Behaviour Analysis for Validating Communication Patterns

Torben Amtoft and Hanne Riis Nielson and Flemming Nielson
DAIMI, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark
{tamtoft,hrn,fn}@daimi.aau.dk

September 10, 1997

Abstract

The communication patterns of concurrent programs can be expressed succinctly using *behaviours*; these can be viewed as a kind of causal constraints or as a kind of process algebra terms. We present a system which infers behaviours from a useful fragment of Concurrent ML programs; it is based on previously developed theoretical results and forms the core of a system available on the Internet. By means of a case study, used as a benchmark in the literature, we shall see that the system facilitates the validation of certain safety conditions for reactive systems.

1 Introduction

It is well-known that testing can only demonstrate the presence of bugs, never their absence. This has motivated a vast amount of research into techniques for guaranteeing statically (that is, at compile-time rather than at run-time) that the software behaves in certain ways; a prime example is the formal verification of software. In this line of development various notions of *type systems* have been put forward because they allow to perform static checks of certain kinds of bugs: at run-time there may still be a need to check for division by zero but there will never be a need to check for the addition of booleans and files. As programming languages evolve in terms of features like module systems and the integration of different programming paradigms,

the research on type systems is constantly pressed for new problems to be treated.

Our research has been motivated by the integration of the functional and concurrent programming paradigms. We believe this combination to be particularly attractive since *(i)* many real-time applications demand concurrency, and *(ii)* many algorithms can be expressed elegantly and concisely as functional programs. Example programming languages are Concurrent ML (CML) [21] that extends Standard ML (SML) [13] with concurrency, and Facile [27] that follows a similar approach but more directly contains syntax for expressing CCS-like process composition. By the very nature of programming, the overall communication pattern of a CML (or Facile) program may not be immediately transparent, and compact ways of expressing the communications taking place are desired. One such representation is *behaviours* [17, 2], a kind of process algebra terms based on [14]; they are related to "effects" [25] but augment these in that they convey causality information.

To illustrate the use of behaviours to show the absence of bugs in concurrent systems, consider the following safety criterion: a machine M must not be started until the temperature has reached a certain level. In a CML programming environment, this criterion may amount to saying that a process P should never send a signal over the channel `start_M` unless it has just received a signal over the channel `temp_OK`. Now suppose we can show that P has some behaviour b: then it may be immediate that the above safety property holds, for instance if b is defined recursively as $b = \cdots; \texttt{temp\_OK}?; \texttt{start\_M}!; b$ which should be interpreted as follows: P performs a cycle and in each iteration it first performs some "irrelevant" actions not affecting the two channels of interest; then it receives a signal over `temp_OK`; and finally it sends a signal over `start_M`. Other applications of behaviour information are demonstrated in [17, 18].

Theoretical foundations. We have developed an algorithm which given a CML program P returns a behaviour b; in order for this algorithm to be trustworthy it must be ensured that b is "faithful" to the actual run-time behaviour of P. To do so we have followed a classical recipe and

1. defined an *inference system* for behaviours, giving rules for when it is possible to assign a given behaviour to an expression;
2. demonstrated that this inference system is sound wrt. a semantics for

CML, in the sense that "well-typed programs communicate according to their behaviour" (and hence that "well-typed programs do not go wrong", cf. [12]);

3. demonstrated that the algorithm is sound wrt. the inference system, i.e. that its output represents a valid inference.

The above rather extensive development has been carried out in [3] for a core subset[1] of CML; in addition a completeness result has been established, stating that the inferred behaviours are in a certain sense principal.

The system. We have developed a tool for behaviour analysis, based on the above algorithm; it can be accessed for experimentation at

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

The system takes as input a CML program and produces as output its behaviour. The user interface allows to restrict the attention to a selection of channels, in which case the output often becomes both readable and informative and thereby enables one to validate certain safety criteria.

Overview. In Section 2 we give a brief overview of the programming language CML. In Section 3 we introduce the notion of behaviours and illustrate a few of the rules for assigning behaviours to expressions. Section 4 sketches our inference algorithm which returns a set of constraints as output; in Section 5 we shall see how to manipulate, and partially solve, these constraints so to improve readability. The user interface is outlined in Section 6. The use of the system for validating a program implementing the *Karlsruhe production cell* [11] is briefly presented in Section 7; luckily it turns out that a number of safety properties can in fact be validated.

2 Concurrent ML

The language Concurrent ML (CML) extends Standard ML (SML) with primitives for concurrency; before elaborating on these we shall first give a

[1] In most cases it is possible to incorporate extra constants or language constructs into this subset without too much effort. However, to handle functions (such as `wrap`) that manipulate events (cf. Sect. 2) we need to perform a non-trivial reformulation of the semantics, similar to what is done in [19].

short tutorial to features common for SML and CML.

Functional features. SML [13] is an eager (call-by-value) *functional language*, in the sense that a program consists of a sequence of function definitions. Much of the popularity of functional languages stems from the possibility of defining *generic* functions that are applicable in a variety of contexts. As an example of this, consider the function `foldr` defined by

```
fun foldr f [] a = a
|   foldr f (x::xs) a = f (x, foldr f xs a)
```

which processes the list, given as second argument, from right to left. One use for this function is to find the number of non-zero elements in a list:

```
fun num_non_zero xs =
   foldr (fn (x,a) => if x = 0 then a else a+1)
         xs 0
```

SML is equipped with a type system which ensures that "well-typed programs cannot go wrong" [12], that is if it is possible at compile-time to assign a type to a given expression then certain kinds of run-time errors (such as adding a boolean to an integer) cannot happen (whereas others like division by zero may still occur). The SML type system will assign `foldr` the type

$$(\alpha_1 \times \alpha_2 \to \alpha_2) \to \alpha_1\ \texttt{list} \to \alpha_2 \to \alpha_2$$

This reflects that `foldr` demands an argument f which is a binary function, an argument xs which is a list, and an argument a whose type equals the result type of `foldr`.

The type information pinpoints the generic nature of `foldr`:

1. it is *higher-order*, i.e. its arguments may be functions themselves;
2. it is *polymorphic* as can be seen from the presence of α_1 and α_2, these *type variables* may each be instantiated to different types (such as `int` or `bool`) in different contexts.

Concurrent features. CML [21] extends SML with primitives for concurrency: the primitive `spawn` creates a new process; the primitive `channel`

creates a new channel; the primitive `send` sends a value over a channel as soon as some other process is ready to receive, and blocks until this is the case; the primitive `accept` receives a value from a channel as soon as some other process is ready to send, and blocks until this is the case. This is not an exhaustive list of all the concurrency primitives and we shall introduce additional constructs in the sequel.

Also in a concurrent setting generic functions come to good use; as a simple first example of this consider the function below which makes use of the sequencing operator ";" well-known from many imperative languages. First it sends a signal over the channel `start_ch` (this may start some machine); then it waits for the termination of function `wait_fun` (which may loop until the machine is in some desired position); finally it sends a stop signal.

```
fun move_until wait_fun start_ch stop_ch =
        ( send(start_ch,())
        ; wait_fun ()
        ; send(stop_ch,()) )
```

The CML type system will assign it the type

$$(\texttt{unit} \to \alpha) \to \texttt{unit chan} \to \texttt{unit chan} \to \texttt{unit}$$

which is still higher-order but not really polymorphic (as α will typically be `unit`); here `unit` is the singleton type with `()` as its only element.

In order for a concurrent system to behave in a systematic way it is convenient to impose some protocol on the communication pattern. An example protocol is the *double handshake* which allows an "active" part to interact with a "passive" part. When the passive part is ready to interact it sends a signal over a channel; when the active part has received this signal it performs its "critical action" and afterwards sends a signal to indicate completion. In the case of CML only one channel is needed since channels are bidirectional; the following piece of code thus implements the role of the passive part:

```
fun passive_sync ready_ch =
        ( send(ready_ch,())
        ; accept(ready_ch)
        )
```

and the following piece of code implements the role of the active part:

```
fun active_sync ready_ch crit_action =
        ( accept(ready_ch)
        ; crit_action ()
        ; send(ready_ch,())
        )
```

A passive process may be able to interact with several active processes (or vice versa) with the subsequent actions depending on the partner chosen. This can be expressed by the pseudo-code (in the style of [8])

```
if  ready ch1 → passive_sync ch1; cont1 ()
 [] ready ch2 → passive_sync ch2; cont2 ()
fi
```

where `cont1` represent the "rest of the computation" in the case where `ch1` is ready, similarly for `cont2`, and in general `cont1` may differ from `cont2`.

We shall see that this can in fact be expressed in CML, using the notion of *events*: one can think of an event as a communication *possibility*. Events are first class values, that is they can be passed around just like integers.

The CML primitive `sync` *synchronises* an event, which turns into an actual communication; the operation may block if there is no communication partner. The CML primitive `select` is as `sync` except that it takes a *list of* events and synchronises one of these, the choice is deferred until it can be ensured that the selected communication will not block.

To *create* events, CML offers the primitive `transmit` which is a "non-committing" version of `send` in that it only *creates* an event without *synchronising* it; in a similar way the primitive `receive` is a non-committing version of `accept`. (We have the equations `send` x = `sync` (`transmit` x) and `accept` x = `sync` (`receive` x).)

A first attempt to achieve the desired behaviour is then to write

```
  select [transmit(ch1,()), transmit(ch2,())]
; select [receive (ch1),    receive (ch2)   ]
; cont1 ()
```

but this will only work if `cont1` equals `cont2`; furthermore this piece of code does not exploit that the second occurrence of `select` has to conform with the choice made by the first occurrence. In the general case one therefore needs a way of "inlining" a "continuation" into an event, such that the event

applies the continuation if synchronised. This is taken care of by the CML primitive `wrap`; using this primitive the desired behaviour can be achieved by writing

```
select [wrap( transmit(ch1,()),
              fn () => ( accept(ch1)
                       ; cont1 ())),
        wrap( transmit(ch2,()),
              fn () => ( accept(ch2)
                       ; cont2 ()))]
```

This suggests that a passive process should use the following generic function for creating "handshake events":

```
fun passive_sync_event ready_ch cont =
        wrap ( transmit(ready_ch,()),
               fn () => ( accept(ready_ch)
                        ; cont ()
                        ))
```

This function returns an event which

- will only be synchronised if some other process is ready to receive on the channel `ready_ch`;
- if synchronised will *(i)* complete the double handshake, and *(ii)* execute the rest of the computation as specified by `cont`.

Accordingly the CML type system will assign the type

$$\texttt{unit chan} \rightarrow (\texttt{unit} \rightarrow \alpha) \rightarrow \alpha\ \texttt{event}$$

to `passive_sync_event` (where α will typically be `unit`).

Example. So far we have described a variety of building blocks for concurrent programs; we shall now illustrate how they can be combined. This will form the basis of the main running example of this paper.

Consider a conveyor belt, part of the larger production cell described in [11], used for transporting "blanks" from a robot to a crane. The default state of the belt is that there is one blank somewhere on the belt; the procedure to be iterated is

1. transport the blank onto the end of the belt;
2. let the crane pick up the blank;
3. wait for the robot arm to deliver another blank.

To detect whether or not the blank has reached the end of the belt we use two sensors connected to the same photo cell, situated shortly before the end: when its light ray is intercepted it signals on `belt2_blank_at_end`; when its light ray is no more intercepted (i.e. the blank has passed the photo cell and has thus reached the end) it signals on `belt2_no_blank_at_end`. Step 1 can then be implemented using `move_until`, binding its first argument `wait_fun` to a function with body

```
accept(belt2_blank_at_end); accept(belt2_no_blank_at_end)
```

Clearly step 2 and 3 can each be implemented using `passive_sync`, but as the robot may become ready before the crane, it is convenient to be able to switch the ordering. Accordingly we use `passive_sync_event` to create *two* events, "first 3 then 2" and "first 2 then 3".

The resulting code is:

```
let fun belt2_cycle () =
     ( move_until (fn ()=>(accept(belt2_blank_at_end);
                           accept(belt2_no_blank_at_end)))
                  belt2_start
                  belt2_stop;

       select [passive_sync_event belt2_ready_for_arm2
                  (fn () => passive_sync belt2_ready_for_crane),
               passive_sync_event belt2_ready_for_crane
                  (fn () => passive_sync belt2_ready_for_arm2)];

       belt2_cycle ())
 in ... end
```

□

3 Behaviours

As is apparent from the preceding section the CML type system may convey useful information about the *functionality* of a program, but when it

comes to analysing the *communication pattern* it is of little use. To facilitate the latter we augment the system by *annotating* certain CML types with *behaviour* and *region* information. The annotation β in a function type $t_1 \ \rightarrow^{\beta} \ t_2$ describes the behaviour taking place whenever the function is applied, and we shall continue to write $t_1 \rightarrow t_2$ for the type of a function that is applied "silently" (as will be the case for constructors like `pair` and `transmit`). The annotation β in an event type of the form t `event` β describes the behaviour taking place whenever the event is synchronised; finally the annotation ρ in a channel type t `chan` ρ describes the region in which the channel is allocated. Regions can be thought of as sets of channels; we shall return to the issue in Section 5.

Example. The function `move_until` (introduced in Section 2) can be assigned the annotated type

$$(\texttt{unit} \ \rightarrow^{\beta_0} \ \alpha_0) \ \rightarrow^{\beta_1} \ \texttt{unit chan}\ \rho_0 \ \rightarrow^{\beta_2} \ \texttt{unit chan}\ \rho_1 \ \rightarrow^{\beta_3} \ \texttt{unit}$$

Here β_1 as well as β_2 "denotes" the empty behaviour ε, which reflects that `move_until` does not perform any action until it has been supplied with *three* arguments; and β_3 denotes the behaviour

$$(\rho_0\,!\,\texttt{unit}); \beta_0; (\rho_1\,!\,\texttt{unit})$$

which reflects that `move_until` when applied to the three arguments f, ch_0, and ch_1 performs the following actions:

- first it sends the value `()` over the channel ch_0, located in region ρ_0;
- then it calls f with the argument `()`, since f has type `unit` $\rightarrow^{\beta_0} \ \alpha_0$ this will behave as indicated by β_0;
- finally it sends `()` over the channel ch_1, located in region ρ_1. □

The notion of annotated types[2] goes way back in the literature; a classic example being the *effects* of [25]. In the present paper, a behaviour b is either

- a variable β (cf. the use of type variables α);

[2] In the following we shall often write "type" for "annotated type".

- the empty behaviour ε (no "visible" actions take place);
- a sequential composition $b_1; b_2$ (first b_1 and then b_2 takes place);
- a choice operator $b_1 + b_2$ (either b_1 or b_2 takes place);
- *SPAWN* b (a process is spawned which behaves as indicated by b);
- t CHAN ρ (a channel, able to transmit values of type t, is allocated in region ρ);
- $\rho\,?\,t$ (a value of type t is read from a channel situated in region ρ);
- $\rho\,!\,t$ (a value of type t is written to a channel situated in region ρ).

Example. The function `passive_sync` can be assigned the type

$$\texttt{unit chan}\,\rho \;\rightarrow^{\beta}\; \texttt{unit}$$

where β denotes $\rho\,!\,\texttt{unit}; \rho\,?\,\texttt{unit}$

and its cousin `passive_sync_event` can be assigned the type

$$\texttt{unit chan}\,\rho \;\rightarrow^{\beta_0}\; (\texttt{unit} \;\rightarrow^{\beta_1}\; \alpha) \;\rightarrow^{\beta_0}\; \alpha\;\texttt{event}\;\beta$$

where β_0 denotes ε and β denotes $\rho\,!\,\texttt{unit}; \rho\,?\,\texttt{unit}; \beta_1$

Here β_1 is the behaviour of `cont` representing the rest of the computation. □

3.1 Inference system

We now present some of the rules for assigning annotated types to CML expressions; as in [20] (which was in turn inspired by [24, 9]) *judgements* take the form

$$C, A \vdash e : t \,\&\, b$$

Such a judgement states that the CML expression e has type t and that the evaluation of e gives rise to visible actions as indicated by b, assuming that

- the *environment* A contains type information about the identifiers occurring free in e;
- the relation between the various variables in t and b is given by the *constraint set* C.

Conditionals. The type of a conditional is determined by the rule

$$\begin{array}{ll} \text{if} & C, A \vdash e_0 : \texttt{bool} \,\&\, b_0 \\ \text{and} & C, A \vdash e_1 : t \,\&\, b_1 \\ \text{and} & C, A \vdash e_2 : t \,\&\, b_2 \\ \text{then} & C, A \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : t \,\&\, b_0; (b_1 + b_2) \end{array}$$

where the use of the choice operator $b_1 + b_2$ reflects that we cannot predict which branch will be taken.

Function applications. The type of a function application is determined by the rule

$$\begin{array}{ll} \text{if} & C, A \vdash e_1 : (t_2 \ \rightarrow^{\beta}\ t_1) \,\&\, b_1 \\ \text{and} & C, A \vdash e_2 : t_2 \,\&\, b_2 \\ \text{then} & C, A \vdash e_1\ e_2 : t_1 \,\&\, (b_1; b_2; \beta) \end{array}$$

where the behaviour $b_1; b_2; \beta$ clearly states that CML employs a call-by-value evaluation strategy: first the function e_1 is evaluated, then its argument e_2 is evaluated, and finally the function is applied enacting the latent behaviour on the function arrow.

Function abstractions. The type of a function abstraction is determined by the rule

$$\begin{array}{ll} \text{if} & C, A[x : t_x] \vdash e : t \,\&\, \beta \\ \text{then} & C, A \vdash \texttt{fn}\ x \Rightarrow e : t_x \ \rightarrow^{\beta}\ t \,\&\, \varepsilon \end{array}$$

where the body e is analysed in an environment binding x to t_x, with its behaviour β becoming latent in the resulting function type.

3.2 Subtyping and subeffecting

Our system is designed to be a conservative extension of the CML type system, i.e. all programs typeable in the latter are also typeable in the former and vice versa. But when examining the above rules this might seem not to be the case: in particular the rule for function abstraction $\texttt{fn}\ x \Rightarrow e$

cannot be applied unless the body e can be assigned a behaviour which is a variable, as only these may appear on arrows.

On the other hand, even if b is not a variable the judgement

$$C, A[x : t_x] \vdash e : t \,\&\, b$$

allows us to obtain

$$C, A \vdash \texttt{fn}\ x \Rightarrow e : t_x \ \rightarrow^{\beta}\ t \,\&\, \varepsilon$$

provided that it is possible from C to deduce that $b \subseteq \beta$; this will for instance be the case if the *behaviour constraint* ($b \subseteq \beta$) is contained in C. Formally, this is due to the *subeffecting* rule

$$\begin{array}{ll} \text{if} & C, A \vdash e : t \,\&\, b \text{ and } C \vdash b \subseteq b' \\ \text{then} & C, A \vdash e : t \,\&\, b' \end{array}$$

Here the behaviour ordering $C \vdash b_1 \subseteq b_2$ states that (given the assumptions in C) b_1 is a more precise behaviour than b_2 in the sense that any action performed by b_1 can also be performed by b_2.[3]

The rules defining the ordering express that sequential composition ";" is associative with ε as neutral element; that "$\subseteq$" is a congruence wrt. the various behaviour constructors; and that "$+$" is least upper bound wrt. $\subseteq$.

In order to increase the precision of the analysis we need also *subtyping*, as witnessed by the situation below (cf. the considerations in [26, Chap. 5]): we want to type a conditional $\texttt{if}\ e_0\ \texttt{then}\ f_1\ \texttt{else}\ f_2$ occurring inside some function body e situated in the context $(\texttt{fn}\ f_1 \Rightarrow \texttt{fn}\ f_2 \Rightarrow e)\ e_1\ e_2$, where we can assign e_1 the type $t_1 = \texttt{int}\ \rightarrow^{\beta_1}\ \texttt{int}$ and e_2 the type $t_2 = \texttt{int}\ \rightarrow^{\beta_2}\ \texttt{int}$ with $\beta_1 \neq \beta_2$. In order to use the rule for conditional, we must be able to assign f_1 and f_2 a common type $t = \texttt{int}\ \rightarrow^{\beta}\ \texttt{int}$. Assuming that $\beta_1 \subseteq \beta$ and $\beta_2 \subseteq \beta$ can be deduced from the constraint set, there are two approaches to achieve this. *(i)* The insertion of subeffecting may convert the typings of e_1 and e_2 into typings where both expressions are assigned the type t, then the body e is typed using an environment where both f_1 and f_2 are bound to t; the price to pay is that then *all* occurrences of f_1 in e

[3] A similar claim is formalised in [19] where a syntactically defined ordering on behaviours is shown to be a decidable subset of the undecidable simulation ordering, induced by an operational semantics for behaviours.

are indistinguishable from f_2, and vice versa. *(ii)* The use of subtyping, on the other hand, allows f_1 to be bound to t_1 and f_2 to be bound to t_2 when typing e; then t_1 and t_2 are approximated to t immediately before the rule for conditional is applied, using the subtyping rule

$$\begin{array}{ll} \text{if} & C, A \vdash e : t \,\&\, b \text{ and } C \vdash t \subseteq t' \\ \text{then} & C, A \vdash e : t' \,\&\, b \end{array}$$

Here the subtype relation $C \vdash t_1 \subseteq t_2$ states that (given the assumptions in C) t_1 is a more precise type than t_2; it is induced by the subeffecting relation and unlike e.g. [24] we do not have any ordering on base types, such as `int` $\subseteq$ `real`. As is to be expected, the ordering is contravariant in the argument position of a function type:

$$\begin{array}{ll} \text{if} & C \vdash t_1' \subseteq t_1 \text{ and } C \vdash \beta \subseteq \beta' \text{ and } C \vdash t_2 \subseteq t_2' \\ \text{then} & C \vdash t_1 \ \to^{\beta} \ t_2 \subseteq t_1' \ \to^{\beta'} \ t_2' \end{array}$$

Of particular interest is the rule for channel types:

$$\begin{array}{ll} \text{if} & C \vdash t \subseteq t' \textit{ and } C \vdash t' \subseteq t \\ \text{and} & C \vdash \rho \subseteq \rho' \\ \text{then} & C \vdash t \ \texttt{chan}\, \rho \subseteq t' \ \texttt{chan}\, \rho' \end{array}$$

which reflects that the type of communicated values essentially occurs both covariantly (when used in `receive`) and contravariantly (when used in `send`).

3.3 Polymorphism

In order for a function to be used polymorphically the environment must map its name into a type *scheme* rather than just a type. In SML type schemes are of form $\forall\vec{\alpha}.t$ where $\vec{\alpha}$ are the *bound* type variables; in [24], which extends polymorphism with subtyping, type schemes are augmented with constraints so as to be of the form $\forall(\vec{\alpha} : C).\ t$; in our approach we also consider behaviour and region variables and hence type schemes are of the form $\forall(\vec{\alpha}\vec{\beta}\vec{\rho} : C).\ t$.

CML primitives. Closed type schemes have been preassigned to all the primitives, of which we list a few:

`send`	$\forall(\alpha\beta\rho : \{\rho\,!\,\alpha \subseteq \beta\}).\ (\alpha\ \texttt{chan}\ \rho) \times \alpha\ \rightarrow^{\beta}\ \texttt{unit}$
`transmit`	$\forall(\alpha\beta\rho : \{\rho\,!\,\alpha \subseteq \beta\}).\ (\alpha\ \texttt{chan}\ \rho) \times \alpha \rightarrow (\texttt{unit}\ \texttt{event}\ \beta)$
`accept`	$\forall(\alpha\beta\rho : \{\rho\,?\,\alpha \subseteq \beta\}).\ (\alpha\ \texttt{chan}\ \rho)\ \rightarrow^{\beta}\ \alpha$
`receive`	$\forall(\alpha\beta\rho : \{\rho\,?\,\alpha \subseteq \beta\}).\ (\alpha\ \texttt{chan}\ \rho) \rightarrow (\alpha\ \texttt{event}\ \beta)$
`sync`	$\forall(\alpha\beta : \emptyset).\ (\alpha\ \texttt{event}\ \beta)\ \rightarrow^{\beta}\ \alpha$
`spawn`	$\forall(\alpha\beta\beta' : \{SPAWN\,\beta' \subseteq \beta\}).\ (\texttt{unit}\ \rightarrow^{\beta'}\ \alpha)\ \rightarrow^{\beta}\ \texttt{unit}$
`wrap`	$\forall(\cdots : (\beta; \beta' \subseteq \beta'')).\ \alpha\ \texttt{event}\ \beta \times (\alpha\ \rightarrow^{\beta'}\ \alpha') \rightarrow \alpha'\ \texttt{event}\ \beta''$

The types make it clear that `transmit` is a non-committing version of `send`, and similarly that `receive` is a non-committing version of `accept`.

Definitions used polymorphically. In an expression $\texttt{let fun}\ f\ x = e_0\ \texttt{in}\ e\ \texttt{end}$, one can use f polymorphically in e; and as the definition is recursive, one can also use f in e_0 but not polymorphically — we do not allow polymorphic recursion. These considerations are reflected in the rule for typing such definitions, which looks like

if	$C \cup C_0, A[f : t_0][x : t_1] \vdash e_0 : t_2\,\&\,\beta$ with $t_0 = t_1\ \rightarrow^{\beta}\ t_2$
and	$C, A[f : \forall(\vec{\alpha}\vec{\beta}\vec{\rho} : C_0).\ t_0] \vdash e : t\,\&\,b$
then	$C, A \vdash \texttt{let fun}\ f\ x = e_0\ \texttt{in}\ e\ \texttt{end} : t\,\&\,b$
provided	$\cdots$

and it can be applied provided certain side conditions hold. The most familiar of these state that the bound variables $\vec{\alpha}\vec{\beta}\vec{\rho}$ must not occur in C or A; the remaining conditions restrict the form of C_0, and are trivially satisfied in the case where C_0 is empty.

The above rule can be extended to allow for "value polymorphism" as in $\texttt{let val}\ x = e_0\ \texttt{in}\ e\ \texttt{end}$ where the defined entity may be something else than a function. Then even more elaborate side conditions are needed in order to ensure semantic soundness, and as witnessed by the related approach in [1] this is a delicate matter; the basic ideas are *(i)* that we cannot generalise variables free in the behaviour (cf. [25]), and that *(ii)* the division between bound and free variables must in some sense "respect" the constraint set C_0.

4 Inference Algorithm

We shall aim at constructing a type reconstruction algorithm in the spirit of Milner's algorithm $\mathcal{W}$ [12]: given an expression e and an environment A, the recursively defined function $\mathcal{W}$ will produce a substitution S, a type t, and a behaviour b. The definition in [12] employs unification [23]: if e_i has been given type t_i (for $i = 0, 1, 2$) then in order to type $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ one must unify t_0 with $\texttt{bool}$ and t_1 with t_2. Unification works by decomposition: in order to unify $t_1 \to t_2$ and $t_1' \to t_2'$ one recursively unifies t_1 with t_1' and t_2 with t_2'; and to unify a variable α with a type t one produces the substitution $[\alpha \mapsto t]$ (assuming that α does not occur inside t).

In the presence of subtyping, however, the above unification scheme will not work properly: consider the above case where we analyse a conditional $\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ and have found e_1 to have type $\texttt{int} \to^{\beta_1} \alpha_1$ and have found e_2 to have type $\texttt{int} \to^{\beta_2} \alpha_2$. We shall lose precision, cf. the considerations in Sect. 3.2, if we unify β_1 with β_2 via the substitution $[\beta_2 \mapsto \beta_1]$; instead we rather create a fresh variable β and generate the constraints $\{\beta_1 \subseteq \beta, \beta_2 \subseteq \beta\}$. Our version of $\mathcal{W}$ thus follows [10] in that it *generates behaviour constraints*.

In a similar way we shall lose precision if we unify α_1 with α_2, as then α_1 and α_2 cannot later be instantiated to for example function types with different latent behaviours; instead we shall create a fresh type variable α and generate the *type constraints*[4] $\{\alpha_1 \subseteq \alpha, \alpha_2 \subseteq \alpha\}$.

The above considerations suggest the design of an algorithm $\mathcal{F}$ similar in spirit to algorithm MATCH in [6], for doing what [24] calls "forced instantiations": given a set of constraints it rewrites the type constraints and at the same time produces a substitution. A typical rewriting rule is

$$\begin{array}{ll} & \{\alpha \subseteq t_1 \to^{\beta} t_2\} \longrightarrow \{t_1 \subseteq \alpha_1,\ \beta' \subseteq \beta,\ \alpha_2 \subseteq t_2\} \\ \text{via} & \text{the substitution } [\alpha \mapsto (\alpha_1 \to^{\beta'} \alpha_2)] \\ \text{with} & \alpha_1, \beta', \alpha_2 \text{ fresh} \end{array}$$

where in addition an "occur check" is needed: clearly we cannot match α with $\texttt{int} \to^{\beta} \alpha$. However, extra variables are introduced by rules like the

[4] The presence of *type constraints*, in addition to behaviour constraints, is a consequence of our overall design: types and behaviours are inferred simultaneously from scratch. This should be compared with the approach in [26] where an effect system with subtyping but without polymorphism is presented; as the "underlying" types are given in advance it is sufficient to generate behaviour constraints.

above and as the produced substitutions are applied to the other constraints they may become "larger", thus termination is not granted and in fact a naive implementation may loop. To prevent this from happening we have adopted the loop check mechanism, and the termination proof, from [6]. In the case of successful termination of $\mathcal{F}$, all the resulting type constraints will be *atomic*, that is of form $\alpha_1 \subseteq \alpha_2$.

Below we list some typical clauses in the definition of $\mathcal{W}$; each clause produces a quadruple (S, t, b, C).

Function abstractions. A function abstraction is analysed by the algorithm fragment

$$\begin{array}{l}\mathcal{W}(A, \texttt{fn}\ x \Rightarrow e) = \\ \quad \text{let } \alpha \text{ be fresh} \\ \quad \text{let } (S, t, b, C) = W(A[x : \alpha], e) \\ \quad \text{let } \beta \text{ be fresh} \\ \quad \text{in } (S, S\,\alpha \ \rightarrow^{\beta}\ t, \varepsilon, C \cup \{b \subseteq \beta\})\end{array}$$

which generates the behaviour constraint $\{b \subseteq \beta\}$ to express the relation between the behaviour of the function body and the recorded latent behaviour.

Function applications. A function application is analysed by the algorithm fragment

$$\begin{array}{l}\mathcal{W}(A, e_1\ e_2) = \\ \quad \text{let } (S_1, t_1, b_1, C_1) = \mathcal{W}(A, e_1) \\ \quad \text{let } (S_2, t_2, b_2, C_2) = \mathcal{W}(S_1\,A, e_2) \\ \quad \text{let } \alpha, \beta \text{ be fresh} \\ \quad \text{let } (C, S_3) = \mathcal{F}(S_2\,C_1 \cup C_2 \cup \{S_2\,t_1 \subseteq t_2 \ \rightarrow^{\beta}\ \alpha\}) \\ \quad \text{in } (S_3\,S_2\,S_1, S_3\,\alpha, S_3\,(S_2\,b_1; b_2; \beta), C)\end{array}$$

where the constraint $\{S_2\,t_1 \subseteq t_2 \ \rightarrow^{\beta}\ \alpha\}$ expresses that e_1 must be a function whose argument has t_2 as a subtype; as this constraint is not atomic we have to apply $\mathcal{F}$ on the overall constraint set. (Clearly S_2 must be applied to the entities produced by the first recursive call of $\mathcal{W}$.)

Identifiers. An identifier is analysed by looking it up in the environment and taking a fresh instance of the associated type scheme. As an example, if $A(x) = \forall(\alpha_1\alpha_2\beta : \{\varepsilon \subseteq \beta\}).\ \alpha_1 \rightarrow^{\beta} \alpha_2$ then $\mathcal{W}(A, x)$ returns the quadruple

$$(\mathrm{Id}, \alpha_1' \rightarrow^{\beta'} \alpha_2', \varepsilon, \{\varepsilon \subseteq \beta'\})$$

where $\alpha_1', \alpha_2', \beta'$ are fresh variables and where Id is the identity substitution.

CML primitives and channel labels. In a similar way, a CML primitive is analysed by taking a fresh instance of its predefined type (cf. Sect. 3.3). An important case is the call $\mathcal{W}(A, \texttt{channel})$ which returns a quadruple

$$(\mathrm{Id}, \texttt{unit} \rightarrow^{\beta'} \alpha' \ \texttt{chan}\ \rho', \varepsilon, \{\alpha' \ \mathrm{CHAN}\ \rho' \subseteq \beta',\ \{l\} \subseteq \rho'\})$$

with α', β', ρ' fresh variables, and with l a *fresh label* which subsequently will be "attached" to this particular occurrence of `channel`. We thus record the "point of origin" for each channel; the i'th syntactic occurrence of `channel` (starting from zero) will be labelled i.

Definitions used polymorphically. Typing an expression `let fun` $f\ x = e_0$ `in` e `end` is a delicate matter when it comes to deciding which variables can be generalised and thus occur bound in the type scheme; we shall dispense with the details but refer to [15] for a related study (not dealing with causality).

4.1 Constraint simplification

We have seen that our algorithm $\mathcal{W}$ never unifies two variables but rather generates a constraint relating them; this is done for the sake of precision but at the price of the output becoming rather unwieldy, as one will quickly discover when implementing the algorithm. It turns out, however, that a substantial number of the generated constraints *can* be replaced by unifying substitutions without losing precision; this observation dates back to [5, 24] and we therefore equip $\mathcal{W}$ with a simplification procedure, to be called at regular intervals.

The basic idea is that a variable can be *shrinked* into its "immediate predecessor" or it can be *boosted* into its "immediate successor"; to illustrate this

consider a constraint $\alpha_1 \subseteq \alpha_2$ (behaviour or region variables are treated likewise). Now suppose that α_2 does not occur on any other right hand side; then α_2 can be shrinked, i.e. replaced by α_1 globally, provided α_2 occurs "positively" in the type t as is the case for $t = \alpha_1 \rightarrow^\beta \alpha_2$. (In this case one might alternatively boost α_1 into α_2 as α_1 occurs negatively.) Intuitively, due to the presence of subtyping the new type $\alpha_1 \rightarrow^\beta \alpha_1$ has the same information content as the old type $\alpha_1 \rightarrow^\beta \alpha_2$; thus this step does not lose precision.

5 Post-processing the Constraints

In the previous section we saw that our reconstruction algoritm $\mathcal{W}$ when applied successfully to a given program returns a quadruple (S, t, b, C); here S is of no interest (since the top-level environment contains no free variables), and t will in many cases be `unit`. What we are really interested in is the behaviour b, and the relation between the variables occurring there, as given by C; this constraint set may be quite large in spite of the simplifications mentioned in Sect. 4.1 and in this section we shall describe how to transform the constraints so as to improve readability.

There will usually only be a few type constraints, and recall that they will be of the form $\alpha_1 \subseteq \alpha_2$. There is no need to further manipulate them except that cycles may be collapsed, that is if $(\alpha \subseteq \alpha') \in C$ and also $\alpha' \subseteq \alpha$ can be deduced from C then α' may be replaced by α globally, i.e. in C as well as in b.

Region constraints will be dealt with in Sect. 5.1 where we shall see that they can be *solved* and thus completely eliminated; the intuition is that if an expression e has type `int chan` ρ and a solution maps ρ to $\{2, 7\}$ then e is an integer channel allocated by either the 2nd or the 7th syntactic occurrence of `channel`. We shall also see that the "solution of interest", R, is not necessarily the "least" solution.

The user will typically, as illustrated in Sect. 7, restrict his attention to a few selected channel labels. With L_{hid} the remaining labels, we introduce a special behaviour τ which denotes creation of, or communication over, a channel whose label belongs to L_{hid}.

With R and L_{hid} given, Sect. 5.2 lists a number of transformations that can be used to manipulate the behaviour b and the behaviour constraints C; the correctness criterion for these transformations is expressed using

bisimulations[5], as is well-known from other process algebras.

Example. Suppose $\mathcal{W}$ returns the behaviour $\beta_0; \beta_1; (\mathit{SPAWN}\,\beta_2); \beta_3$, together with the region constraints

$$(\{0\} \subseteq \rho_0),\ (\{1\} \subseteq \rho_1),\ (\rho_1 \subseteq \rho_2)$$

and the behaviour constraints

$$(\texttt{unit}\ \text{CHAN}\ \rho_0 \subseteq \beta_0),\ (\texttt{unit}\ \text{CHAN}\ \rho_1 \subseteq \beta_1),$$
$$(\rho_2\,!\,\texttt{unit}; \rho_0\,?\,\texttt{unit} \subseteq \beta_2),\ (\rho_2\,?\,\texttt{unit}; \rho_0\,!\,\texttt{unit} \subseteq \beta_3).$$

The mapping R given by $\mathsf{R}(\rho_0) = \{0\}$ and $\mathsf{R}(\rho_1) = \mathsf{R}(\rho_2) = \{1\}$ is the least solution to the region constraints, and it is possible to eliminate all behaviour constraints by "unfolding" $\beta_0, \beta_1, \beta_2, \beta_3$: in the case where $\mathsf{L}_{\text{hid}} = \{1\}$ the overall behaviour is transformed into[6]

$$\texttt{unit}\ \text{CHAN}\ \{0\}; \tau; \mathit{SPAWN}\,(\tau; \{0\}\,?\,\texttt{unit}); \tau; \{0\}\,!\,\texttt{unit}$$ □

5.1 Solving region constraints

From Sect. 4 it is apparent that the region constraints are of the form $\rho' \subseteq \rho$ or $\{l\} \subseteq \rho$; the former kind may be produced when $\mathcal{F}$ decomposes a type and the latter when analysing an occurrence of `channel`. Clearly there exists a least solution to these constraints, mapping each region variable into a set of labels, and it is computable using standard iteration techniques.

The least solution, however, is not necessarily the one of interest, as demonstrated by the program

```
let fun f ch = if ... then ch else channel () in f end
```

for which $\mathcal{W}$ will infer the type

[5] The transition relation is defined from the type system: $C \vdash b \rightarrow^a b'$ if $C \vdash a; b' \subseteq b$, and an action a is of form either $\mathit{SPAWN}\,b$ or t CHAN ρ or $\rho\,?\,t$ or $\rho\,!\,t$ or τ. Now $(b_1, C_1) \sim (b_2, C_2)$ if whenever $C_1 \vdash b_1 \rightarrow^{a_1} b_1'$ there exists a_2 and b_2' such that $C_2 \vdash b_2 \rightarrow^{a_2} b_2'$ with $(a_1, C_1) \dot{\sim} (a_2, C_2)$ as well as $(b_1', C_1) \sim (b_2', C_2)$, and vice versa; the bisimulation relation $\dot{\sim}$ on actions, among other things, formalises that τ has the intended meaning: we have e.g. that $(\rho\,!\,t, C_1) \dot{\sim} (\tau, C_2)$ if $\mathsf{R}(\rho) \subseteq \mathsf{L}_{\text{hid}}$.

[6] When printing behaviours, we often replace ρ by the value of $\mathsf{R}(\rho) \setminus \mathsf{L}_{\text{hid}}$.

$$\alpha \ \texttt{chan}\ \rho \ \rightarrow^{\beta} \ \alpha \ \texttt{chan}\ \rho$$

and also generate the constraint $\{0\} \subseteq \rho$ with 0 being the label assigned to the occurrence of `channel`. The result produced by `f` may be a channel allocated by this occurrence but it may also be a channel given as input to the program, and the latter possibility is not recorded by the least solution which maps ρ to $\{0\}$; therefore we shall rather prefer a solution which maps ρ to $\{0\} \cup \rho$.

The above can be generalised, observing that "input channels" correspond to region variables occurring negatively in the overall type: a solution should map this kind of variable into a set containing not only labels but also a meta variable (the variable itself can be used). Again it is clearly possible to compute the least such solution, to be denoted R.

5.2 Transforming behaviour, and behaviour constraints

From Sect. 4 it is apparent that a behaviour constraint is of the form $b \subseteq \beta$; it may be produced when $\mathcal{F}$ decomposes a type (in which case also b is a variable) or when a function abstraction is analysed.

A catalogue of basic transformation steps, to be iteratively performed by the post-processor until no more are applicable, is listed in the subsequent paragraphs.

Collapsing cycles. As was the case for type variables, also cycles among behaviour variables may be collapsed; this is done once and for all.

Hiding. In the case where attention is restricted to a selection of channels, that is when $\mathsf{L}_{\mathrm{hid}} \neq \emptyset$, actions affecting non-selected channels only are hidden: a behaviour t CHAN ρ is replaced by τ provided $\mathsf{R}(\rho) \subseteq \mathsf{L}_{\mathrm{hid}}$, similarly for $\rho\,?\,t$ and $\rho\,!\,t$. Also this step may be done once and for all.

Equivalence transformation. Suppose that b and b' are equivalent, that is $\emptyset \vdash b \subseteq b'$ and $\emptyset \vdash b' \subseteq b$, then b may be replaced by b' if the latter is smaller. A very frequent application is to replace $b;\varepsilon$ or $\varepsilon;b$ by b.

Unfolding. Suppose that $(b \subseteq \beta)$ is the only constraint with β on the right hand side, then this constraint can be eliminated and β globally re-

placed by b provided *(i)* β does not occur in b, and *(ii)* β does not appear inside any type (in which case replacement with a non-variable b would result in an ill-formed type).

As a preparation for this step, it may be necessary to combine a sequence of constraints $b_1 \subseteq \beta, \cdots, b_n \subseteq \beta$ into the single constraint $b_1 + \cdots + b_n \subseteq \beta$.

To prevent "code explosion", unfolding should be performed only if either *(i)* there is at most one occurrence of β to replace, or *(ii)* b is very small (for example ε).

Sharing code. For further keeping the output small it turns out to be crucial that "shared code" can be detected, as illustrated below: if the only constraint with β_1 on the right hand side has left hand side $(\{7\}\,!\,\texttt{int}; \beta_1)$, and the only constraint with β_2 on the right hand side has left hand side $(\{7\}\,!\,\texttt{int}; \beta_2)$, then β_1 can be globally replaced by β_2.

Introducing new behaviour constants. For the sake of readability, we use τ_n $(n \geq 1)$ as an abbreviation for

$$\overbrace{\tau; \cdots; \tau}^{n}$$

Similarly, τ_∞ intuitively abbreviates an unbounded number of τ-actions: if for example there is a constraint $\tau; \tau; \beta \subseteq \beta$, in effect saying that β performs two τ-action before it recurses, then β may be globally replaced by τ_∞ (again provided β does not occur inside any type).

6 User Interface

We have developed a system based on the algorithm from Sect. 4 and the post-processing tools mentioned in Sect. 5. Below we describe how a user can interact with our system, and mention the options available for tuning the output so as to suit his particular needs.

The system is accessed via the web page

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

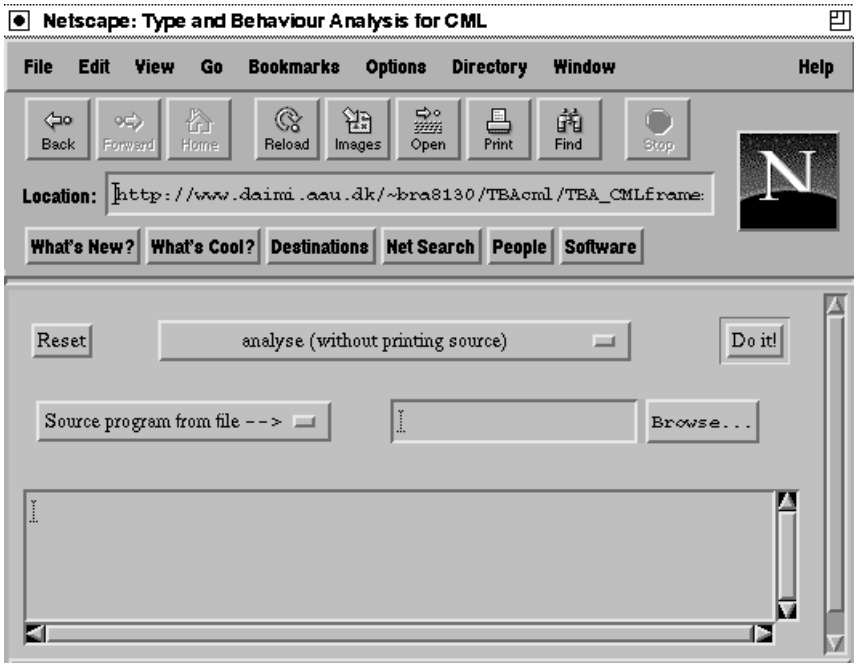

Figure 1: The initial menu.

that contains a short description of the system, including the syntax of our CML fragment, as well as a link to the system itself. The link leads to a page divided into two frames with the upper frame containing the initial menu (Fig. 1); in particular notice that the program to be analysed can be taken either from a file or from the keyboard. The system works by translating the CML fragment into a certain core subset; for debugging purposes it may be useful to see this intermediate form, hence there is an option allowing to display it.

The various features of the system are best explained by feeding an example program into the analyser; we shall use the CML program[7] listed in Figure 2 which is built from the components mentioned in Sect. 2. Its overall purpose is to control a belt in the Karlsruhe production cell and accordingly it initially declares two channels for starting and stopping the belt, two channels for testing whether there is a blank at the end, and two channels for communicating with a robot arm (placing blanks on the belt) and with a crane (removing blanks from the belt). The main function `belt2` spawns a process

[7] On "top-level", `fun f x = e ; ...` is equivalent to `let fun f x = e in ... end`.

which initially performs a double handshake with the robot so as to place one blank on the belt; then it enters the main loop `belt2_cycle` that is already explained in some detail (Sect. 2), making use of the previously defined generic functions `move_until`, `passive_sync`, and `passive_sync_event`.

When pressing the button "Do it!" the inference algorithm $\mathcal{W}$ from Sect. 4 will be applied to the program, and after a short while the lower frame will look as depicted in Figure 3: it contains

(up) a scroll-down menu for "post-processing", offering the features described in Section 5;

(left) a list of "region names", i.e. program identifiers bound to channels;

(right) a list of functions defined polymorphically.

In the subsequent subsections, each of these components will be treated in some detail.

6.1 The list of region names

In our example, as will often be the case, there is a one-to-one correspondence between program identifiers and channel labels; as an example `belt2_start` is always bound to a channel allocated by the 0th syntactic occurrence of `channel`. For labels which are in this way associated to a unique identifier, the post-processor can print that identifier, rather than printing the label itself. As an alternative option the user may choose "numbered regions" instead of "named regions" such that e.g. "3" is printed instead of "`belt2_no_blank_at_end`" (the former is shorter but the latter is far more readable).

We should mention that in general the above correspondence may be many-to-one and/or one-to-many: for an example of the former, consider the program

```
let val ch = channel () val ch1 = ch in (ch,ch1) end
```

where `ch` as well as `ch1` "will be bound to" the same channel label; for an example of the latter, consider the program

```
val ch = if ...  then channel () else channel ();
```

```
   (* command channels *)
val belt2_start    = channel () ;
val belt2_stop     = channel () ;

   (* sensor channels *)
val belt2_blank_at_end    = channel () ;
val belt2_no_blank_at_end = channel () ;

   (* synchronisation channels *)
val belt2_ready_for_arm2 = channel(): unit chan;
val belt2_ready_for_crane = channel(): unit chan;

(*** Global help-functions ***)

fun move_until wait_fun start_ch stop_ch =
        ( send(start_ch,())
        ; wait_fun ()
        ; send(stop_ch,()) );

fun passive_sync ready_ch =
        ( send(ready_ch,())
        ; accept(ready_ch) );

fun passive_sync_event ready_ch cont =
        wrap (  transmit(ready_ch,()),
                fn () => ( accept(ready_ch)
                         ; cont ()
                         )
             );

(* belt2 unit ************************************************************)

fun belt2 () =
   let fun belt2_cycle () =
       ( move_until (fn ()=>(accept(belt2_blank_at_end);
                             accept(belt2_no_blank_at_end)))
                    belt2_start
                    belt2_stop;
         select [passive_sync_event belt2_ready_for_arm2
                         (fn () => passive_sync belt2_ready_for_crane),
                 passive_sync_event belt2_ready_for_crane
                         (fn () => passive_sync belt2_ready_for_arm2)];
         belt2_cycle ())

in spawn(fn () => (passive_sync belt2_ready_for_arm2;
                   belt2_cycle ())) end;

(* starting *) belt2 ();
```

Figure 2: A belt for transporting blanks.

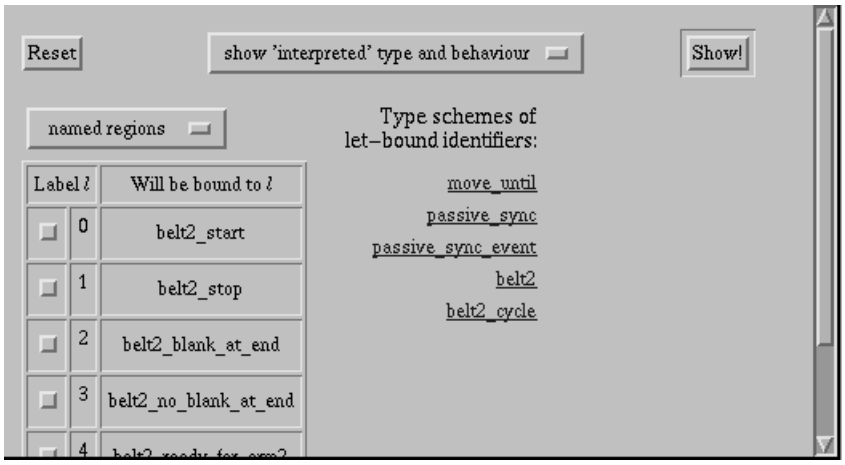

Figure 3: The "intermediate" menu.

where `ch` "*may* be bound to" the 0th occurrence of `channel` but it may also be bound to the 1st occurrence.

6.2 Post-processing

Using the scroll-down menu from Figure 3, the result produced by $\mathcal{W}$ can be examined and manipulated in various ways:

Show 'interpreted' type and behaviour: Using the methods described in Sect. 5, a solution R to the region constraints is found, and the behaviour is transformed together with the behaviour constraints. We say that a behaviour variable β can be interpreted as b, and write "β means b", if the resulting constraint set contains $(b \subseteq \beta)$ and β appears on no other right hand side.

By selecting this option the entire communication pattern is displayed: no actions are hidden, that is $\mathsf{L}_{\text{hid}} = \emptyset$.

Only see restricted channels: As above, except that some channels are hidden; the user has previously selected the channels of interest by clicking on their occurrence in the list of region names.

Print dots for hidden channels: As above, but with "..." printed instead of "τ_n" and "τ_∞" (this may improve readability).

Show 'raw' type, behaviour, constraints: This option suppresses post-processing and displays the constraint set produced by $\mathcal{W}$; for all but very small programs it will be so large that one can hardly expect to extract useful information manually.

Show time information: The total time spent by $\mathcal{W}$ is displayed, and additionally also the time spent in some of its auxiliary functions[8].

Returning to our example program in Figure 2 (which can be analysed in about 4 seconds), the result of selecting the first option is

```
Interpretation of inferred type:

unit

Interpretation of inferred behaviour:

unit chan {belt2_start};unit chan {belt2_stop};
a0 chan {belt2_blank_at_end};a1 chan {belt2_no_blank_at_end};
unit chan {belt2_ready_for_arm2};
unit chan {belt2_ready_for_crane};spawn (B0;B1)

Interpretation of type/behaviour variables:

B0 'means' {belt2_ready_for_arm2}!unit;
           {belt2_ready_for_arm2}?unit
B1 'means' {belt2_start}!unit;{belt2_blank_at_end}?a0;
           {belt2_no_blank_at_end}?a1;{belt2_stop}!unit;
           ({belt2_ready_for_crane}!unit;
            {belt2_ready_for_crane}?unit;B0
            + {belt2_ready_for_arm2}!unit;
              {belt2_ready_for_arm2}?unit;
              {belt2_ready_for_crane}!unit;
              {belt2_ready_for_crane}?unit);B1
```

We see that after the initial channel allocation the main program spawns a process which first performs `B0`, i.e. asks the robot arm to place a blank on the belt, and then iterates as indicated by `B1`, i.e. follows the code in

[8] This information is useful only for the system maintainer!

`belt2_cycle`. The latter function selects between two events which are symmetric as can be made apparent[9] by unfolding `B0`.

If we restrict our attention to `belt2_start` we get

```
Interpretation of inferred behaviour:

unit chan {belt2_start};tau5;spawn (tau2;B0)

Interpretation of type/behaviour variables:

B0 'means' {belt2_start}!unit;tau7;B0
```

reflecting that each iteration of `belt2_cycle` starts by writing on `belt2_start` and then performs 7 hidden communications.

6.3 The list of functions defined polymorphically

One can click on each of these and a post-processed version of its associated type scheme will show up in the upper frame; for `passive_sync` we get

```
unit chan R0 --B0-> unit
  where
B0 'means' R0!unit;R0?unit
```

as predicted in Sect. 3. One can follow a further link and get the "raw" (i.e. uninterpreted) version of the type scheme (reflecting its use in $\mathcal{W}$):

```
unit chan R0 --B0-> unit
  where
 B2;B3 <= B1, R1!unit <= B2, ep <= B3, R2?unit <= B4,
 ep;B1;B4 <= B0, R0 <= R1, R0 <= R2
```

7 Case Study

To investigate the usefulness of our system we apply it to a larger example: a CML program, written by a group of DAIMI students [4], that implements[10]

[9] The system may be improved so as to recognise "common subexpressions"; then *(i)* the occurrence in the code for `B1` of the code for `B0` may be replaced by `B0`, and *(ii)* a new variable `B2` may be introduced, with the interpretation `belt2_ready_for_crane!unit; belt2_ready_for_crane?unit`.

[10] There exists a simulator which visualises execution of the CML program.

the various components of the Karlsruhe production cell put forward as a benchmark in [11]. We have already (Fig. 2) seen the code for the belt transporting blanks between a robot and a crane; additionally the cell contains a press for forging the blanks, an elevating rotary table, and another belt.

We shall now validate a certain safety property: that two blanks cannot be put on top of each other on the belt. We therefore restrict our attention to the channels `belt2_start` and `belt2_ready_for_arm2` and this yields the output depicted in Figure 4; by manual elimination of irrelevant information we end up with the behaviour

```
spawn (B4;B5);spawn B2
```

and the interpretation

```
B2 'means' ...;
           ({belt2_ready_for_arm2}?unit;...;
            {belt2_ready_for_arm2}!unit;B2
            + ...;B2)
B4 'means' {belt2_ready_for_arm2}!unit;
           {belt2_ready_for_arm2}?unit
B5 'means' {belt2_start}!unit;...;
           (...;B4
            + {belt2_ready_for_arm2}!unit;
              {belt2_ready_for_arm2}?unit;...);B5
```

where clearly `B2` is the behaviour of the robot (arm). Concerning the behaviour of the conveyor belt, we see *(i)* from `B4` that it initially asks for a blank, *(ii)* from `B5` that it then repeatedly moves the belt before asking for another blank. This convinces us that the desired safety property holds.

For a more comprehensive account of how our system can be used for validating purposes, see [16] which examines another program for controlling the Karlsruhe production cell. This program is developed [22] via formal methods, but nevertheless an examination of its behaviour revealed an initialisation error.

8 Conclusion

In this paper we have described a system (available over the Internet) for extracting communication behaviours from CML programs. Clearly the

```
Interpretation of inferred behaviour:

...;unit chan {belt2_start};...;unit chan {belt2_ready_for_arm2};
...;B0;spawn B1;...;spawn B3;B0;spawn (B4;B5);spawn B2;B0

Interpretation of type/behaviour variables:

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' ...;
           ({belt2_ready_for_arm2}?unit;...;
            {belt2_ready_for_arm2}!unit;B2
            + ...;B2)
B3 'means' B6;...;B6;...;B3
B4 'means' {belt2_ready_for_arm2}!unit;
           {belt2_ready_for_arm2}?unit
B5 'means' {belt2_start}!unit;...;
           (...;B4
            + {belt2_ready_for_arm2}!unit;
              {belt2_ready_for_arm2}?unit;...);B5
B6 'means' B0;...
```

Figure 4: Checking for blank collisions.

development of such a system is an open-ended story; it would, for example, be quite useful to extend the analysis with control flow analysis [7] (so as to exploit contexts in the form of call strings) and constant propagation (for tracking constants sent over channels) etc. However, by studying a case study we demonstrated that already the present system is useful for analysing reactive systems written in CML.

The development of the system was based on a formal system for assigning behaviours to CML programs. We then transformed it into an algorithm for inferring these behaviours automatically. In a sense this is all what is needed for the theoretical development, but to present a useful tool we had to combat (1) the size of the output presented to the user, and (2) the time taken to produce that output. Sections 5 and 6 gave an overview of the rather extensive set of techniques needed in order to overcome (1). The system itself is written in Moscow ML[11] (a variant of SML) and also quite some programming, using efficient data structures, was needed to overcome

[11] The Moscow ML home page is `http://www.dina.kvl.dk/~sestoft/mosml.html`.

(2): initially the example displayed here took hours to analyse, whereas now it takes only seconds. One lesson learned is that (1) and (2) are closely related in that the techniques used to reduce the size of the output (mainly the constraints) also benefits the overall running time: indeed the "optimal" placement of the constraint simplification steps (Sect. 4.1) have required some experimentation.

We believe that the insights presented here should enable the development of similar systems, for other programming languages and other choices of relevant behaviour, and hope to explore this possibility in our future work.

Acknowledgements. We wish to thank Hans Rischel for drawing our attention to the problem of validating a CML implementation of the production cell. Kirsten L.S. Gasser developed the "front end" of our system: a parser for CML and a translator into our intermediate language. The CML program, used as a running example throughout this paper, was written by Peter Andersen, Anette Christensen, Henning Jehøj, Jacob Grydholt Jensen, Tina Olesen, and Jan-Henrik Paulsen.

This research has been supported in part by the DART (Danish Science Research Council) and LOMAPS (ESPRIT BRA 8130) projects.

References

[1] Torben Amtoft and Flemming Nielson and Hanne Riis Nielson and Jürgen Ammann: Polymorphic subtypes for effect analysis: the dynamic semantics. In *Analysis and Verification of Multiple-Agent Languages*, pages 172–206, SLNCS 1192, 1997.

[2] Torben Amtoft and Flemming Nielson and Hanne Riis Nielson: Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, May 1997.

[3] Torben Amtoft and Flemming Nielson and Hanne Riis Nielson: Polymorphic subtyping for behaviour analysis. Version 1 is available from `file://ftp.daimi.aau.dk/pub/LOMAPS/LOMAPS-DAIMI-32.ps.Z`.

[4] Peter Andersen and Anette Christensen and Henning Jehøj and Jacob Grydholt Jensen and Tina Olesen and Jan-Henrik Paulsen: Produktionsenheden i Concurrent ML. Student report, DAIMI, 1997 (in Danish).

[5] You-Chin Fuh and Prateek Mishra: Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT '89*. SLNCS 352, 1989.

[6] You-Chin Fuh and Prateek Mishra: Type inference with subtypes. *Theoretical Computer Science*, 73, 1990.

[7] Kirsten L. Solberg Gasser and Flemming Nielson and Hanne Riis Nielson: Systematic realisation of control flow analyses for CML. In *Proc. ICFP '97*, pages 39–51, 1997.

[8] David Gries: The Science of Programming. Springer Verlag, 1981.

[9] Mark P. Jones: A theory of qualified types. In *Proc. ESOP '92*, pages 287–306. SLNCS 582, 1992.

[10] Pierre Jouvelot and David K. Gifford: Algebraic reconstruction of types and effects. In *Proc. POPL'91*, pages 303–310. ACM Press, 1991.

[11] Claus Lewerentz and Thomas Lindner (editors): Formal development of reactive systems; Case study Production cell. SLNCS 891, 1995.

[12] Robin Milner: A theory of type polymorphism in programming. *Journal of Computer Systems*, 17:348–375, 1978.

[13] Robin Milner and Mads Tofte and Robert Harper: The definition of Standard ML. MIT Press, 1990.

[14] Flemming Nielson and Hanne Riis Nielson: From CML to process algebras. In *Proc. CONCUR'93*, SLNCS 715, 1993. Full version in *Theoretical Computer Science*, 155:179–219, 1996.

[15] Flemming Nielson and Hanne Riis Nielson and Torben Amtoft: Polymorphic subtypes for effect analysis: the algorithm. In *Analysis and Verification of Multiple-Agent Languages*, pages 207–243, SLNCS 1192, 1997.

[16] Hanne Riis Nielson and Torben Amtoft and Flemming Nielson: Behaviour analysis and safety conditions: a case study in CML. Manuscript, 1997.

[17] Hanne Riis Nielson and Flemming Nielson: Higher-order concurrent programs with finite communication topology. In *Proc. POPL'94*, pages 84–97. ACM Press, 1994. Full version in [19].

[18] Hanne Riis Nielson and Flemming Nielson: Static and dynamic processor allocation for higher-order concurrent languages. In *Proc. TAPSOFT'95 (FASE)*, pages 590–604, SLNCS 915, 1995.

[19] Hanne Riis Nielson and Flemming Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency: Design, Analysis, Implementation and Application* (editor: Flemming Nielson), Springer-Verlag, 1996.

[20] Hanne Riis Nielson and Flemming Nielson and Torben Amtoft: Polymorphic subtypes for effect analysis: the static semantics. In *Analysis and Verification of Multiple-Agent Languages*, pages 141–171, SLNCS 1192, 1997.

[21] John H. Reppy: Concurrent ML: Design, application and semantics. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198. SLNCS 693, 1993.

[22] Hans Rischel and Hongyan Sun: Design and prototyping of real-time systems using CSP and CML. *Euro-micro Real Time Workshop*, 1997.

[23] J.A. Robinson: A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[24] Geoffrey S. Smith: Polymorphic inference with overloading and subtyping. In *Proc. TAPSOFT '93*, SLNCS 668, 1993. Also see: Principal type schemes for functional programs with overloading and subtyping: *Science of Computer Programming* 23, pp. 197–226, 1994.

[25] Jean-Pierre Talpin and Pierre Jouvelot: The type and effect discipline. *Information and Computation*, 111, 1994. (A preliminary version appeared in *Proc. LICS '92*, pages 162–173.)

[26] Yan-Mei Tang: Control flow analysis by effect systems and abstract interpretation. PhD thesis, Ecoles des Mines de Paris, 1994.

[27] Bent Thomsen, Lone Leth and Tsung-Min Kuo: FACILE—from Toy to Tool. In *ML with Concurrency: Design, Analysis, Implementation and Application* (editor: Flemming Nielson), Springer-Verlag, 1996.