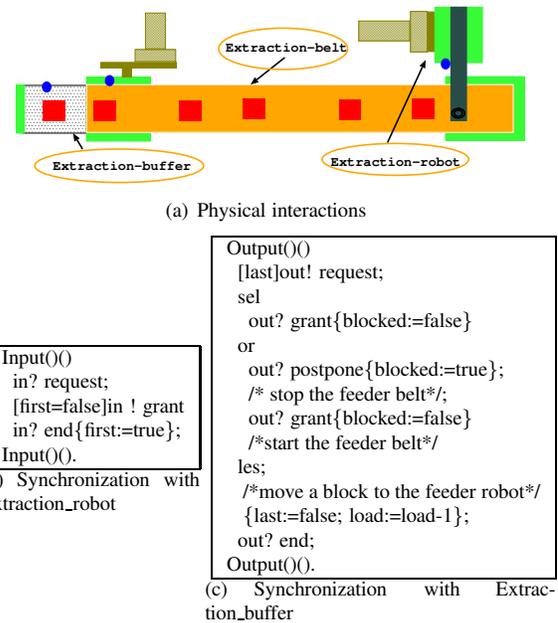(a) Physical interactions

```
Input()()
  in? request;
  [first=false]in ! grant
  in? end{first:=true};
Input()().
```
(b) Synchronization with Extraction_robot

```
Output()()
  [last]out! request;
  sel
    out? grant{blocked:=false}
  or
    out? postpone{blocked:=true};
    /* stop the feeder belt*/;
    out? grant{blocked:=false}
    /*start the feeder belt*/
  les;
  /*move a block to the feeder robot*/
  {last:=false; load:=load-1};
  out? end;
Output()().
```
(c) Synchronization with Extraction_buffer

```
Remove_first()()                    Add_last()()
  [first & (blocked=false)] skip;     [(last=false) & (load >0)] skip;
  /*the right sensor is off*/          /*the left sensor is on*/
  {first:=false; load:=load+1};        {last:=true};
Remove_first()().                    Add_last()().
```
(d) Internal behavior of Extraction_belt

Figure 7. The behavior of Extraction_belt

comments in Figure 6-(c) guides the refinement in later design stages.

Example 2 *The Extraction_belt player*
Extraction_belt interacts with two other players: Extraction_robot and Extraction_buffer. Different from the previous example, the handshake conditions are changed not only by the interactions between the players, but also by the internal behavior of Extraction_belt. We use flag "first" to indicate whether a block is at the right-end of the belt, flag "last" to indicate whether a block is at the left-end of the belt and flag "blocked" to represent a block is blocked at the left-end of the belt. Two additional activities (Remove_first()() and Add_last()()) are given to abstract the internal behavior of Extraction_belt. Remove_first()() states that the block at the right-end can be removed, if no block is blocked at the left-end. Add_last()() specifies that a block will reach the left-end if there are blocks on the belt.

The above examples illustrate 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. Figure 8
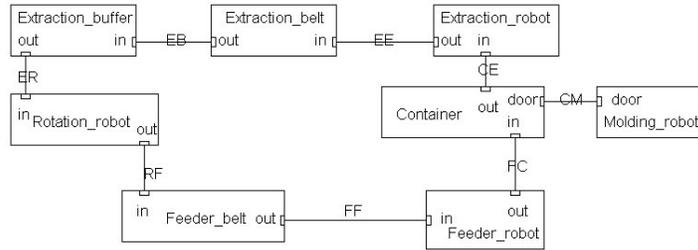
Figure 8. The C-model of the production cell system

shows the C-model of the production cell system, where the eight processes correspond to the eight players in Figure 2. 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 Extraction_Buffer.

- There is at most one block in Container.

- Feeder_robot can only push a block into Container when the door is closed and Container is empty.

As we have mentioned previously, players negotiate with each other for reliable "physical" interactions between them (as shown in Figure 4). However, these interactions are not explicitly specified in the C-model yet. In the next subsection, we incorporate these interactions into the M-model.

4.4 M-model

A mechatronic system is built by several disciplines. The software of the system glues these disciplines as a whole. In general, software engineering mainly addresses the high-level control of the system, which plans actions for the physical elements. Control engineering deals with the low-level 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 (e.g. Feeder_robot reaches Container), 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 (e.g. the refinement of Rotation_robot in Figure 9).

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. In the following, we use the Rotation_robot player as an example to illustrate the refinement from the C-model to the M-model.

Example 3 *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).*

As shown in Figure 9, 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 Rotation_robot

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

Figure 11. The behavior of the motor_low_ctl process

in the C-model. Recall the handshake mechanisms in Figure 4. The interacting behavior between players are preformed during their handshakes. For instance, after Rotation_robot receives the grant message from Feeder_belt, both of them are ready to carry out the actual "physical" interaction. As shown in Figure 10-(b), Rotation_robot turns on the magnet to pick up the block and then starts its motor to move the block to 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, Rotation_robot reverses its direction ("motor! reverse") and then moves back to the table to pick up the next block.

Since the low-level control processes (e.g. the motor_low_ctl and the magnet_low_ctl process in Figure 9) 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 11 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;).

Figure 9. The refinement of Rotation_robot in the M-model

Figure 10. The high-level control in Rotation_robot

4.5 R-model

In the previous models, the system behavior is analyzed qualitatively. To reason about quantitative properties of the system such as deadlines and throughput, or to generate the final control software, we need to incorporate timing information into the model.

The timing information for the high-level control can be specified by the quantitative timing relations between two adjacent discrete events. For instance, in Figure 10-(a), Rotation_robot turns on the magnet to pick up a block. It should start to move after the block has been elevated which takes less than 0.1 seconds in time. This timing relation can be specified by inserting a time delay statement ("delay 0.1") in front of "motor! start".

The timing information for the low-level control can be added to the model in various ways. When the continuous time behavior is not the concern of the analysis, the timing information can be added in the same way as that in the high level control. However, when the continuous time behavior is of interest, instead of adding the timing information between discrete events, control loops with the timing
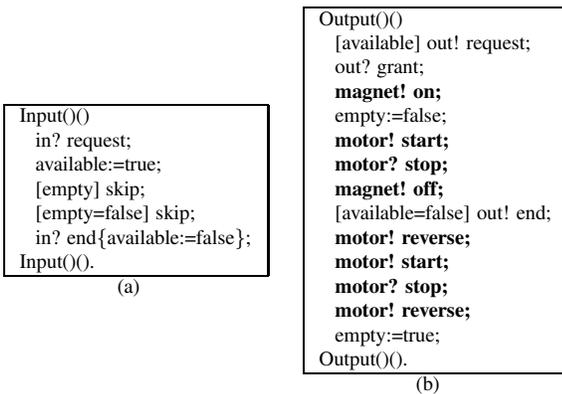
```
Discrete()()
    motor? start {rot Start; running:=true};
    [rot Active=false] motor! stop{rot Stop; running:=false};
    motor ? reverse{rot SetBackwards};
    motor? start {rot Start; running:=true};
    [rot Active=false] motor! stop{rot Stop; running:=false};
    motor ? reverse{rot SetForwards};
Discrete()().
```

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

Figure 12. The behavior of the motor_low_ctl process

information are incorporated into the model. 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.

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][5]. 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 [6]. 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 4 *The motor_low_ctl process in Figure 11 incorporates the continuous time behavior by specifying a separate activity (Continuous()()), which is in parallel with the discrete one (Discrete()()) as shown in Figure 12. 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 12. 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 9). 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 con-
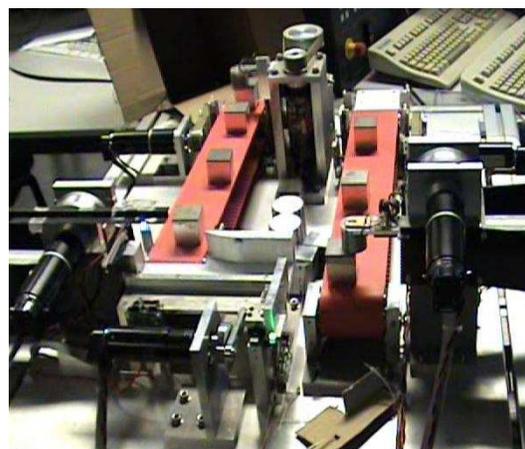

Figure 13. A snapshot of the implementation

trol (the high_ctl process) remain the same as those in the M-model.

5 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. However, for analysis convenience, the timing behavior of the system has been idealized in the model, which cannot be completely implemented in reality. For instance, actions are instantaneous in the model but are always durational in the implementation. This timing mismatch is often neglected by existing synthesis tools. Consequently, the generated implementation may exhibit unexpected behaviors. In [6], we have proposed a synthesis approach for real-time systems. Different from existing approaches, our proposed approach generates an implementation, the behavior of which can be interpreted in two time domains, namely, virtual time and physical time. The virtual time associated with an implementation is the same as the time in the model. The physical time refers to the time in the environment, which the implementation interacts with. To ensure that the implementation has almost the same be-

havior as the model in the physical time domain, the following two techniques are used during the execution 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.

For the production cell system, the implementation is generated from the R-model by using the synthesis approach proposed in [6]. Figure 13 gives a snapshot of the running system, which is controlled by the synthesized software.

6 Conclusions

Fast time to market and dynamics of the market demand an efficient design process for mechatronic 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 software design in these mechatronic systems to reduce design difficulty 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. Geilen, J. Voeten, P. van der Putten, L. van Bokhoven, and M. Stevens. Object-oriented modelling and specification using SHE. *Journal of Computer Languages*, 27, 2001.
[5] 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.
[6] J. Huang, J. Voeten, and H. Corporaal. Predictable real-time software synthesis. *To be published in journal of real-time systems*.
[7] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9 (Hard) 0-13-115007-3 (Pbk).
[8] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In A. K. G. Larsen, editor, *Proceedings of the 3rd Workshop on Computer-Aided Verification, LNCS 575*, pages 376–398, Alborg, Denmark, July 1991. Springer-Verlag.
[9] L. van Bokhoven. *Constructive Tool Design for Formal Languages from semantics to executing models*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2002.
[10] P. van der Putten and J. Voeten. *Specification of Reactive Hardware/Software Systems*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.

IEEE
COMPUTER
SOCIETY