Behaviour Analysis and Safety Conditions: a Case Study in CML

Hanne Riis Nielson Torben Amtoft Flemming Nielson

Computer Science Department, Aarhus University, Denmark
e-mail: {hrn,tamtoft,fn}@daimi.aau.dk

Abstract. We describe a case study where novel program analysis technology has been used to pinpoint a subtle bug in a formally developed control program for an embedded system. The main technology amounts to first defining a process algebra (called behaviours) suited to the programming language used (in our case CML) and secondly to devise an annotated type and effect system for extracting behaviours from programs in a such a manner that an automatic inference algorithm can be developed. The case study is a control program developed for the "Karlsruhe Production Cell" and our analysis of the behaviours shows that one of the safety conditions fails to hold.

Keywords. Embedded systems, formal program development, program analysis.

1 Introduction

There are several approaches for how to close the gap between the specification of a system and its actual realisation as a program in some programming language. Different procedures for systematic design have been developed with the goal of reducing the likelihood of introducing errors, and concise notations have been introduced for documenting and reasoning about systems.

Unfortunately, a system may have been developed using formal methods but still have bugs. Advanced proof techniques may have been used to show that the specification fulfils certain safety and liveness properties, but there is always the risk that the formalisation does not fully correspond to the informal description (or even a formal description in another framework) and that the code written does not fully correspond to the specification. Clearly the risk of such unfortunate scenarios gets smaller the more care is taken in the development of the system but we believe that it is not feasible to completely eliminate the risk. Indeed there always is the risk of human mistake (like using a previous incorrect version of the system instead of the current correct version) and of malicious behaviour (a subcontractor cutting corners to increase profit).

While formal methods clearly are very useful for increasing our confidence in the system, it would seem that more is needed. In this paper we demonstrate that

technology from program analysis can be invaluable in spotting some of the subtle bugs that may have survived the careful use of formal methods. Traditionally, program analysis has been used in optimising compilers but due to their ability to analyse programs automatically and systematically we claim that they also have an important role to play in program validation. Although the kind of properties of interest in program validation may differ from those of interest in optimising compilers, we demonstrate in this paper that recent developments have paved the way for adapting program analysis to the new application domain.

Background. In [7, 8] we present an annotated type system for extracting the communication topology of programs written in a subset of CML [9]. We introduce a formalism of behaviours, a process algebra like CCS or CSP but tailored to the characteristics of CML. The traditional type system for CML is then extended such that it determines behaviours of expressions as well as their types. Both CML and the behaviours are equipped with a small-step operational semantics and a key theoretical result is a subject reduction result ensuring that whenever the CML program engages in a communication, then also the behaviour will be able to do so. This means that safety results obtained by analysing the behaviour also apply to the original CML program.

In [1] we develop an algorithm for type and behaviour reconstruction. The development is sufficiently general that (1) the behaviours contain causality information, (2) ML-like polymorphism is supported, and (3) the algorithm is sound as well as complete with respect to the annotated type system. These properties are crucial for the application described in the present paper. The causality of the various operations is often an integral part of safety conditions for systems; without causal behaviours one can only validate rather few properties of interest. Polymorphism is important when analysing generic programs; without polymorphism (or perhaps polyvariance) one will need to merge information from different function calls and this may make it impossible to validate many interesting properties. The soundness result ensures that the behaviours obtained by the algorithm are correct with respect to the semantics of the program and the completeness result ensures that the behaviours are as precise as is possible according to the annotated type system; it should be obvious that these are crucial properties as well.

Having established the theoretical foundations [1] we have implemented a prototype for extracting behaviours from programs [2]. The present version is able to deal with a fairly large subset of CML and provides the basis for the experiments reported here.

Accomplishments. We study a CML program for the well-known "Production Cell" [4] developed by FZI in Karlsruhe as a benchmark for the development of verified software for embedded systems. The CML program used has been developed using systematic design methods: its functionality has been specified in CSP and many of its safety conditions have been formally verified [10]. Fur-

thermore, it has been combined with the FZI simulator to a working prototype that has subsequently been tested.

None the less, our program analysis reveals that the program does *not* fulfil all of its safety conditions. Our experiments show that the program makes certain assumptions about the initial configuration of the system – a bug that has escaped the formal verification. Furthermore, it turns out that the simulator makes similar assumptions about the initial configuration so that this particular bug will never turn up during testing. We should stress that we do not mean to criticise neither the formal development nor the verification methods nor the programmers. We merely see it as an illustration of a typical problem in the development of complex software systems as was alluded to above.

We believe that the results of our case study presents convincing arguments for also using novel program analysis techniques when validating safety conditions of embedded systems. Although we have been able to validate many of the safety conditions of interest, and to find one that does not hold, there is room for extending our techniques because some of the safety conditions require information not presently included in the behaviours.

Overview. In Section 2 we give a brief introduction to the basic primitives of CML and we present a fragment of the program used in the case study. Then in Section 3 we introduce the behaviours and sketch some of the central rules for how to obtain behaviours from a CML program. In Section 4 we examine three of the safety conditions of the Production Cell and in Section 5 we discuss some further enhancements of our techniques. Finally, Section 6 contains the concluding remarks.

2 The case study

The Production Cell is designed to process metal blanks in a press [4]; its various components are shown from above on Figure 1 which is a picture from the FZI simulator. The work pieces (metal blanks) enter the system on the feed belt (the bottom one on Figure 1) and are then transfered one at a time to a rotating table; the table is then lifted and rotated such that one of the two robot arms can take the work piece and place it in the press. After processing the work piece, the other robot arm will take it out of the press and deliver it to a deposit belt (the top one on Figure 1). For testing purposes a crane has been added to move the work pieces from the deposit belt back to the feed belt.

We shall concentrate on just one of these entities, namely the rotating table. The table can be in one of two vertical positions and it can be rotated clockwise as well as counterclockwise. The following safety conditions have been supplied for the table:

1: The table must not be moved downward if it is in its lower position, and it must not be moved upward if it is in its upper position.

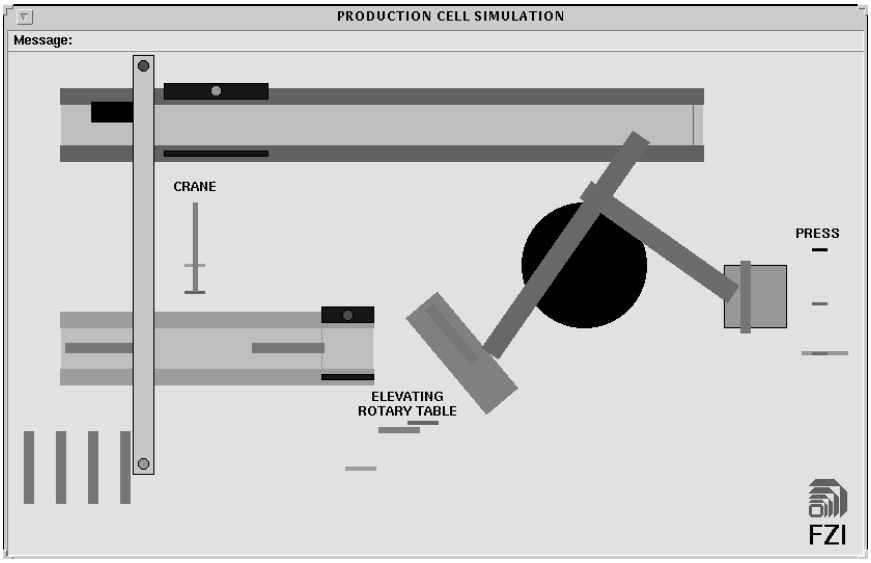

Fig. 1. The Karlsruhe Production Cell.

2: The table must not be rotated clockwise if it is in the position required for transferring work pieces to the robot, and it must not be rotated counter-clockwise if it is in the position to receive work pieces from the feed belt.
3: There can only be one work piece at the table at any time.

The program. CML [9] is an extension of the higher-order functional language SML [5] with constructs for communication. Processes and channels can be created dynamically using the constructs `spawn` and `channel`; the constructs `send` and `accept` are available for synchronous communication. Functions as well as channels are first class values and so are events: an event is a potential communication created by one of the constructs `transmit` and `receive`. There is also an explicit synchronisation operation `sync` so the construct `send(ch,v)` is equivalent to `sync(transmit(ch,v))` and similarly `accept(ch)` is equivalent to `sync(receive(ch))`. Events can be manipulated using the construct `wrap`; this corresponds to a kind of speculative post-processing of an event in that it will only take effect if and when the event is synchronised. Finally, we shall mention the construct `choose` which can be used to choose one of several events.

The CML program for the Production Cell consists of 7 processes. They communicate with the simulator using 63 channels and they communicate internally using 16 channels. The part of the program controlling the movements of the table is shown in Figure 2. It uses the following channels for communicating with the simulator:

```
(* actuator channels *)
val table_left     = channel(): unit chan;
val table_stop_h   = channel(): unit chan;
val table_right    = channel(): unit chan;
val table_upward   = channel(): unit chan;
val table_stop_v   = channel(): unit chan;
val table_downward = channel(): unit chan;

(* sensor channels *)
val table_is_bottom     = channel(): unit chan;
val table_is_not_bottom = channel(): unit chan;
val table_is_top        = channel(): unit chan;
val table_is_not_top    = channel(): unit chan;
val table_angle         = channel(): int  chan;
val new_table_angle     = channel(): unit chan;
```

Internally, the table synchronises its movements with the feed belt and the robot and for this it uses the following channels:

```
val belt1_transmit_ready = channel(): unit chan;
val belt1_transmit_done  = channel(): unit chan;

val table_transmit_ready = channel(): unit chan;
val table_transmit_done  = channel(): unit chan;
```

We shall not explain the program in detail here; some of the points will naturally be dealt with when we come to discussing aspects of its behaviour.

3 Behaviours

The safety requirements imposed on the Production Cell are to a large extent concerned with the order in which the communications are performed. This is exactly the kind of information that is available in the behaviours. The behaviours are terms of a process calculus designed to match the structure of CML. The basic behaviours are:

- ϵ is the behaviour of a program that does not create any channels or processes and that is not involved in any communication;
- t CHAN r is the behaviour of a program that creates a channel that can be used to communicate values of type t and where the channel belongs to the region r (a region is an indication of where in the program the channel has been created);
- FORK b is the behaviour for a program that spawns a new process that will behave as described by the behaviour b;
- $r!t$ is the behaviour of a program that sends a value of type t on one of the channels created in the region r; and

```
fun table () =
let
   fun clockwise (a) =                 (*rotate clockwise until degree a*)
       let val x = accept(table_angle)
       in (send(table_right,());
           while (accept(new_table_angle); accept(table_angle)) < a
           do ();
           send(table_stop_h,()) )
       end;

   fun counterclockwise (a) =   (*rotate counterclockwise until degree a*)
       let val x = accept(table_angle)
       in (send(table_left,());
           while (accept(new_table_angle); accept(table_angle)) > a
           do ();
           send(table_stop_h,()) )
       end;

   fun main () =
       (accept(belt1_transmit_ready); accept(belt1_transmit_done);
        clockwise(50);
        send(table_upward,());
        accept(table_is_top);
        send(table_stop_v,());
        send(table_transmit_ready,()); send(table_transmit_done,());
        send(table_downward,());
        accept(table_is_bottom);
        send(table_stop_v,());
        counterclockwise(0);
        main())
in
   spawn(fn () => main())
end;
```

Fig. 2. CML program for the table.

– $r?t$ is the behaviour of a program that receives a value of type t on one of the channels created in the region r.

The basic behaviours can then be combined using sequencing (expressed by ';') and choice (expressed by '+') and they can be recursively defined.

As an example consider the following behaviours:

$B_c =$ `{table_angle}?int;{table_right}!unit;` B_1 `;{table_stop_h}!unit`

$B_1 =$ `{new_table_angle}?unit;{table_angle}?int;(`ϵ `+` B_1`)`

The behaviour B_c expresses that first there will be a communication on the channel `table_angle` (obtaining the current angle of the table) and next there will

be a communication on the channel `table_right` (starting a clockwise rotation of the table). Then the behaviour of B_1 will be executed and finally there will be a communication on the channel `table_stop_h` (stopping the rotation). The behaviour B_1 is recursive: first there will be a communication over the channel `new_table_angle` (indicating that the angle has changed) and subsequently there is a communication on the channel `table_angle` (to obtain the new angle). After that the program may exit (the angle has the required value) or it may repeat the behaviour of B_1 (still waiting for the angle to get the required value).

It turns out that B_c is the behaviour corresponding to the body of the function `clockwise` of Figure 2. Comparing the code for the function with the behaviour above shows that we have recorded which communications take place and in which order, but we have ignored all values and tests. So while the behaviour retains the overall control structure of the code, it loses those details of tests that determine which branch is taken in conditionals (as e.g. that the clockwise rotation of the table is stopped at the angle given as argument to the function).

Construction of behaviours. The behaviours are extracted from the CML program by an extension of the standard polymorphic type system. The idea is that each of the concurrency primitives when supplied with the appropriate parameters gives rise to one of the basic behaviours, and the composite expressions will tell how these behaviours are combined into larger behaviours. A function may require some arguments in order to exhibit its behaviour and an event may need to be synchronised in order to exhibit its behaviour, and to capture this we shall annotate the types with behaviour information. So a function may have the type $t_1 \rightarrow^b t_2$ meaning that it takes an argument of type t_1, gives a result of type t_2 and in doing so it will perform communications as described by the behaviour b. Similarly, an event may have the type t `event` b meaning that when synchronised it will give rise to a value of type t and in doing so it will perform communications as described by b. The following specifies the annotated types of some of the primitive operations:

`send`:	$(t$ `chan` $r) \times t \rightarrow^{r!t}$ `unit`
`accept`:	$(t$ `chan` $r) \rightarrow^{r?t} t$
`transmit`:	$(t$ `chan` $r) \times t \rightarrow^{\epsilon}$ `unit event` $(r!t)$
`receive`:	$(t$ `chan` $r) \rightarrow^{\epsilon} t$ `event` $(r?t)$
`sync`:	$(t$ `event` $b) \rightarrow^b t$
`wrap`:	$(t_1$ `event` $b_1) \times (t_1 \rightarrow^b t_2) \rightarrow^{\epsilon} t_2$ `event` $(b_1; b)$
`choose`:	$(t$ `event` $b)$ `list` $\rightarrow^{\epsilon} t$ `event` b

The construction of the behaviours can be formulated as an annotated type system and below we illustrate the basic idea; for the details we refer to [7, 1].

A type environment *tenv* gives the annotated type of a variable and just mentioning a variable x (in a call-by-value language like CML) does not give rise to any interesting behaviour so we write this as

$$tenv \vdash x : t \;\&\; \epsilon \;\text{ if } tenv(x) = t$$

We have a similar axiom for constants: mentioning a constant (like a numeral or one of the primitive operators above) does not involve any computation so we have

$$tenv \vdash c : t_c \ \& \ \epsilon$$

where t_c is (an instance of) the type of c.

For ordinary function abstraction we take

$$\frac{tenv[x \mapsto t_1] \vdash e : t_2 \ \& \ b}{tenv \vdash \texttt{fn}\ x\ \texttt{=>}\ e : t_1 \rightarrow^b t_2 \ \& \ \epsilon}$$

So we guess a type t_1 for the formal parameter x and analyse the body of the abstraction to determine its type t_2 and its behaviour b. We record the behaviour as part of the overall type of the abstraction and note that as far as communication goes nothing interesting has happened so the overall behaviour will again be ϵ. The case of recursive function definition is fairly similar

$$\frac{tenv[f \mapsto t_1 \rightarrow^b t_2; x \mapsto t_1] \vdash e : t_2 \ \& \ b}{tenv \vdash \texttt{fun}\ f\ x\ \texttt{=>}\ e : t_1 \rightarrow^b t_2 \ \& \ \epsilon}$$

and here we will typically rely on b being a recursive behaviour that can be unfolded as demanded by the unfolding of the recursive function call.

Turning to the rule for function application we have

$$\frac{tenv \vdash e_1 : t_1 \rightarrow^b t_2 \ \& \ b_1, \quad tenv \vdash e_2 : t_1 \ \& \ b_2}{tenv \vdash e_1\ e_2 : t_2 \ \& \ (b_1; b_2; b)}$$

The idea is that we first determine the annotated type and the behaviour of the operator and the operand. CML has a call-by-value parameter mechanism so operationally we will first observe the communications originating from the operator, then those from the operand and finally those from the called function. Hence the application will have the behaviour $b_1; b_2; b$ – note that the causality of the communications are recorded.

In order for this approach to work we have to be able to enlarge the behaviours. As an example, all the elements in the argument list to the `choose` primitive must have the same behaviour and to achieve this we shall need a subsumption rule like

$$\frac{tenv \vdash e : t \ \& \ b}{tenv \vdash e : t \ \& \ b'} \ \text{if } b \sqsubseteq b'$$

Here $b \sqsubseteq b'$ is some ordering on behaviours that for example will express that `+` is an upper bound operator so b_1 can be enlarged to b_1 `+` b_2. The ordering will also express that ϵ is a left and right identity for sequencing ($\epsilon; b = b = b; \epsilon$) and this allows us to get rid of a lot of uninteresting occurrences of ϵ.

The full type system employs a general subtyping rule and also has rules for dealing with ML-like polymorphism; we shall spare the reader for these details as they do not seem so important for the current discussion. Instead we refer

$$
\begin{aligned}
B &= \textsc{fork}(B_0)\\
B_0 &= \texttt{\{belt1\_transmit\_ready\}?unit;\{belt1\_transmit\_done\}?unit;}\\
&\quad \texttt{\{table\_angle\}?int;\{table\_right\}!unit;}\, B_1 \,\texttt{;\{table\_stop\_h\}!unit;}\\
&\quad \texttt{\{table\_upward\}!unit;\{table\_is\_top\}?unit;\{table\_stop\_v\}!unit;}\\
&\quad \texttt{\{table\_transmit\_ready\}!unit;\{table\_transmit\_done\}!unit;}\\
&\quad \texttt{\{table\_downward\}!unit;\{table\_is\_bottom\}?unit;\{table\_stop\_v\}!unit;}\\
&\quad \texttt{\{table\_angle\}?int;\{table\_left\}!unit;}\, B_1 \,\texttt{;\{table\_stop\_h\}!unit;}\\
&\quad B_0\\
B_1 &= \texttt{\{new\_table\_angle\}?unit;\{table\_angle\}?int;(}\epsilon\ \texttt{+}\ B_1\texttt{)}
\end{aligned}
$$

Fig. 3. Behaviour for the table.

to the development in [1] for the many fine details concerning the ordering $\sqsubseteq$, subtyping, polymorphism, constraint simplification, semantic soundness of the inference system, and syntactic soundness and completeness of the inference algorithm.

The type and behaviour reconstruction algorithm has been implemented in Moscow ML and is available on the web[1]. It has been used to analyse the CML program implementing the Production Cell. For the part of the program corresponding to Figure 2 the algorithm will determine the type `unit` $\rightarrow^B$ `thread_id` where B is the behaviour of Figure 3.

Correctness issues. The language CML as well as the language of behaviours are equipped with a small-step operational semantics. This forms the basis for a correctness proof that essentially says that whenever the CML program performs a sequence of steps then also the associated behaviour can perform similar steps. To be more specific: when the semantics of the CML program performs a step corresponding to sending a value v of type t on some channel ch in some region r then the semantics of the behaviour can take a step that will execute the basic behaviour $r!t$, and similarly for the other primitive actions. Thus the behaviours give a *safe* approximation of the communications performed by the CML program.

The behaviour may be able to perform more actions than are possible by the CML program, for example because it will always be able to take both branches of a conditional. However, in the case where the behaviour only can perform one action then the CML will eventually have to perform a matching action – unless it is deadlocked or is looping. To illustrate this, consider a behaviour that contains the sequence

$$\texttt{\{table\_is\_not\_top\}?unit; \{table\_upward\}?unit}$$

[1] `http://www.daimi.aau.dk/~bra8130/TBAcml/TBA_CML.html`

and assume the behaviour of the process of interest only has those two occurrences of communications on the channels `table_is_not_top` and `table_upward`. Then the correctness result will tell us two things. First, if the CML program engages in a communication on `table_upward` then it will already have communicated on `table_is_not_top`. Second, after having engaged in a communication on `table_is_not_top` then it will eventually perform a communication on `table_upward` – unless it enters a looping computation or a deadlock between the two communications.

4 Safety conditions

Most safety conditions of the Production Cell [4] are concerned about the interplay between communications of only a few channels. Much of this information is directly available in the behaviours and we can easily attempt validating the three conditions mentioned in Section 2 based on the behaviours given in Figure 3. However, it is convenient to be able to ignore those channels that are not relevant for validating the condition at hand, i.e. to abstract away from communications on those channels.

As an example, suppose that we want to validate the following safety condition:

> *The engine starting the vertical movement of the table is always turned off before it is turned on (assuming that it is initially turned off).*

We shall rely on some assumptions about the environment: The engine can only be turned on using one of the two channels `table_upward` and `table_downward` and it can only be turned off using the channel `table_stop_v`. We shall therefore replace all communications mentioned in Figure 3 that do *not* involve any of these three channels with ellipses and then we shall apply some straightforward simplifications in order to obtain:

$$\begin{aligned} B_0 = {} & \cdots ; \texttt{\{table\_upward\}!unit}; \cdots ; \texttt{\{table\_stop\_v\}!unit}; \\ & \cdots ; \texttt{\{table\_downward\}!unit}; \cdots ; \texttt{\{table\_stop\_v\}!unit}; \\ & \cdots ; B_0 \end{aligned}$$

This simplified behaviour clearly shows that the engine is turned on and off in the manner described by the safety condition.

Just as our prototype is responsible for producing the behaviour of Figure 3 it can also be used to produce the above simplified behaviours. The theoretical foundations for the simplified behaviours are established in [1].

We shall now go through the three safety conditions of the rotating table mentioned in Section 2 and discuss to what extent they can be validated using the behaviours. Based on the informal description of the condition and some overall assumptions about the environment we shall decide which channels are of relevance for the condition and extract that part of the behaviour. It turns out that

this will be a fairly simple behaviour so we can immediately judge whether or not the safety condition is fulfilled; clearly a more formal approach is possible as well.

Condition 1.

> *The table must not be moved downward if it is in its lower position, and it must not be moved upward if it is in its upper position.*

Validation of this condition relies on some assumptions about the environment: The vertical movement of the table can only be initiated by communicating on the two channels `table_upward` and `table_downward`. Information about the vertical position of the table can only be obtained from the four channels `table_is_bottom`, `table_is_not_bottom`, `table_is_top` and `table_is_not_top`.

We therefore select these six channels and obtain the following simplified behaviour from Figure 3:

$$
\begin{aligned}
B_0 = \cdots &\texttt{;\{table\_upward\}!unit;\{table\_is\_top\}?unit;} \\
\cdots &\texttt{;\{table\_downward\}!unit;\{table\_is\_bottom\}?unit;} \\
\cdots &\texttt{;}B_0
\end{aligned}
$$

Thus we see that all communications on `table_downward` are preceeded by a communication on `table_is_top`. By unfolding the behaviour is is also easy to see that, except for the initial case, all communications on `table_upward` are preceeded by a communication on `table_is_bottom`.

However, this is not the case for the initial communication on `table_upward`. The behaviour will *never* allow a communication on any of the four channels giving information about the vertical position of the table before the initial communication on the channel `table_upward`. It follows that the CML program will never be able to do that either. Hence the analysis has shown that the CML program does *not* fulfil Condition 1!

Condition 2.

> *The table must not be rotated clockwise if it is in the position required for transferring work pieces to the robot, and it must not be rotated counterclockwise if it is in the position to receive work pieces from the feed belt.*

Again we have to rely on some assumptions about the environment. The rotation of the table can only be initiated by communication on one of the two channels `table_right` and `table_left` and it is stopped by communication on the channel `table_stop_h`. The horizontal position of the table can be obtained from the channel `table_angle`.

We therefore extract the behaviour involving the four channels mentioned above and get:

$$
\begin{aligned}
B_0 = \cdots & \texttt{;\{table\_angle\}?int;\{table\_right\}!unit;} B_1 \texttt{;\{table\_stop\_h\}!unit;} \\
& \cdots \texttt{;\{table\_angle\}?int;\{table\_left\}!unit;} B_1 \texttt{;\{table\_stop\_h\}!unit;} \\
& B_0 \\
B_1 = \cdots & \texttt{;\{table\_angle\}?int;(}\epsilon \texttt{ + } B_1\texttt{)}
\end{aligned}
$$

From this it is easy to see that we have validated the following version of the safety condition:

> *The table is alternating between being rotated clockwise and counterclockwise.*

However there is no information in the behaviours ensuring that the clockwise rotation stops when the angle is 50 (as required for the robot) or that the counterclockwise rotation stops when the angle is 0 (as required for the feed belt). More powerful analysis techniques will be needed to capture this kind of information; we shall return to this in Section 5.

Condition 3.

> *There can only be one work piece at the table at any time.*

This condition is concerned about the synchronisation between the individual processes of the system and hence its validation will depend on properties of the other processes, in particular those for the feed belt and the robot. The table is the passive part in both of these synchronisations. The channels `belt1_transmit_ready` and `belt1_transmit_done` are used to synchronise with the feed belt; between these two communications it is the responsibility of the feed belt to place a work piece on the table. The channels `table_transmit_ready` and `table_transmit_done` are used to synchronise with the robot; between these two communications it is the responsibility of the robot to remove a work piece from the table.

The analysis of the table will therefore need to make some assumptions about the feed belt and the robot. These assumptions will later have to be validated by analysing the behaviour of the program fragments for the respective processes. The assumptions are:

(a) Whenever the feed belt leaves the critical region specified by the two channels `belt1_transmit_ready` and `belt1_transmit_done` it will have moved one (and only one) work piece to the table.
(b) Whenever the robot leaves the critical region specified by the two channels `table_transmit_ready` and `table_transmit_done` it will have emptied the table.

Under these assumptions we can now validate Condition 3.

We shall concentrate on the four channels specifying the critical regions and we obtain the following simplified behaviour for the table:

$$
\begin{aligned}
B_0 = {} & \texttt{\{belt1\_transmit\_ready\}?unit;\{belt1\_transmit\_done\}?unit;} \cdots; \\
& \texttt{\{table\_transmit\_ready\}!unit;\{table\_transmit\_done\}!unit;} \cdots; \\
& B_0
\end{aligned}
$$

Clearly this shows that the two pairs of communications alternate. Also it shows that the synchronisation with the feed belt happens first and by assumption (a) a work piece is placed on the table. The simplified behaviour shows that subsequently there will be a synchronisation with the robot and by assumption (b) the work piece will be removed from the table. Hence Condition 3 has been validated with respect to the assumptions.

5 Discussion and further work

The results obtained from the analysis depend to a large extent on the programming style. As an example, an alternative program for the Production Cell uses the following function instead of the two functions `clockwise` and `counterclockwise`:

```
fun turn_to(a) =
    let val x = accept(table_angle) in
    if x < a then
       (send(table_right,());
        while (accept(new_table_angle); accept(table_angle)) < a
        do ();
        send(table_stop_h,()) )
    else if x > a then
       (send(table_left,());
        while (accept(new_table_angle); accept(table_angle)) > a
        do ();
        send(table_stop_h,()) )
    else ()
    end;
```

In the setting provided by Condition 2 we now get the following simplified behaviour for the program:

$$
\begin{aligned}
B_0 = {} & \cdots; B_1; \cdots; B_1; B_0 \\
B_1 = {} & \texttt{\{table\_angle\}?int;} \\
& (\epsilon \texttt{ + \{table\_left\}!unit;} B_2 \texttt{;\{table\_stop\_h\}!unit} \\
& \quad \texttt{+ \{table\_right\}!unit;} B_2 \texttt{;\{table\_stop\_h\}!unit)} \\
B_2 = {} & \cdots \texttt{;\{table\_angle\}?int;(}\epsilon \texttt{ + } B_2\texttt{)}
\end{aligned}
$$

As expected we cannot validate Condition 2 from this. But even worse, we cannot even validate that the table is alternating between being rotated clockwise and counterclockwise; only that it is rotated an even number of times. The reason for the latter is that the current version of our technology does not incorporate any information about values of variables and the entities communicated and therefore we cannot prune the behaviour for `turn_to` to take the branch of interest for a given value of the parameter. We expect that techniques from Control Flow Analysis [3, 6] will prove useful when further developing the technology.

The CML program for the Production Cell is basically a first-order program and hence it does not exploit the higher-order constructs of CML. Our technique has no problems handing higher-order functions nor communication of channels. To illustrate a simple version of this, consider the following generic function

```
fun move start doit stop = (send(start,()); doit(); send(stop,()))
```

that takes a channel, a function and yet another channel as arguments. Let us rewrite the program to use this function:

```
fun table () =
let
   fun clockwise (a) =
       let val x = accept(table_angle);
       in move table_right
               (fn () => while (accept(new_table_angle);
                                accept(table_angle)) < a do ())
               table_stop_h
       end;

   fun counterclockwise (a) =
       let val x = accept(table_angle)
       in move table_left
               (fn () => while (accept(new_table_angle);
                                accept(table_angle)) > a do ())
               table_stop_h
       end;

   fun main () =
       (accept(belt1_transmit_ready); accept(belt1_transmit_done);
        clockwise(50);
        move table_upward (fn () => accept(table_is_top)) table_stop_v;
        send(table_transmit_ready,()); send(table_transmit_done,());
        move table_downward (fn () => accept(table_is_bottom)) table_stop_v;
        counterclockwise(0);
        main())
in
   spawn(fn () => main())
end;
```

The behaviour of this version of the program is exactly as in Table 3; in particular the techniques easily distinguish between the different sets of parameters supplied to the four calls of the `move` function.

6 Conclusion

We have argued that even the careful use of formal program development techniques may in practice produce bugs that go undetected. To increase the available techniques for validating embedded systems we have argued that the use of novel program analysis technology is likely to be indispensable and we have substantiated this claim by the development of a prototype.

Acknowledgements. We should like to thank H. Rischel for providing us with the simulator for Production Cell as well as the CML program for controlling the Production Cell, and also A. P. Ravn for general discussions about the analysis of embedded systems. This work has been supported in part by the DART project funded by the Danish Science Research Council and also builds on theories and tools developed during the LOMAPS project funded by ESPRIT BRA.

References

1. T. Amtoft, F. Nielson, and H. R. Nielson. Polymorphic subtyping for side effects. Book manuscript, DAIMI PB-529, Aarhus Univesity, 1997.
2. T. Amtoft, H. R. Nielson, and F. Nielson. Behaviour analysis for validating communication patterns. DAIMI PB-527, Aarhus University, 1997.
3. K. L. S. Gasser, F. Nielson, and H. R. Nielson. Systematic realisation of control flow analyses for CML. In *Proceedings of ICFP'97*, pages 38–51. ACM Press, 1997.
4. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems, Case Study "Production Cell"*. SLNCS vol 891, Springer Verlag, 1995.
5. R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.
6. F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis: Flows and Effects*. To appear, 1999.
7. H. R. Nielson and F. Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. In *Proc. POPL '94*, 1994.
8. H. R. Nielson and F. Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monographs in Computer Science. Springer-Verlag, 1997.
9. J.H. Reppy. Concurrent ML: Design, application and semantics. In *Proc. Functional programming, Concurrency, Simulation and Automated Reasoning, SLNCS 693*, pages 165–19, 1993.
10. H. Rischel and H. Sun. Design and prototyping of real-time systems using CSP and CML. In *Proc. 9th Euromicro Workshop on Real-Time Systems*, pages 121–127. IEEE Computer Society Press, 1997.