Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Behaviour Analysis for Validating Communication Patterns

Torben Amtoft, Hanne Riis Nielson, Flemming Nielson

DAIMI, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
e-mail: {tamtoft,hrn,fn}@daimi.aau.dk

The date of receipt and acceptance will be inserted by the editor

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 that 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 programs.

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. An impressive application is [33] where an annotated type system is used for efficient stack implementation of a call-by-value functional language; this provides a main theoretical basis for the ML Kit[1].

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) [25] that extends Standard ML (SML) [14] with concurrency, and Facile [32] 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* [21,1], a kind of process algebra terms based on [17]; they are related to "effects" [30] but augment these in conveying causality information.

To illustrate how the causal nature of behaviours is used to show the absence of certain bugs in concurrent systems, consider the following safety criterion: a machine M must not be started until the temperature has reached a certain level. Assuming that there is a channel `start_M` such that M is started by sending a signal over this channel, and that there is a channel `temp_OK` such that a thermometer sends a signal over this channel when the temperature reaches the appropriate level, this criterion amounts 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`. As we shall see, our system will be able to supply simplified behaviours of the form displayed for b.

[1] The ML Kit home page is `http://www.diku.dk/research-groups/topps/activities/kit2/index.html`.

The system. The paper reports on a tool for behaviour analysis; 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 manually validate certain safety criteria.

The system is written in Moscow ML[2], a variant of SML. This language is convenient as it supports modular programming via separate compilation, and as it contains support[3] for a web interface: from Moscow ML code one can create `cgi` scripts and there are functions for reading the contents of `html`-forms.

Theoretical foundations. The basis for our system is an algorithm that 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. [13, 15, 16]);
3. demonstrated that the algorithm is sound wrt. the inference system, i.e. that its output represents a valid inference.

The rather complex development is covered[4] in [2] which also establishes a completeness result, stating that the inferred behaviours are in a certain sense principal.

Analysing behaviours. As demonstrated in [21, 22] there are several other applications of behaviour information (such as testing for finite communication topology); thus a variety of analyses may be put on top of our system. Since a behaviour abstracts the program from which it is derived, the result of analysing the former yields correct information about the latter. One might even devise a tool for testing certain properties of behaviours, à la the Mobility Workbench [34] which operates on π-calculus expressions.

A case study. The presentation in this paper will be based on CML programs for the Karlsruhe production cell [12] that has been put forward as a benchmark for designing and validating concurrent systems. The overall purpose of the cell is to process "blanks" (pieces of metal) in a press; its various components are shown a bird's eye view of on Fig. 1 which is a picture taken from a simulator, and at the bottom of the figure a supply of blanks is depicted. The blanks enter the system on the feed belt depicted just above and are then transfered one at a time to a rotating table; the table is then elevated and rotated such that one of the two robot arms can take the blank and place it in the press. After the blank has been forged by the press, the other robot arm will take it out and deliver it to a deposit belt (the top one on Fig. 1). For testing purposes a crane has been added to move the blanks from the deposit belt back to the feed belt.

Overview. In Section 2 we give a brief overview of the programming language CML and its type system. 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. Section 5 outlines how the system, guided by the user, manipulates and partially solves these constraints so as to improve readability. The user interface is described in Section 6. The use of the system for validating a program implementing the Karlsruhe production cell is presented in Section 7; a number of safety properties can in fact be validated while the validation of other properties would require information not presently included in the behaviours. We present our concluding remarks, including some notes on implementation issues, in Section 8.

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 short tutorial to features common for SML and CML.

Functional features. SML [14] 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" [13], that is if it is

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

[3] ©Jonas Barklund, Computing Science Dept., Uppsala University, 1996.

[4] For a core subset of CML; to handle functions (such as `wrap`) that manipulate events (cf. Sect. 2) we need a non-trivial reformulation of the semantics, similar to what is done in [23].

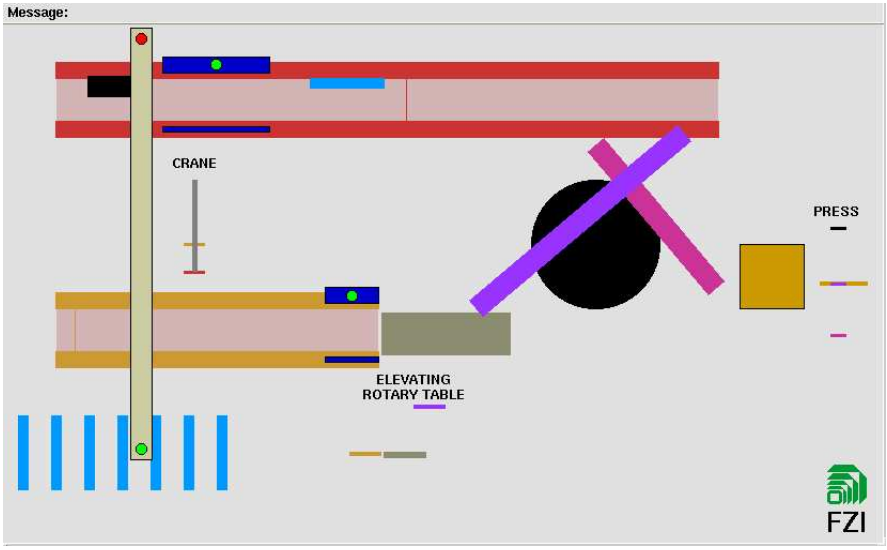

Fig. 1. The Karlsruhe Production Cell.

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 [25] 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 are useful; as a simple first example of this consider the function below which makes use of the sequencing operator ";" well-known from many imperative languages for executing one statement before another. 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. 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,())
        )
```

Events. A more complex scenario is when a passive process wants to interact with *two* active processes in a "first come first served" manner. This can be expressed by the pseudo-code (in the style of [9])

```
if ready ch1 → passive_sync ch1; passive_sync ch2
[] ready ch2 → passive_sync ch2; passive_sync ch1
fi
```

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", which means that they can be passed around just like integers.

To *create* events, CML offers the primitive `transmit` which is a "non-committing" version of `send`; in a similar way the primitive `receive` is a non-committing version of `accept`.

The CML primitive `sync` *synchronises* an event, i.e. turns it into an actual communication. Consider the event created by evaluating the expression `transmit(ch1,())`: when `sync` is applied to this event, the value `()` is sent over the channel `ch1` provided some other process is ready to receive from `ch1`, otherwise the operation blocks. (We have the equations `send` x = `sync` (`transmit` x) and `accept` x = `sync` (`receive` x).)

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.

An attempt to achieve part of the desired protocol is then to write

```
select [transmit(ch1,()), transmit(ch2,())]
```

but we must ensure that if `select` synchronises the first event (created by `transmit(ch1,())`) then the continuation will be to execute

```
accept(ch1); passive_sync ch2
```

and similarly if `select` chooses the second event. We thus need 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 protocol can be achieved by writing

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

A more succinct way of expressing this is

```
select [passive_sync_event ch1
              fn () => passive_sync ch2,
        passive_sync_event ch2
              fn () => passive_sync ch1]
```

where we have used a new generic function for creating "handshake events":

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

The function `passive_sync_event` returns an event which

- will only be synchronised if some other process is ready to receive on the channel `ready_ch`;
- if synchronised it 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` (α will typically be `unit`).

During our presentation we have introduced a number of building blocks; in a subsequent chapter we shall see (Fig. 5) how these can be combined into a somewhat larger program.

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. (We use β to range over behaviour variables and ρ to range over region variables, cf. the use of type variables α.) The annotation b in a function type $t_1 \rightarrow^b t_2$ 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 b in an event type of the form t `event` b describes the behaviour taking place whenever the event is synchronised; finally the annotation r in a channel type t `chan` r describes the region in which the channel is allocated.

A region explicitly denotes a set of program points corresponding to occurrences of `channel`, and implicitly denotes the set of channels created from these points. Thus our intention is that region information should enable us to distinguish between channels allocated by different syntactic occurrences of `channel`, but we do not aim at distinguishing between channels allocated by different invocations[5] of the same syntactic occurrence.

Example 1. The function `move_until` (introduced in Section 2) can be assigned the annotated type

$$(\texttt{unit} \rightarrow^{\beta} \alpha_0) \rightarrow \texttt{unit chan}\ \rho_0 \rightarrow \texttt{unit chan}\ \rho_1 \rightarrow^{b} \texttt{unit}$$

which reflects that `move_until` does not perform any action until it has been supplied with *three* arguments. Here b denotes the behaviour

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

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;

[5] See e.g. [5] for a methodology to achieve this level of detail.

– then it calls f with the argument `()`, since f has type `unit` $\rightarrow^{\beta}$ α_0 this will behave as indicated by β;
– finally it sends `()` over the channel ch_1, located in region ρ_1. □

The notion of annotated types[6] goes way back in the literature; a classic example being the *effects* of [30]. In the present paper, a behaviour b is either

– a variable β;
– 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 r (a channel, able to transmit values of type t, is allocated in region r);
– $r\,?\,t$ (a value of type t is read from a channel located in region r);
– $r\,!\,t$ (a value of type t is written to a channel located in region r).

Example 2. The function `passive_sync` can be assigned the type

$$\texttt{unit chan}\ \rho\ \rightarrow^{b}\ \texttt{unit}$$

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

and its cousin `passive_sync_event` can be assigned the type

$$\texttt{unit chan}\ \rho\ \rightarrow\ (\texttt{unit}\ \rightarrow^{\beta}\ \alpha)\ \rightarrow\ \alpha\ \texttt{event}\ b$$

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

Here β is the behaviour of `cont` representing the rest of the computation.

3.1 Inference system

We now present some of the rules given in [2] for assigning annotated types to CML expressions; as in [24] (which was in turn inspired by [28, 10]) *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.

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

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

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

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

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

where the behaviour $b_1; b_2; b$ 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

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

where the body e is analysed in an environment binding x to t_x, with its behaviour b 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: if a program can be assigned a type t in the latter system then it can also be assigned an annotated type t', where the "underlying" type of t' is t, in the former system; and vice versa. But when examining the above rules this might seem not to be the case, as witnessed by the situation below.

Suppose we want to type a conditional `if` e_0 `then` f_1 `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 e_1 is of the form $\texttt{fn}\ x_1 \Rightarrow e_1'$ and e_2 is of the form $\texttt{fn}\ x_2 \Rightarrow e_2'$. Further suppose that we have judgements

$$C, A[x_1 : \texttt{int}] \vdash e_1' : \texttt{int} \,\&\, b_1 \text{ and}$$
$$C, A[x_2 : \texttt{int}] \vdash e_2' : \texttt{int} \,\&\, b_2$$

with $b_1 \neq b_2$. We can thus assign e_1 and hence f_1 the type $t_1 = \texttt{int}\ \rightarrow^{b_1}\ \texttt{int}$, and we can assign e_2 and hence f_2 the type $t_2 = \texttt{int}\ \rightarrow^{b_2}\ \texttt{int}$; but in order to use the rule for conditional we must be able to assign f_1 and f_2 a common type $t = \texttt{int}\ \rightarrow^{b_{12}}\ \texttt{int}$. This can be achieved using either *subeffecting* or *subtyping* as explained below.

Subeffecting. One approach is to find b_{12} such that

$C, A[x_1 : \texttt{int}] \vdash e_1' : \texttt{int} \,\&\, b_{12}$ and
$C, A[x_2 : \texttt{int}] \vdash e_2' : \texttt{int} \,\&\, b_{12}$

for then e_1 and e_2, and hence also f_1 and f_2, can be assigned the type $t = \texttt{int} \rightarrow^{b_{12}} \texttt{int}$. These judgements can be obtained provided it is possible from C to deduce that $b_1 \subseteq b_{12}$ and $b_2 \subseteq b_{12}$; this will for instance be the case if $b_{12} = b_1 + b_2$ or if b_{12} is a variable β with $(b_1 \subseteq \beta)$ and $(b_2 \subseteq \beta)$ contained in C. Formally, this is due to the subeffecting rule

if $C, A \vdash e : t \,\&\, b$ and $C \vdash b \subseteq b'$
then $C, A \vdash e : t \,\&\, b'$

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[7] that any action performed by b_1 can also be performed by b_2.

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. $\subseteq$; it is beyond the scope of this paper to repeat the formal details.

Subtyping. The price to pay for using subeffecting only is that then *all* occurrences of f_1 in e are indistinguishable from f_2, and vice versa. In order to increase the precision of the analysis we may use subtyping, cf. the considerations in [31, Chap. 5]. This method 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

if $C, A \vdash e : t \,\&\, b$ and $C \vdash t \subseteq t'$
then $C, A \vdash e : t' \,\&\, b$

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. [28] we do not have any ordering on base types, such as $\texttt{int} \subseteq \texttt{real}$. As is to be expected, the ordering is contravariant in the argument position of a function type:

if $C \vdash t_1' \subseteq t_1$ and $C \vdash b \subseteq b'$ and $C \vdash t_2 \subseteq t_2'$
then $C \vdash t_1 \rightarrow^{b} t_2 \subseteq t_1' \rightarrow^{b'} t_2'$

Of particular interest is the rule for channel types:

if $C \vdash t \subseteq t'$ *and* $C \vdash t' \subseteq t$
and $C \vdash r \subseteq r'$
then $C \vdash t \,\texttt{chan}\, r \subseteq t' \,\texttt{chan}\, r'$

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

[7] A similar claim is formalised in [23] 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.

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 [28], 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 $\forall(\vec{\alpha\beta\rho} : C).\ t$.

CML primitives. Closed type schemes have been preassigned to all the primitives, of which we list a few in Table 1. 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 as we do not allow polymorphic recursion). These considerations are reflected in the rule for typing such definitions:

if $C \cup C_0, A[f : t_0][x : t_1] \vdash e_0 : t_2 \,\&\, b_0$
with $t_0 = t_1 \rightarrow^{b_0} t_2$
and $C, A[f : \forall(\vec{\alpha\beta\rho} : C_0).\, t_0] \vdash e : t \,\&\, b$
then $C, A \vdash \texttt{let fun}\ f\ x = e_0\ \texttt{in}\ e\ \texttt{end} : t \,\&\, b$
provided $\cdots$

This rule can only be applied provided certain side conditions hold but for reasons of space we dispense with the precise details. The most familiar of these state that the bound variables $\vec{\alpha\beta\rho}$ 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, as witnessed by the related approach in [3]; one of the basic ideas is that we cannot generalise variables free in the behaviour (cf. [30]). It is beyond the scope of this paper to explain the details of the side conditions.

4 Inference Algorithm

We shall aim at constructing a type reconstruction algorithm in the spirit of Milner's algorithm $\mathcal{W}$ [13]: 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 [13] employs unification [27]: 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 `bool` and t_1 with t_2. Unification works by decomposition: in order to unify $t_1 \rightarrow t_2$ and $t_1' \rightarrow t_2'$ one recursively unifies t_1

Table 1. Types schemes of CML primitives.

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

with t_1' and t_2 with t_2'; and to unify a variable α with a type $t \neq \alpha$ one produces the substitution $[\alpha \mapsto t]$, provided that α does not occur in t — this "occur-check" is needed to avoid generation of infinite types.

The above unification scheme cannot directly be extended to work for annotated types, as it is in general not possible to unify behaviours since these do not constitute a "free algebra": for example, $b_1; b_2$ may be equivalent to $b_1'; b_2'$ without b_1 being equivalent to b_1' (take $b_1 = b_2' = \varepsilon$ and $b_2 = b_1' = \rho\,!\,\texttt{int}$). As in [29] the remedy is to ensure that the reconstruction algorithm operates on so-called *simple types*[8] only: these are types where all behaviour and region annotations are variables.

It is now straightforward to extend unification to work for annotated *simple* types but in the presence of subtyping we can do better: 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 [11] 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*[9] $\{\alpha_1 \subseteq \alpha, \alpha_2 \subseteq \alpha\}$.

The above considerations suggest the design of an algorithm $\mathcal{F}$ similar in spirit to algorithm MATCH in [7], for doing what [28] 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 we perform an occur-check: α must not occur in t_1 or t_2. However, extra variables are introduced by rules like the 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 [7]. 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 $\mathcal{W}$ given in [2]; each clause produces a quadruple (S, t, b, C).

Function abstractions. A function abstraction is analysed as follows:

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

which generates the behaviour constraint $\{b \subseteq \beta\}$ to express the relation between the behaviour of the function body and the recorded latent behaviour (which must be a variable).

Function applications. A function application is analysed as follows:

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

where the constraint $\{S_2\,t_1 \subseteq t_2 \to^{\beta} \alpha\}$ 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. (As in [13], S_2 must be applied to the entities produced by the first recursive call of $\mathcal{W}$.)

[8] The development in [2], including the inference system, considers *simple* types only.

[9] 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 [31] 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.

Identifiers and CML primitives. 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. 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

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

with α', β', ρ' fresh variables, and with l the label denoting the program point of this particular occurrence of `channel` (cf. Sect. 3).

Definitions used polymorphically. Typing `let` expressions 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 [19] 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 [6, 28] and we therefore equip $\mathcal{W}$ with a simplification procedure which is called at regular intervals.

The basic idea is that a variable can be (using the terminology of [28]) *shrunk* 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 shrunk, 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, that is all types $t_1 \rightarrow^\beta t_2$ with $t_1 \subseteq t_2$, as the old type $\alpha_1 \rightarrow^\beta \alpha_2$; this correctness property is formalised in [2].

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 therefore the user is given the option to let the system transform these constraints. To further improve readability the user may additionally choose to restrict his attention to a few selected channel labels; for that purpose we introduce a special behaviour τ which denotes creation of, or communication over, a "hidden" channel — that is a channel located in the set $\mathsf{L}_{\mathsf{hid}}$ of labels *not* selected. In this section we shall describe the transformation tools employed by the system.

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

Region constraints will be dealt with in Sect. 5.1 where we shall see that there exists a mapping R which solves them; hence they can be completely eliminated.

With R and $\mathsf{L}_{\mathsf{hid}}$ 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 [2] using bisimulations[10], as is well-known from other process algebras.

Example 3. 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

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

The mapping R given by $\mathsf{R}(\rho_0) = \{0\}$ and $\mathsf{R}(\rho_1) = \mathsf{R}(\rho_2) = \{1\}$ solves 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 $\mathsf{L}_{\mathsf{hid}} = \{1\}$ the overall behaviour is transformed into

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

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

[10] 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 $(\rho\,!\,t, C_1) \dot{\sim} (\tau, C_1)$ if $\mathsf{R}(\rho) \subseteq \mathsf{L}_{\mathsf{hid}}$, and that $(\tau, C_1) \dot{\sim} (\tau, C_2)$. (The definition of $\dot{\sim}$ given in [2, Chap.6] is slightly flawed and must be modified according to the guidelines from `http://www.daimi.aau.dk/~tamtoft/erratPB529.html`.)

analysing an occurrence of `channel`. A solution must map each region variable into a region; as mentioned in Sect. 3 a region is a set of program points but to cope with "open systems" we also have to allow for "external" program points. To see this, consider the program

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

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

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

and also generate the constraint $\{0\} \subseteq \rho$. The result produced by `f` may be a channel allocated by the occurrence of `channel` but it may also be a channel given as input to the program, through `ch` which has type α `chan` ρ. It therefore seems reasonable to implicitly add the constraint $r_\rho \subseteq \rho$ where r_ρ is an "input region" corresponding to the variable ρ; in general such a constraint must be added for each region variable occurring negatively in the overall type.

Clearly there exists a least solution, denoted R, to these augmented constraints; and it is computable using standard iteration techniques. In the example above, $\mathsf{R}(\rho) = \{0\} \cup r_\rho$ which corresponds to our intuition[11].

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 $\mathsf{L}_{\mathsf{hid}} \neq \emptyset$, actions affecting non-selected channels only are hidden: a behaviour t CHAN ρ is replaced by τ provided $\mathsf{R}(\rho) \subseteq \mathsf{L}_{\mathsf{hid}}$, 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 syntactically 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 replaced 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 a type that is not *simple*).

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. It turns out to be crucial that "shared code" can be detected, as otherwise the size of the output will often come close to the size of the source program. An instance of the technique is 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 and the former constraint thus removed.

Introducing new behaviour constants. For the sake of readability[12], we use τ_n as an abbreviation for a sequence of at most n τ-actions, that is

$$\tau_n \equiv \overbrace{\tau;\cdots;\tau}^{n} + \overbrace{\tau;\cdots;\tau}^{n-1} + \cdots + \tau + \varepsilon$$

Similarly, τ_∞ abbreviates a potentially 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

Our system is 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 the 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`

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; the program to be analysed can be taken either from a file or from the keyboard and is submitted by clicking on "Do it!". The system works by translating the CML fragment into a certain core subset; for debugging purposes (but not for the casual user) 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, as done in the following sections.

[11] Instead of printing r_ρ, the system will print just ρ.

[12] The development in [2] does not support this transformation.

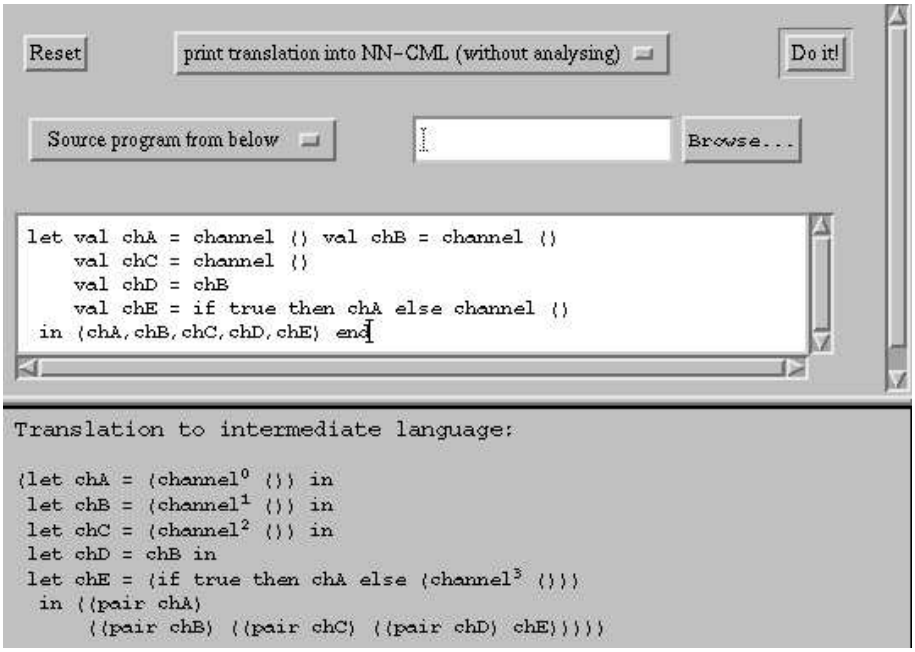

Fig. 2. A source program and its intermediate form.

6.1 A toy example

Consider the CML program listed in the upper frame of Fig. 2; it returns a tuple of newly allocated channels some of which may be identical.

In Fig. 2 we have chosen the option "print translation…", hence the intermediate form (cf. above) of the source program is shown in the lower frame. The most interesting feature is that all occurrences of `channel` have been labelled so as to record their program points (cf. Sect. 3); moreover some program constructs have been transformed into simpler constructs.

Next we instead choose the default option "analyse …" in the upper frame, yielding the lower frame depicted in Fig. 3 containing information about the relation between program identifiers and channel labels. The system is able to infer that the identifier `chA` will always be bound to a channel located in the region containing channel label 0; and similarly the system assigns exact regions to `chB`, `chC`, and `chD`. Concerning the value of `chE`, the system does not attempt to predict the outcome of tests (even though it would be trivial in this case) and hence has to approximate its region to $\{0, 3\}$.

So far we have not obtained any type or behaviour information; to do so we click on "Show!" and obtain the upper frame depicted[13] in Fig. 4. As expected the overall type of the program is a 5-tuple; the first component of this tuple (`chA`) is located in region $\{0\}$ and the third component (`chC`) is located in region $\{2\}$. But note that instead of printing "$\{2\}$" the system prints[14] "`chC`", since Fig. 3 reveals that this identifier may become bound to no other channel labels and no other identifier may become bound to this channel label. In this way it becomes much easier for the user to grasp the output produced by the system.

6.2 Another example

We shall use the CML program[15] listed in Fig. 5, written by a group of DAIMI students [4]; it is built from the components mentioned in Sect. 2 and in particular employs the generic functions `move_until`, `passive_sync`, and `passive_sync_event`. Its overall purpose is to control the deposit belt in the Karlsruhe production cell (cf. the Introduction) 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 the robot arm (placing blanks on the belt) and with the crane (removing blanks from the belt). The main function `belt2` spawns a process 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`. The invariant of this loop 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

[13] Also some time measurement is given, useful for the system maintainer only!

[14] There exists an alternative option "numbered regions" in which case "$\{2\}$" will be printed instead of "`chC`"; this yields a more uniform representation in the case where there is not a one-to-one correspondence between program identifiers and channel labels.

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

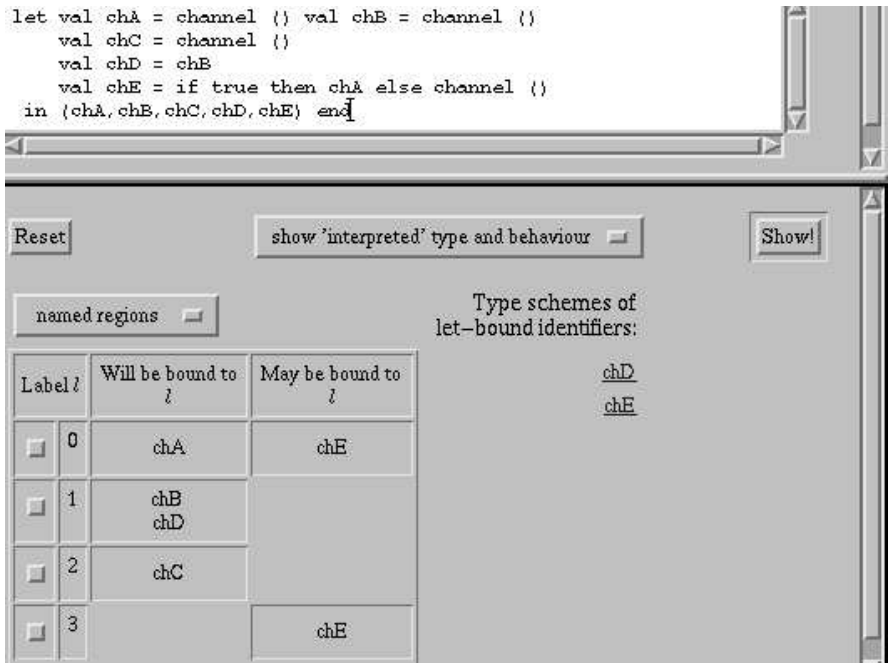

Fig. 3. The relation between program identifiers and channel labels.

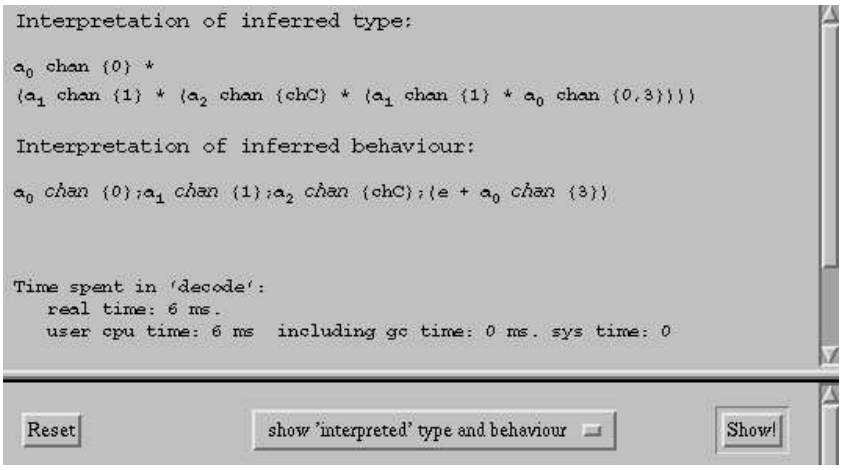

Fig. 4. Type and behaviour information.

using `move_until`. As the robot may become ready before the crane, we can follow the recipe from the end of Sect. 2 and express step 2 and 3 in terms of `passive_sync_event`.

When this program is submitted to the system, it takes about 4 seconds to run the inference algorithm $\mathcal{W}$ from Sect. 4 and the lower frame menu depicted in Fig. 6 appears.

On the right there is a list of the functions defined polymorphically[16]; one can click on each of these and its associated type scheme[17] 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.

We shall now focus on the pull-down menu on top of the lower frame which enables us to examine and manipulate the result produced by $\mathcal{W}$ in various ways, making use of the methods described in Sect. 5. The system prints "β means b" if the post-processed constraint set contains $(b \subseteq \beta)$ and β appears on no other right hand side.

Show 'interpreted' type and behaviour. This is the default option and is a rather verbose way of presentation, as the entire communication pattern is displayed: no actions are hidden, that is $\mathsf{L}_{\mathsf{hid}} = \emptyset$. Selecting this option yields an upper frame whose contents are reproduced in Fig. 7.

[16] Rather: those entities defined via `let` and with a body different from `channel ()`.

[17] Actually a post-processed version; one can follow a further link and get the "raw" (i.e. not post-processed) version of the type scheme (reflecting its use in $\mathcal{W}$).

```
   (* 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 ();
```

Fig. 5. The deposit belt (for transporting blanks).

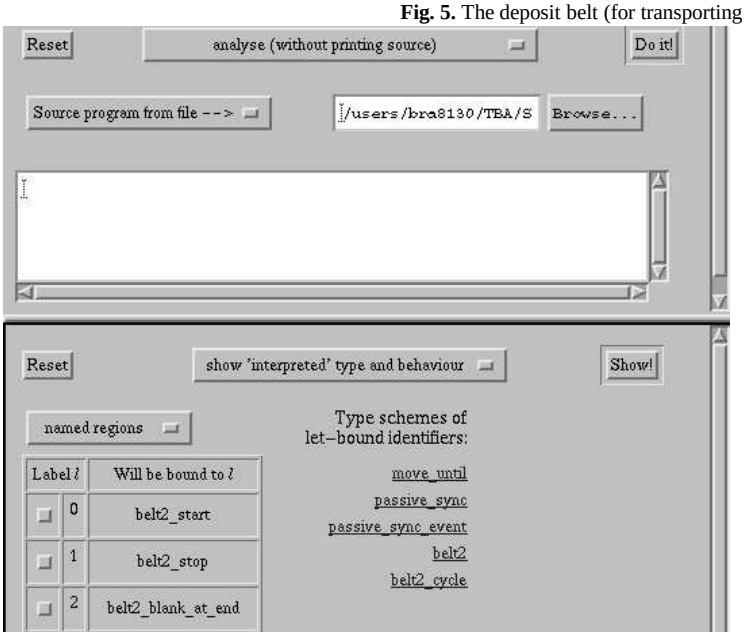

Fig. 6. Having analysed the program from Fig. 5.

```
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
```

Fig. 7. The entire communication pattern.

We see that after the initial channel allocations 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 `belt2_cycle`. The latter function selects between two events which are symmetric as can be made apparent by unfolding `B0`.

Only see restricted channels. Before using this option the user must select the channels of interest by clicking on their occurrence in the list of channel labels.

Fig. 8 depicts[18] the result of restricting our attention to `belt2_start`, and reflects that each iteration of `belt2_cycle` starts by writing on `belt2_start` and then performs 7 hidden communications.

Print dots for hidden channels. This is basically as the previous option, but to further improve readability "..." is printed instead of "τ_n" and "τ_∞".

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; the latter information is useful only for the system maintainer!

[18] The system uses `fork` for the behavior *SPAWN* .

7 Case Study

To investigate the usefulness of our system we apply it to a larger example: a CML program, the derivation of which is described in [26], that implements the Karlsruhe production cell put forward as a benchmark in [12]. The system can analyse this 488 lines program in less than four minutes and then various safety properties can be tested by selecting appropriate channels; in the following we look at three such properties and examine whether the system will enable us to validate or falsify them. For a more comprehensive account of how the system can be used for validation purposes, see [20].

7.1 A property that can be validated

We want to ensure that two blanks cannot be put on top of each other on the deposit belt. For this validation we rely on some assumptions (which may be validated subsequently) about the environment: that the robot only drops a blank on the belt after having successfully sent a request[19] on the channel `arm2_transmit_ready`.

We therefore restrict our attention to the channels `belt2_start` and `arm2_transmit_ready` and this yields the output depicted in Fig. 9; where clearly `B3` is the behaviour of the robot (arm) and `B2` is the behaviour of the belt. By unfolding `B4` we end up with the following interpretation of `B2`

```
B2 'means' {arm2_transmit_ready}?unit;...;
           {belt2_start}!unit;...;B5
B5 'means' ...;B2 +
```

[19] The protocol for communicating between the belt and the robot thus differs slightly from the one used in Sect. 6.2 (and Fig. 5).

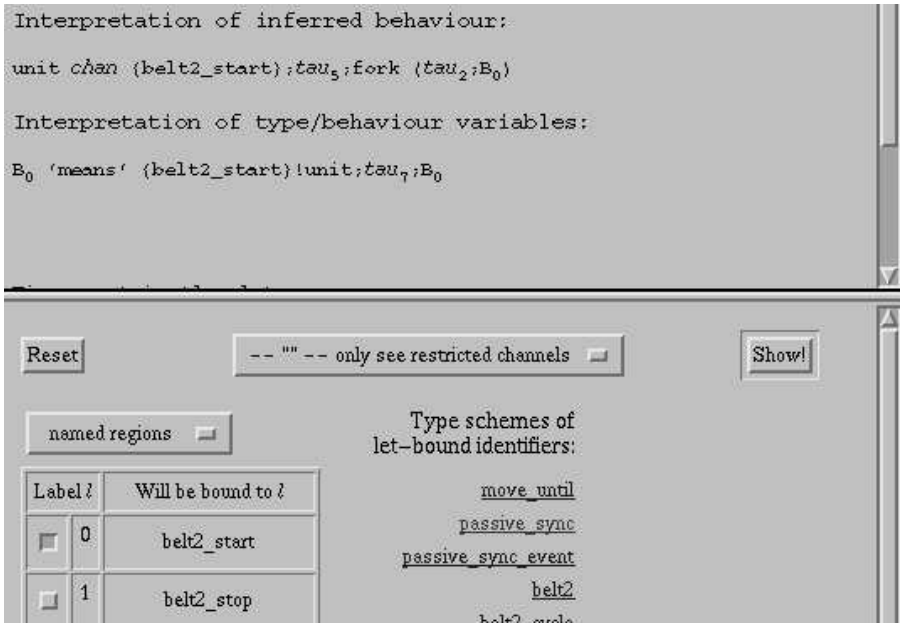

Fig. 8. Focusing on `belt2_start`.

```
{arm2_transmit_ready}?unit;...;
{belt2_start}!unit;...;B5
```

from which we see that the deposit belt initially asks for a blank and then repeatedly moves before asking for another blank. This convinces us that the desired safety property holds.

7.2 A property that turns out not to hold

We want to ensure that the elevating (rotating) table is not moved upwards if it is in its upper position, and that is not moved downwards if it is in its lower position.

We therefore restrict our attention to the channels `table_upward` and `table_downward` as all vertical movements of the table are initiated via those channels, and to the sensor channels `table_is_bottom`, `table_is_top`, `table_is_not_bottom`, and `table_is_not_top`. This yields the output depicted in Fig. 10, where `B3` is the behaviour of the table.

Thus all communications on `table_downward` are preceeded by a communication on `table_is_top`; and by unfolding `B3` is is 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 it to be preceeded by a communication on any sensor channel. Hence our system has been used to demonstrate that the above safety property does *not* hold, even though the program is developed [26] via formal methods! 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 systems and as a new area for applying program analysis technology.

7.3 A property than cannot be decided by the system

We now want to ensure that the (elevating) rotating table is alternating between being rotated clockwise and counterclockwise. For that purpose we restrict our attention to the channels `table_left`, `table_right`, and `table_stop_h` as all horizontal movements of the table are controlled by those channels, and to the sensor channel `table_angle` which yields information about the horizontal position of the table. This yields the output depicted in Fig. 11, where again `B3` is the behaviour of the table.

Unfortunately, this behaviour enables us neither to validate nor to disprove the desired property: `B3` is a recursive behaviour which calls `B6` twice, and each invocation of `B6` may move the table in either direction.

To see the reason for this shortcoming, we take a closer look at the source code which implements the horizontal table movement and whose behaviour is given by `B6`:

```
turn_to(a) =
  let val x = accept(table_angle) in
  if x < a then send(table_right,()) ...
  else if x > a then send(table_left,()) ...
```

The problem is that our system does not incorporate any information about values of variables such as `a` and about the entities communicated over channels such as `table_angle`. Such information would allow for different "versions" of `B6`, where there for each version will be only one possible branch as the others can be pruned.

This example also shows that our analysis in sensible to programming style, in that it performs better if the user writes separate procedures for left movement respectively right movement. This naturally suggests ways to increase the power of the system.

```
Interpretation of inferred behaviour:

...;unit chan {belt2_start};...;unit chan {arm2_transmit_ready};
...;B0;B0;B0;spawn B1;spawn B2;B0;spawn B3

Interpretation of type/behaviour variables:

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' {arm2_transmit_ready}?unit;...;B4;B5
B3 'means' ...;B6
B4 'means' {belt2_start}!unit;...
B5 'means' ...;B2 + {arm2_transmit_ready}?unit;...;B4;B5
B6 'means' ...;B7
B7 'means' ...;
           (...;B8
            + ...;(...;B8 + ...;{arm2_transmit_ready}!unit;...);
              B3)
B8 'means' ...;
           ({arm2_transmit_ready}!unit;...;B6
            + ...;{arm2_transmit_ready}!unit;...;B7)
```

Fig. 9. Checking for blank collisions.

```
Interpretation of inferred behaviour:

...;unit chan {table_upward};...;unit chan {table_downward};...;
a0 chan {table_is_bottom};a1 chan {table_is_not_bottom};
a2 chan {table_is_top};a3 chan {table_is_not_top};...;B0;B0;B0;
spawn B1;spawn B2;spawn B3;spawn B4

Interpretation of type/behaviour variables:

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' ...;B5
B3 'means' ...;{table_upward}!unit;{table_is_top}?a2;...;
           {table_downward}!unit;{table_is_bottom}?a0;...;B3
B4 'means' ...;B6
B5 'means' ...;B2 + ...;B5
B6 'means' ...;B7
B7 'means' ...;(...;B8 + ...;(... + ...;B8);B4)
B8 'means' ...;(...;B6 + ...;B7)
```

Fig. 10. Checking for excessive vertical table movements.

8 Conclusion

We have described a system[20] for extracting communication behaviours from CML programs. Future developments may incorporate control flow analysis [8, 18] (so as to exploit contexts in the form of call strings) and constant propagation (for tracking constants sent over channels) in order to increase the power of the system; for further reducing the output size it would be helpful to add recognition of common subexpressions[21].

For more traditional reasoning about behaviours one might consider developing an automated tool (like [34] for the π-calculus).

Already the present system is useful for analysing reactive systems written in CML. We strongly believe that the insights presented here should enable the development of similar systems, for other programming languages and other choices of behaviour. In this way, the current research opens up a new avenue for the use of program analysis for validation of software [20].

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

[21] In Fig. 7 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`.

```
Interpretation of inferred behaviour:

...;unit chan {table_left};unit chan {table_stop_h};
unit chan {table_right};...;int chan {table_angle};...;B0;B0;B0;
spawn B1;spawn B2;spawn B3;spawn B4

Interpretation of type/behaviour variables:

B0 'means' spawn ...
B1 'means' ...;B1
B2 'means' ...;B5
B3 'means' ...;B6;...;B6;B3
B4 'means' ...;B7
B5 'means' ...;B2 + ...;B5
B6 'means' {table_angle}?int;
           (e
            + {table_left}!unit;B8;{table_stop_h}!unit
            + {table_right}!unit;B8;{table_stop_h}!unit)
B7 'means' ...;B9
B8 'means' ...;{table_angle}?int;(e + B8)
B9 'means' ...;(...;B10 + ...;(... + ...;B10);B4)
B10 'means' ...;(...;B7 + ...;B9)
```

Fig. 11. Checking for alternating horizontal table movements.

Implementation issues. 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 4.1 (simplification during analysis) and 5 (post-processing) gave an overview of the rather extensive set of techniques needed in order to overcome (1); by performing constraint simplification regularly (the "optimal" placement has required some experimentation) this naturally also contributed to reducing (2).

But to overcome (2), quite some programming using efficient data structures was needed: initially the program in Sect. 7 was too large for our system and to analyse subparts took hours, whereas now the whole program takes only a couple of minutes. A main challenge was to ensure that certain operations on variables could be performed efficiently, including "shrink" and "boost" (cf. Sect 4.1) and various "closures" (for dealing with polymorphism). To achieve this goal we devised a graph representation for constraints, making use of the imperative features of Moscow ML (such as arrays); it also turned out to be advantageous to switch between various representations according to the operation in question.

Acknowledgements. We wish to thank Hans Rischel [26] for drawing our attention to the problem of validating a CML implementation of the production cell. We would also like to thank Peter Sestoft for inspiring us to make a web-interfaced system, and for answering questions about Moscow ML. Kirsten L.S. Gasser developed the "front end" of our system: a parser for CML and a translator into our intermediate language. The code in Fig. 5, used to illustrate several features of CML and of our system, is part of a larger program written by Peter Andersen, Anette Christensen, Henning Jehøj, Jacob Grydholt Jensen, Tina Olesen, and Jan-Henrik Paulsen [4].

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: Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, May 1997.
2. Torben Amtoft and Flemming Nielson and Hanne Riis Nielson: Polymorphic subtyping for side effects. Technical report DAIMI PB-529, 1997.
3. Torben Amtoft and Flemming Nielson and Hanne Riis Nielson and Jürgen Ammann: Polymorphic subtypes for effect analysis: the dynamic semantics. In *Analysis and Verifi cation of Multiple-Agent Languages*, pages 172–206, SLNCS 1192, 1997.
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. Christopher Colby: Determining storage properties of sequential and concurrent programs with assignment and structured data. In *Proc. SAS'95*, pages 64–81, SLNCS 983, 1995.
6. You-Chin Fuh and Prateek Mishra: Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. TAPSOFT '89*, pages 167–183, SLNCS 352, 1989.
7. You-Chin Fuh and Prateek Mishra: Type inference with subtypes. *Theoretical Computer Science*, 73:155–175, 1990.
8. Kirsten L. Solberg Gasser and Flemming Nielson and Hanne Riis Nielson: Systematic realisation of control flow analyses for CML. In *Proc. ICFP '97*, pages 38–51. ACM Press, 1997.
9. David Gries: The Science of Programming. Springer Verlag, 1981.

10. Mark P. Jones: A theory of qualified types. In *Proc. ESOP'92*, pages 287–306, SLNCS 582, 1992.
11. Pierre Jouvelot and David K. Gifford: Algebraic reconstruction of types and effects. In *Proc. POPL'91*, pages 303–310. ACM Press, 1991.
12. Claus Lewerentz and Thomas Lindner (editors): Formal development of reactive systems; Case study Production cell. SLNCS 891, 1995.
13. Robin Milner: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
14. Robin Milner and Mads Tofte and Robert Harper: The definition of Standard ML. MIT Press, 1990.
15. Flemming Nielson: A formal type system for comparing partial evaluators. In *Proc. Partial Evaluation and Mixed Computation* (editor: D. Bjørner and A. P. Ershov and N. D. Jones), pages 349–384, North-Holland, 1988.
16. Flemming Nielson: The typed λ-calculus with first-class processes. In *Proc. PARLE'89*, pages 357–373, SLNCS 366, 1989.
17. 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.
18. Flemming Nielson and Hanne Riis Nielson: Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proc. POPL'97*, pages 332–345. ACM Press, 1997.
19. 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.
20. Hanne Riis Nielson and Torben Amtoft and Flemming Nielson: Behaviour analysis and safety conditions: a case study in CML. Technical report DAIMI PB-528, 1997. To appear in proceedings of FASE'98.
21. Hanne Riis Nielson and Flemming Nielson: Higher-order concurrent programs with finite communication topology. In *Proc. POPL'94*, pages 84–97. ACM Press, 1994.
22. 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.
23. 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, Monographs in Computer Science, 1997.
24. 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.
25. John H. Reppy: Concurrent ML: Design, application and semantics. In *Proc. Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198, SLNCS 693, 1993.
26. Hans Rischel and Hongyan Sun: Design and prototyping of real-time systems using CSP and CML. In *Proc. 9th Euromicro Workshop on Real-Time Systems, Toledo, Spain*, pages 121–127. IEEE Computer Society Press, 1997.
27. J.A. Robinson: A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
28. Geoffrey S. Smith: Polymorphic type inference with overloading and subtyping. In *Proc. TAPSOFT '93*, pages 671–685, SLNCS 668, 1993. Also see: Principal type schemes for functional programs with overloading and subtyping: *Science of Computer Programming* 23, pp. 197–226, 1994.
29. Jean-Pierre Talpin and Pierre Jouvelot: Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
30. 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.)
31. Yan-Mei Tang: Control flow analysis by effect systems and abstract interpretation. PhD thesis, Ecoles des Mines de Paris, 1994.
32. 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, Monographs in Computer Science, 1997.
33. Mads Tofte and Jean-Pierre Talpin: Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. POPL'94*, pages 188–201. ACM Press, 1994.
34. Björn Victor and Faron Moller: The Mobility Workbench —A Tool for the π-calculus. In *Proc. CAV'94: Computer Aided Verification*, pages 428–440, SLNCS 818, 1994.