Design and prototyping of real-time systems using CSP and CML

Rischel, Hans; Sun, Hong Yan

Published in:

Link to article, DOI:
10.1109/EMWRTS.1997.613772

Publication date:
1997

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
10.1109/EMWRTS.1997.613772
Design and Prototyping of Real-Time Systems Using CSP and CML

Hans Rischel and Hongyan Sun

Department of Information Technology
Technical University of Denmark
Building 344, DTU, 2800 Lyngby, Denmark
E-mail: rischel@it.dtu.dk and hs@it.dtu.dk
Fax: (+45) 45930074

Abstract

A procedure for systematic design of event based systems is introduced by means of the Production Cell case study. The design is documented by CSP-style processes, which allow both verification using formal techniques and also validation of a rapid prototype in the functional language CML.

1. Introduction

Notations like CSP [1] or CCS [2] provide concise notations for documenting the design of reactive or real-time systems. These notations further allow verification of properties through calculation, or model checking [3]. Yet there is a sizable gap from such specifications to executable programs needed to validate or test the design [4, 5, 6, 7].

In this paper we demonstrate how this gap is closed by CML [8], an extension of ML [9]. As shown in this paper, it is easy to get from a CSP design to an executable CML program, and the program can be interfaced to programs in other programming languages. We illustrate this idea by applying the design method for real-time systems presented in [10, 11] to a well-known example, the Production Cell [12], which has been developed by FZI in Karlsruhe [12] as a benchmark example of real-time systems development. Our CML program has been combined with the FZI simulator [12] to a working prototype.

The design method as presented in this paper consists of the following sequence of steps, each leading to a documentation with a specific form and scope.

1. System partition: Define components or subsystems for a system.
2. Interface definition: Define interface events.
3. Event structuring: Define sequencing of events.
4. Program structuring: Define functionality of the program modules.
5. Functionality check: Check for satisfaction of functional requirements.
6. Prototyping: Test a prototype program in a real or simulated environment.

In the next section, we give an overview of the Production Cell and the safety requirements. Section 3 describes the partition of the system into subsystems. Each subsystem corresponds to a physical component of the Production Cell. Section 4 defines interfaces between interacting subsystems by synchronization events. In section 5, the event structure is defined as a sequence of synchronization events by means of a CSP expression. We perform a functionality check in section 6 by applying algebraic laws of CSP. Section 7 contains some remarks about timing check (which is not formalized in this paper). In section 8, the prototype CML program is obtained from the CSP expressions. Finally, section 9 presents our conclusions.

2. The Production Cell

The production cell is an actual industrial unit in a metal processing plant in Karlsruhe. It is composed of a feed belt, an elevating rotary table, a two-armed robot, a press and a deposit belt (cf. Figure 1). In the simulated system a crane is added in order to recycle the metal blanks.

Safety requirements: Safety requirements of the production cell are classified into four groups: machine mobility must be restricted to certain limits; machine collisions must be avoided; metal blanks must not be dropped outside the safe areas; and metal blanks must not be placed on top of each other.

In the case of the elevating rotary table, for example, safety requirements include:
The elevating rotary table must not rotate clockwise if it is in the position required for delivering a blank to the robot. It must not rotate anticlockwise if it is in the position required for receiving a blank from the feed belt.

The elevating rotary table must not move down further if the table is in the position required for receiving a blank from the feed belt. It must not move up further if it is in the position required for delivering a blank to the robot.

The elevating rotary table must be in the desired position when delivering a blank to the robot or when receiving a blank from the feed belt.

The elevating rotary table receives a blank only if there is no blank on the table.

The robot moves to the position where arm 1 points to the elevating rotary table and picks up the blank. It then rotates until arm 2 points to the press, extends the arm into the press, and then unloads the forged blank from the press. Afterwards, the robot rotates until arm 2 points to the deposit belt, extends the arm to the belt and unloads the blank onto the belt.

The deposit belt conveys the blank delivered by arm 2 of the robot to the position where the travelling crane can pick up the blank.

The crane picks up the blank from the deposit belt, and transfers it to the feed belt for a new cycle of the system.

Each subsystem comprises sensors and actuators for the physical component in the subsystem plus a program for controlling these sensors and actuators.

Examining the requirements we find that the processing of a metal blank comprises two kinds of action:

- A local processing inside one subsystem, e.g. the blank is moved by the feed belt or the table, or the blank is forged in the press.

- A transfer from one subsystem to the other, e.g. the blank is conveyed from the feed belt to the table.

The first kind of action is performed completely within one subsystem while the second requires cooperation between two subsystems.

### 4. Interface Definition

Interfaces between interacting subsystems are defined by synchronization events. For example, the table subsystem with synchronization events is shown in Figure 2.

![Figure 1. The production cell](image)

![Figure 2. The table subsystem with synchronization events](image)

The table subsystem interfaces with the feed belt subsystem by the begin_fb_1 and end_fb_1 events, and with the robot subsystem by the begin_r_1 and end_r_1 events. The events are shown in Table 1.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin_fb_1</td>
<td>The feed belt begins to convey a blank.</td>
</tr>
<tr>
<td>end_fb_1</td>
<td>The feed belt ends to convey a blank.</td>
</tr>
<tr>
<td>begin_r_1</td>
<td>The robot begins to load a blank.</td>
</tr>
<tr>
<td>end_r_1</td>
<td>The robot ends to load a blank.</td>
</tr>
</tbody>
</table>

1The events begin_b_fb and end_b_fb are used to get extra blanks onto the feed belt from outside. The real system in the factory has no crane.
The table subsystem is further subdivided into TableMain, Turn and Updown programs (cf. Figure 3). The Updown and Turn programs control the vertical and horizontal movements of the table through the updown and turn controllers. The main program for the table subsystem, TableMain, synchronizes with these controllers in order to obtain the proper movement of the table. The table subsystem with local synchronization events is illustrated in Figure 3.

so the events begin.db.c, end.db.c, begin.c.fb, and end.c.fb will not be present in this system. Instead there will be events begin.db.o and end.db.o to synchronize the transfer of processed blanks out of the system, and blanks are transferred into the system via the events begin.b.fb and end.b.fb.

Table 1. Synchronization events between subsystems

<table>
<thead>
<tr>
<th>Events</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin.b.fb</td>
<td>The feed belt is ready to receive a blank</td>
</tr>
<tr>
<td>end.b.fb</td>
<td>A blank has been put on the feed belt</td>
</tr>
<tr>
<td>begin.fb.t</td>
<td>The table is empty and in the receiving position</td>
</tr>
<tr>
<td>end.fb.t</td>
<td>A blank has been conveyed to the table via the feed belt</td>
</tr>
<tr>
<td>begin.t.a1</td>
<td>The table is in the delivering position and arm 1 is ready</td>
</tr>
<tr>
<td>end.t.a1</td>
<td>Arm 1 has taken a blank from the table and the table is empty</td>
</tr>
<tr>
<td>begin.a1.p</td>
<td>The press is in the middle position</td>
</tr>
<tr>
<td>end.a1.p</td>
<td>Arm 1 has been retracted after loading a blank onto the press</td>
</tr>
<tr>
<td>begin.p.a2</td>
<td>The press is in the lower position</td>
</tr>
<tr>
<td>end.p.a2</td>
<td>Arm 2 has been retracted after unloading a forged blank from the press</td>
</tr>
<tr>
<td>begin.a2.db</td>
<td>The deposit belt is ready to receive a blank from arm 2</td>
</tr>
<tr>
<td>end.a2.db</td>
<td>Arm 2 has delivered a blank on the deposit belt</td>
</tr>
<tr>
<td>begin.db.c</td>
<td>Both the crane and the deposit belt are ready</td>
</tr>
<tr>
<td>end.db.c</td>
<td>The crane has taken a blank from the deposit belt</td>
</tr>
<tr>
<td>begin.c.fb</td>
<td>The feed belt is ready to receive a blank from the crane</td>
</tr>
<tr>
<td>end.c.fb</td>
<td>The crane has transported a blank onto the feed belt</td>
</tr>
</tbody>
</table>

Table 2. Synchronization events within the table subsystem

<table>
<thead>
<tr>
<th>Events</th>
<th>Data</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin.turn</td>
<td>integer</td>
<td>The table is requested to move to a horizontal position</td>
</tr>
<tr>
<td>end.turn</td>
<td></td>
<td>The table has rotated to the desired position</td>
</tr>
<tr>
<td>begin.updown</td>
<td>up/down</td>
<td>The table is requested to move to a vertical position</td>
</tr>
<tr>
<td>end.updown</td>
<td></td>
<td>The table has moved to the desired position</td>
</tr>
</tbody>
</table>

The behaviour of each subsystem is controlled by a group of synchronization events. The subsystem restricts the occurrence of these events in order to meet both functional and safety requirements of the system. For example, synchronization events in the main program of the table subsystem are structured in a CSP expression as follows:
TableMain =
(begin.turn(0) -> begin.updown(down) ->
end.turn -> end.updown ->
begin.fbt -> end.fbt ->
begin.turn(45) -> begin.updown(up) ->
end.turn -> end.updown ->
begin.tal -> end.tal1 -> TableMain).

The first two lines in this expression describe the movement of the table to the receiving position by commands to the turn and the updown controllers. The third line describes the interfaces with the feed belt to agree on conveying a metal blank from the feed belt to the table. The fourth and fifth lines describe the movement of the table to the delivering position by commands to the turn and the updown controllers. The last line describes the interfaces with the robot to allow the blank to be picked up by arm 1. Whereupon this the whole sequence is repeated.

Apparently, TableMain has a simple sequential structure as events happen in a pre-specified order. But the event structure for the robot subsystem will show branching corresponding to a choice between events. For example, when arm 2 unloads a blank from the press and arm 1 is empty, the robot can either rotate so that arm 1 can first pick up a blank, then deliver it onto the deposit belt, or vice versa, which depends on which synchronization event, begin.tal or begin.a2.db, is first satisfied. This kind of choice between events is expressed in CSP by the operator “|”.

6. Functionality Check

The event expressions are processes in the sense of CSP (cf. [1]), so the algebraic laws of CSP can be applied to prove properties of the programs.

For example, one of the safety requirements for the elevating rotary table is R: the elevating rotary table must be in the receiving position when a blank is conveyed from the feed belt.

To check this safety requirement, we add an observer process OBS to the system. Once the safety requirement R is violated, OBS should indicate a failure by allowing the failure event †. We also include the events in Table 3 in the table and the feed belt subsystems for the synchronization between OBS and the subsystem in question.

<table>
<thead>
<tr>
<th>Events</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe.t</td>
<td>The table is in the receiving position</td>
</tr>
<tr>
<td>unsafe.t</td>
<td>The table may be outside the receiving position</td>
</tr>
<tr>
<td>safe.fbt</td>
<td>The feed belt may drop a blank onto the table</td>
</tr>
<tr>
<td>unsafe.fbt</td>
<td>The feed belt will not drop a blank onto the table</td>
</tr>
</tbody>
</table>

Thus, the main process for the table subsystem, TABLE, is extended by including the safe.t and unsafe.t events:

Table =
(begin.turn(0) -> begin.updown(down) ->
end.turn -> end.updown ->
safe.t -> begin.fbt -> end.fbt ->
begin.turn(45) -> unsafe.t ->
begin.updown(up) -> end.turn ->
end.updown -> begin.tal1 ->
end.tal1 -> TABLE).

The table is in the safe position when it has been turned to angle 0 and moved down to the position for receiving a blank from the feed belt, so the event safe.t is inserted after the end.turn and end.updown events. The table becomes unsafe as soon as any movement has been initiated, so the event unsafe.t is inserted just after the begin.turn event. The events safe.fbt and unsafe.fbt are similarly inserted in the program of the feed belt subsystem.

An observer process OBS with the alphabet αOBS = {safe.t, unsafe.t, safe.fbt, unsafe.fbt, †} is given by the following expressions:

OBS = (safe.t -> OBS | safe.fbt -> OBS |
unsafe.t -> A | unsafe.fbt -> B).

A = (safe.t -> OBS | safe.fbt -> A |
unsafe.t -> A | unsafe.fbt -> †).

B = (safe.t -> B | safe.fbt -> OBS |
unsafe.t -> † | unsafe.fbt -> B).

The observer process is always ready to participate in any safe or unsafe event, and it becomes ready for the † event if a dangerous situation should occur. Hence, if we can prove that tr{†} = {} for all tr \in traces(TABLE||FB||OBS), then the satisfaction of R is proved. Here FB denotes the main process of the feed belt subsystem.

The proof is given in the appendix. It is done by using the laws of CSP only. The proof could probably be automatized by using the FDR tool (cf. [3]).

7. Timing

Timing requirements of an individual component arise in two ways:

- when distributing a global timing requirement over components
- when implementing a functional requirement by a timing condition

For example, the requirement “TableMain should send the begin.turn command at most 100ms after the end.fbt command has been received” can be part of implementing the global timing requirement: “the production cell should
produce 500 plates per hour”. And the requirement “Turn should send the table.stop.turn command at most 10ms after the final table angle value has been received” can be part of implementing the functional requirement “inaccuracy in the table angle in the position for receiving a blank from the feed belt must not exceed 5 degrees”.

The notation in this paper does not include the formalization and verification of timing requirements, but it seems possible to extend the notation by using suitable concept from the recent book [4, 5] on mathematical methods for real-time systems.

8. Prototyping

The concurrent ML language (CML) is an extension of the standard ML (SML) programming language [9, 13], which is a functional programming language with a flexible type system and a powerful expression language where expressions may denote composite values of an arbitrary type. It provides synchronous communication over typed channels as the basic communication and synchronization mechanism. Basic channel operations in CML are listed in Table 4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Type</th>
<th>Legend</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>unit→'la chan</td>
<td>Create a new channel</td>
</tr>
<tr>
<td>send</td>
<td>'a chan→'a</td>
<td>Send a synchronous message to a channel</td>
</tr>
<tr>
<td>accept</td>
<td>'a chan→'a</td>
<td>Read a synchronous message from a channel</td>
</tr>
</tbody>
</table>

Table 4. Basic channel operations in CML

The functions send and accept are used in pairs, i.e. if one process uses send, the other process must use accept to synchronize the communication over the channel. If one process has a parameter to pass to the other, it should use send. Both processes will wait until the communication has taken place. The language allows a process to make a choice, synchronizing on the first arriving communication over a set of channels. It also allows a process to test whether a communication is pending on a channel.

The communication between subsystems (cf. Table 1) is implemented by means of channels. The same is the case for the local synchronizations inside a subsystem (cf. Figure 4). It is then straightforward to derive the CML programs for the CSP processes, as the recursive definition of CSP process expressions can be preserved in the CML program. For the table subsystem we hence get the Table program as shown in Figure 4. It contains a main program TableMain, and programs Updown and Turn for the updown and turn controllers.

For the event structure with branching, e.g. the robot subsystem, the choice between two events is implemented in CML by the select operation.
The main program *ProductionCell* is composed of seven subprograms, *FeedBelt*, *Table*, *Robot*, *Press*, *DepositBelt*, *Crane* and *Blank*. The subprogram *Blank* is used to put extra blanks onto the feed belt in order to start the system during the simulation. The remaining six subprograms implement the subsystems. These main components are executed as parallel programs.

The local control programs, e.g. *Updown* and *Turn* of the *Table* program, are designed with a unified interface consisting of a pair of synchronizations *(begin\_x, end\_x)* with the higher-level program, e.g. *TableMain*. Actually, these controllers have different interfaces to the physical environment, but these differences are local to the individual program for each controller and not visible from the outside.

The CML program for the *Production cell* has been exercised with the FZI simulator. The simulator has two significant functions. One is to simulate physical components including internal controllers of each component. The other is to visualize the simulated movements of each physical component during the CML program execution. This requires some extension of the simulator such that the interfaces are expressed in terms of CML channels. The running system including the simulator is composed of two UNIX processes connected by UNIX pipes as shown in Figure 5.

The communication over the UNIX pipes uses an ASCII protocol which is part of the FZI system. The interface program (programmed in CML) performs the multiplexing/demultiplexing into a set of CML channels. The control program could in principle be used for controlling a real, physical plant by connecting the CML channels directly to I/O driver programs for peripherals connected to the physical units in the plant.

Figure 6 is a screen dump of the working window of the FZI simulator controlled by the CML program.

The program for each subsystem can also be tested separately with the simulator. Testing e.g. the *table* subsystem requires a small CML program to simulate the interfaces to the other components on the channels *(begin\_fb, end\_fb, begin\_tal, end\_tal)* and *(begin\_a1, end\_a1)*, and the test can be executed by letting this program interact with the operator via the terminal.

9. Conclusion

We have shown, in this paper, how to apply a design method with a particular case study *Production cell*. The method itself is engineering-oriented, and it is based on a sound theoretical foundation. The use of CML for programming concurrent systems in practice has shown a satisfactory result as we have obtained a running prototype by combining our program with the FZI simulator. Each synchronization event, which is the key element in our method, can be directly transferred into a CML channel, and the event expressions are easily converted to CML functions. The resulting program satisfies the functional and safety requirements of the system as shown by proofs and by simulation results.

Acknowledgment

We would like to thank Professor Anders P. Ravn for many helpful discussions and valuable suggestions.

References


We first select $S' = \alpha TABLE \cup \alpha FB - \alpha OBS - (\alpha TABLE \cap \alpha FB)$.

According to law L1 in [1] (3.5.3) it will suffice to find a set $S$ of events such that $\{\} \not\subseteq S$ and such that $tr[\{\}] = \{\}$ for all $tr \in traces((TABLE||FB||OBS) \setminus S)$.

So we can use law L6 in [1] (3.5.1), thus,

$$(TABLE||FB||OBS) \setminus S' = (TABLE||FB||OBS) \setminus (TABLE||FB||OBS) \setminus S$$

and

$$(TABLE||FB) \setminus S' = (TABLE||FB) \setminus (TABLE||FB) \setminus S$$

while

$$(TABLE \setminus S')(TABLE \setminus S') = (safe.f \rightarrow begin.fb.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow end.fb.t \rightarrow unsafe.t \rightarrow (TABLE \setminus S'))((TABLE \setminus S'))$$

By law L12 in [1] (3.5.1), $OBS \setminus S' = OBS$.

We then select $S = S' \cup (\alpha TABLE \cap \alpha FB)$, thus

$$(TABLE||FB||OBS) \setminus S = (TABLE||FB) \setminus S \setminus OBS \setminus S$$

and

$$(TABLE||FB) \setminus S = (safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow unsafe.t \rightarrow (TABLE||FB) \setminus S)$$

again, $OBS \setminus S = OBS$.

Let $TFB = (TABLE||FB) \setminus S$, then

$$TFB||OBS = (safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow unsafe.t \rightarrow TFB)||OBS$$

and

$$(unsafe.t; TFB)||OBS = (unsafe.t \rightarrow safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow (unsafe.t \rightarrow TFB)||OBS)$$

that is

$$(unsafe.t; TFB)||OBS = \mu X.(unsafe.t \rightarrow safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow X).$$

Process $TFB||OBS$ can hence be reformulated as two sequential processes: $TFB||OBS = P; Q$,

where

$$P = (safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow SKIP).$$

$$Q = \mu X.(unsafe.t \rightarrow safe.t \rightarrow unsafe.fb \rightarrow safe.fb \rightarrow X).$$

By law L1 in [1] (5.3.1),

$$traces(P; Q) = \{s; t \mid s \in traces(P) \land t \in traces(Q)\}.$$