DTU Library

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

Rischel, Hans; Sun, Hong Yan

Published in:
Real-Time Systems, 1997. Proceedings., Ninth Euromicro Workshop on

Link to article, DOI:
10.1109/EMWRTS.1997.613772

Publication date:
1997

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

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:

0-8186-8034-2/97 $10.00 © 1997 IEEE

- 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.

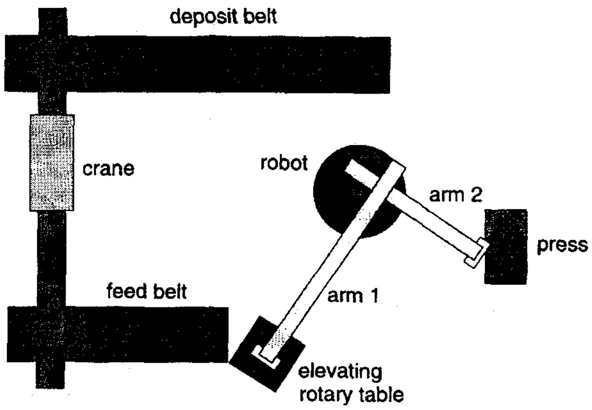


Figure 1. The production cell

3. System Partition

It seems reasonable to partition the system into subsystems corresponding to the physical components illustrated in Figure 1. Each subsystem fulfils a specific task during the metal blank processing:

The **feed belt** conveys a metal blank to the elevating rotary table when the table is in the position for receiving a metal blank from the feed belt.

The **table** (elevating rotary table) performs an upward movement and a anticlockwise rotation in order to transfer the blank to the desired position where the robot can pick it up.

The **press** moves its lower plate upwards to the position where arm 1 can load the blank. After the robot loads the blank onto the press, the press forges the blank and then moves the lower plate downwards to the position where the blank can be unloaded by arm 2 of the robot.

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.

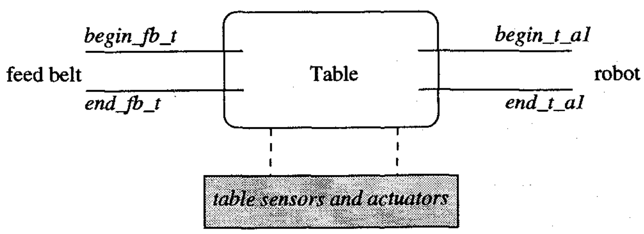


Figure 2. The table subsystem with synchronization events

The *table* subsystem interfaces with the *feed belt* subsystem by the *begin_fb_t* and *end_fb_t* events, and with the *robot* subsystem by the *begin_t_al* and *end_t_al* events. The events are shown in Table 1[1].

[1]The 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,

Events	Legend
begin_b_fb	The feed belt is ready to receive a blank
end_b_fb	A blank has been put on the feed belt
begin_fb_t	The table is empty and in the receiving position
end_fb_t	A blank has been conveyed to the table via the feed belt
begin_t_a1	The table is in the delivering position and arm 1 is ready
end_t_a1	Arm 1 has taken a blank from the table and the table is empty
begin_a1_p	The press is in the middle position
end_a1_p	Arm 1 has been retracted after loading a blank onto the press
begin_p_a2	The press is in the lower position
end_p_a2	Arm 2 has been retracted after unloading a forged blank from the press
begin_a2_db	The deposit belt is ready to receive a blank from arm 2
end_a2_db	Arm 2 has delivered a blank on the deposit belt
begin_db_c	Both the crane and the deposit belt are ready
end_db_c	The crane has taken a blank from the deposit belt
begin_c_fb	The feed belt is ready to receive a blank from the crane
end_c_fb	The crane has transported a blank onto the feed belt

Table 1. Synchronization events between subsystems

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*.

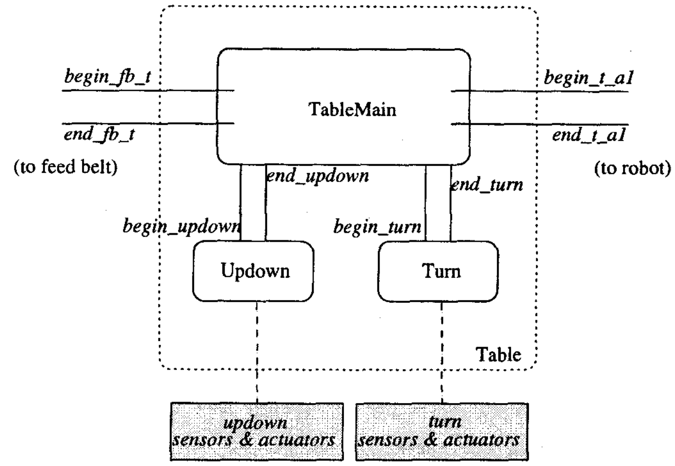


Figure 3. The table subsystem with local synchronization events

Local interfaces within the *table* subsystem are defined by four synchronization events: *begin_turn*, *end_turn*, *begin_updown*, and *end_updown*, which are shown in Table 2.

Events	Data	Legend
begin_turn	*integer*	The table is requested to move to a horizontal position
end_turn		The table has rotated to the desired position
begin_updown	***up*|*down***	The table is requested to move to a vertical position
end_updown		The table has moved to the desired position

Table 2. Synchronization events within the table subsystem

The *begin_turn* and *begin_updown* events contain corresponding data values indicating the desired horizontal and vertical end position for the table.

5. Event Structuring

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:

$$\begin{aligned}&TableMain = \\&(begin\_turn(0) \to begin\_updown(down) \to \\&\ end\_turn \to end\_updown \to \\&\ begin\_fb\_t \to end\_fb\_t \to \\&\ begin\_turn(45) \to begin\_updown(up) \to \\&\ end\_turn \to end\_updown \to \\&\ begin\_t\_a1 \to end\_t\_a1 \to TableMain).\end{aligned}$$

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_t_a1* 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 $\mathfrak{R}$: *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 $\mathfrak{R}$ 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.

Events	Legend
safe_t	The table is in the receiving position
unsafe_t	The table may be outside the receiving position
safe_fb	The feed belt may drop a blank onto the table
unsafe_fb	The feed belt will not drop a blank onto the table

Table 3. Events for OBS observation

Thus, the main process for the *table* subsystem, *TABLE*, is extended by including the *safe_t* and *unsafe_t* events:

$$\begin{aligned}&TABLE = \\&(begin\_turn(0) \to begin\_updown(down) \to \\&\ end\_turn \to end\_updown \to \\&\ safe\_t \to begin\_fb\_t \to end\_fb\_t \to \\&\ begin\_turn(45) \to unsafe\_t \to \\&\ begin\_updown(up) \to end\_turn \to \\&\ end\_updown \to begin\_t\_a1 \to \\&\ end\_t\_a1 \to TABLE).\end{aligned}$$

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_fb* and *unsafe_fb* are similarly inserted in the program of the *feed belt* subsystem.

An observer process OBS with the alphabet $\alpha OBS = \{safe\_t, unsafe\_t, safe\_fb, unsafe\_fb, \dagger\}$ is given by the following expressions:

$$\begin{aligned}OBS &= (safe\_t \to OBS \mid safe\_fb \to OBS \\&\quad \mid unsafe\_t \to A \mid unsafe\_fb \to B). \\A &= (saft\_t \to OBS \mid safe\_fb \to A \\&\quad \mid unsafe\_t \to A \mid unsafe\_fb \to \dagger). \\B &= (safe\_t \to B \mid safe\_fb \to OBS \\&\quad \mid unsafe\_t \to \dagger \mid unsafe\_fb \to B).\end{aligned}$$

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 $tr \restriction \{\dagger\} = \langle\rangle$ for all $tr \in traces(TABLE \| FB \| OBS)$, then the satisfaction of $\mathfrak{R}$ 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_fb_t* 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

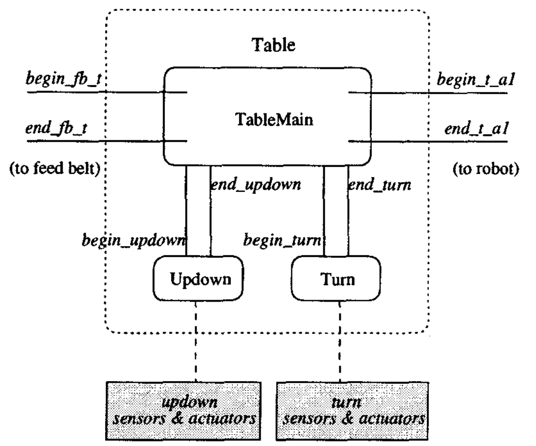
The concurrent ML language (CML) is an extension of the standard ML (SML) programming language [9, 13], which is a functional programming language with a flexible type system and a powerful expression language where expressions may denote composite values of an arbitrary type. It provides synchronous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

Operation	Type	Legend
channel	*unit* $\rightarrow$ *'1a chan*	Create a new channel
send	*'a chan* $*$ *'a* $\rightarrow$ *unit*	Send a synchronous message to a channel
accept	*'a chan* $\rightarrow$ *'a*	Read a synchronous message from a channel

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[2], 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.




```
fun Table() =
  let datatype updown  = up | down;
    val begin_updown = updown chan;
    val end_updown   = unit chan;
    val begin_turn   = int chan;
    val end_turn     = unit chan;
    fun TableMain () =
        (send (begin_turn, 0)
        ;send (begin_updown, down)
        ;accept (end_turn)
        ;accept (end_updown);
        ;send (begin_fb_t)
        ;accept (end_fb_t)
        ;send (begin_updown, up)
        ;send (begin_turn, 45)
        ;accept (end_updown)
        ;accept (end_turn)
        ;send (begin_t_al)
        ;accept (end_t_al)
        ;TableMain (););
    fun Updown () = ... ;
    fun Turn () = ... ;
    ...
  end;
```

Figure 4. A sketch of the program structure of the table subsystem

For the event structure with branching, e.g. the *robot* subsystem, the choice between two events is implemented in CML by the *select* operation.

[2]In the *Production Cell* example, the values to be transmitted over the channels are simple constants. Other systems may include extensive computations and local state variables, but that can also be handled by the method.

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.

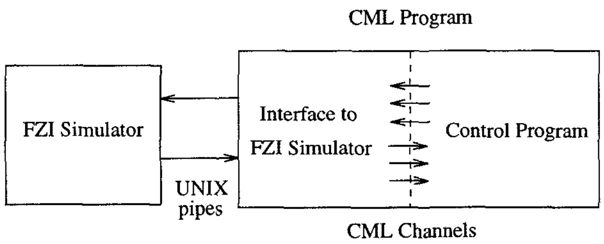


Figure 5. FZI simulator and CML control program

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[3]. 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.

[3]Each implementation of a control program for the FZI production cell has its individual interface program for transforming between the FZI ASCII protocol and the communication primitives in the programming language used for the control program.

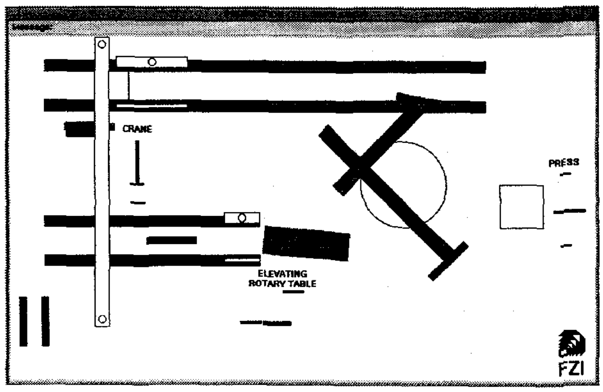


Figure 6. Working window of the FZI simulator

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_t, end_fb_t, begin_t_al* and *end_t_al*, 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

[1] C. A. R. Hoare. *Communicating Sequential Processes*, Computer Science Series, Prentice Hall, 1985.

[2] Robin Milner. *Communication and Concurrency*, Computer Science Series, Prentice Hall, 1989.

[3] Formal Systems Ltd. *Failures Divergence Refinement - User Manual and Tutorial*, Version 1.4, Formal Systems (Europe) Ltd., 1994.

[4] M. Joseph (Ed). *Real-Time Systems: Specification, Verification and Analysis*, Computer Science Series, Prentice Hall, 1996.

[5] C. Heitmeyer and D. Mandrioli (Eds). *Formal Techniques In Real-Time Systems*, Trends in Software-Engineering Series, Wiley, 1996.

[6] J. Hooman and J. Vain. An Integrated Technique for Developing Real-Time Systems, *Proceedings of the Seventh Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 14-16, 1995, pp236-243.

[7] N. Nissanke. Towards Refinement in Realtime Programming, *Proceedings of the Seventh Euromicro Workshop on Real-Time Systems*, Odense, Denmark, June 14-16, 1995, pp244-251.

[8] John H. Reppy. *Concurrent Programming with Events - The Concurrent ML Manual*, Version 0.9.8, AT&T Bell Lab., February 1, 1993.

[9] L. C. Paulson. *ML for the Working Programmer*, Cambridge University Press, 1991.

[10] Anders P. Ravn, Hans Rischel and Hans Henrik Løvengreen. A Design Method for Embedded Software Systems, *BIT 28*, 1988, pp427-438.

[11] Hans Henrik Løvengreen, Anders P. Ravn and Hans Rischel. Design of Embedded, Real-time Systems: Developing a Method for Practical Software Engineering, *COMPEURO 90*, Tel-Aviv, Israel May 7-9, 1990.

[12] Claus Lewerentz and Thomas Lindner (Eds). *Formal Development of Reactive Systems: Case Study Production Cell*, LNCS 891, Springer-Verlag, 1995.

[13] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*, The MIT Press, 1990.

Appendix

Proof.

We have to prove that $tr \restriction \{\dagger\} = \langle\rangle$
for all $tr \in traces((TABLE \| FB \| OBS) \setminus S)$.

According to law L1 in [1] (3.5.3) it will suffice to find a set S of events such that $\{\dagger\} \notin S$ and such that $tr \restriction \{\dagger\} = \langle\rangle$
for all $tr \in traces((TABLE \| FB \| OBS) \setminus S)$.

We first select

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

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

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

and

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

while

$$(TABLE \setminus S') \| (FB \setminus S') = \\ (safe\_t \to begin\_fb\_t \to unsafe\_fb \to \\ safe\_fb \to end\_fb\_t \to unsafe\_t \to \\ (TABLE \setminus S')) \| (FB \setminus S')$$

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

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

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

and

$$(TABLE \| FB) \setminus S = \\ (safe\_t \to unsafe\_fb \to safe\_fb \to \\ unsafe\_t \to (TABLE \| FB) \setminus S)$$

again, $OBS \setminus S = OBS$.

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

$$TFB \| OBS = \\ (safe\_t \to unsafe\_fb \to safe\_fb \to \\ (unsafe\_t \to TFB)) \| OBS$$

and

$$(unsafe\_t; TFB) \| OBS = \\ (unsafe\_t \to safe\_t \to unsafe\_fb \to \\ safe\_fb \to (unsafe\_t \to TFB)) \| OBS$$

that is

$$(unsafe\_t; TFB) \| OBS = \\ \mu X.(unsafe\_t \to safe\_t \to \\ unsafe\_fb \to safe\_fb \to X).$$

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

$$P = (safe\_t \to unsafe\_fb \to safe\_fb \to SKIP).$$
$$Q = \mu X.(unsafe\_t \to safe\_t \to \\ unsafe\_fb \to safe\_fb \to X).$$

By law L1 in [1] (5.3.1),

$$traces(P; Q) = \{s; t | s \in traces(P) \wedge t \in traces(Q)\}.$$

According to law L5 in [1] (1.8.1) and by analogy with X2 in [1] (1.8.1),

$$traces(P) = \{s | s \leq \langle safe\_t, unsafe\_fb, safe\_fb, \surd\rangle\}.$$
$$traces(Q) = \\ \bigcup_{n \geq 0} \{t | s \leq \langle unsafe\_t, safe\_t, unsafe\_fb, safe\_fb\rangle^n\}.$$

Apparently, $tr \restriction \{\dagger\} = \langle\rangle$ for all $tr \in traces(P; Q)$, i.e.
$tr \restriction \{\dagger\} = \langle\rangle$
for all $tr \in traces((TABLE \| FB \| OBS) \setminus S)$.

We, therefore, conclude that
$tr \restriction \{\dagger\} = \langle\rangle$ for all $tr \in traces(TABLE \| FB \| OBS)$
as $\{\dagger\} \notin S$.

Thus the satisfaction of $\Re$ is proved. $\square$