From CML to Process Algebras

(Extended Abstract)

Flemming Nielson, Hanne Riis Nielson
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
e-mail: {fnielson, hrnielson}@daimi.aau.dk

Abstract

Reppy’s language CML extends Standard ML of Milner et al. with primitives for communication. It thus inherits a notion of strong polymorphic typing and may be equipped with a structural operational semantics. As a first step we formulate an effect system for statically expressing the communication behaviours of CML programs as these are not reflected in the types. As a second step we adapt the structural operational semantics of CML so as to incorporate behaviours. We then show how types and behaviours evolve in the course of computation: types may decrease and behaviours may loose alternatives as well as decrease. As the syntax of behaviours is rather similar to that of a process algebra our main result may therefore be viewed as regarding the semantics of a process algebra as an abstraction of the semantics of an underlying programming language. This establishes a new kind of connection between “realistic” concurrent programming languages and “theoretical” process algebras.

1 Introduction

One trend in the research on process algebras is to extend them with “higher-order” features somewhat analogous to the “higher-order” role that functions play in functional languages. Some approaches allow passing labels or ports, e.g. [9], whereas others allow passing processes, e.g. [17, 18]. Sometimes this leads to hybrid calculi that contain the syntax of a process algebra as well as that of the \( \lambda \)-calculus, e.g. [3, 10, 6]. Putting more emphasis on the functional features, another approach is to extend a “realistic” functional language with primitives for communication. Good examples include CML [14, 15, 9] and Facile [4] but also Concurrent Clean [13] may be viewed in this way. We refer to [7] for a much more detailed survey of some of these issues.

We follow the latter approach and base ourselves on Reppy’s language CML. It
is an extension of Standard ML with primitives for communication; among other things this allows channels to be created and processes to be forked and then processes may send and receive values over channels. Since CML is an extension of Standard ML it inherits a notion of strong typing. However, the types are very close to those of Standard ML and therefore do not contain much information about the communication that takes place during computation. Following [10, 6] we believe that it is desirable with some type-like “formula” that gives a concise summary of the possible communication behaviours. Our approach deviates from [10, 6] in separating the type and communication information by using the notion of effect system previously developed for functional languages, e.g. [8, 16]. Section 2 gives a presentation of this system.

Both [15] and [1] give a structural operational semantics for CML. As is usual the types do not influence the semantics but for the purpose of proofs it may be desirable to label the transition relation with additional book-keeping details (and to retain some type information in the expression). The main difference between [15] and [1] is that the latter is a traditional operational semantics whereas the former uses the notion of “evaluation context” in order to present the rules more concisely and in order to facilitate proofs. In Section 3 we present a definition close to that of [15] but with additional book-keeping details; in keeping with tradition the types and behaviours do not influence the semantics.

The impact of the operational semantics on types and behaviours emerges when showing “subject reduction” and related results. Actually, types may decrease in the course of computation and this phenomenon also arose in [2] in the context of modelling object-oriented programming. In a similar way the behaviours may decrease in the course of computation but additionally certain alternatives may disappear due to the choices made during computation. It is instructive to regard this combined decreasing and disappearance of behaviours as an operational semantics for behaviours. Since behaviours syntactically resemble process algebras (e.g. the one in [5]) this suggests the viewpoint that the semantics of a process algebra is an abstraction of the semantics of an underlying programming language. This is quite unlike previous attempts to relate languages like CML to process algebras where programs from CML are directly translated into primitives of some given process algebra or vice versa. Section 4 provides the precise formulations of the results we have to offer.

We finish with prospects for future research and concluding remarks in Section 5. The full version is available as [12] and contains further discussions, examples, variations on the type system presented here and full proofs.

2 CML with Behaviours of Communication

We follow [15, 1] in embedding the essential features of CML into a small fragment of Standard ML. For simplicity we restrict the attention to a monomorphic fragment and we take care to structure the syntax in a way that facilitates adding
new constructs as the need arises.

The syntax of expressions \( e \in \text{Exp} \) and weakly evaluated expressions \( w \in \text{WExp} \) is given by:

\[
\begin{align*}
  e & ::= w \mid e_1 \; e_2 \mid \text{let } i = e_1 \text{ in } e_2 \mid \text{rec } i_0 : e \Rightarrow e \\
  \text{if } e \text{ then } e_1 \text{ else } e_2 & ::= t \\
  w & ::= c : t \mid i \mid \text{fn } i : t \Rightarrow e \mid \cdots
\end{align*}
\]

They are defined by mutual recursion and include among other things constants with an explicit monotype, identifiers \( i \in \text{Ident} \) (unspecified), and \text{let-} abstraction without any polymorphism.

The syntax of constants \( c \in \text{Const} \) is given by:

\[
\begin{align*}
  c & ::= () \mid \text{true} \mid \text{false} \mid n \\
  & \mid \text{pair} \mid \text{fst} \mid \text{snd} \mid \text{nil} \mid \text{cons} \mid \text{hd} \mid \text{tl} \mid \text{ismil} \\
  & \mid \text{send} \mid \text{receive} \mid \text{choose} \mid \text{noevent} \mid \text{wrap} \mid \text{sync} \mid \text{fork} \mid \text{channel}
\end{align*}
\]

The concurrency primitives allow to send values over channels, receive values over channels, choose between a list of computations (and writing \text{noevent} for \text{choose nil}), and modify a value that is communicated by applying a function to it; actually, these primitives construct “delayed” communications that may be enacted using synchronization. Finally, we may fork a new process to the pool of processes and we may allocate a new free channel for communication.

For types \( t \in \text{Type} \) we take:

\[
\begin{align*}
  t & ::= \text{unit} \mid \text{bool} \mid \text{int} \mid t v \mid t_1 \times t_2 \mid \text{list} \mid t_1 \rightarrow t_2 \mid \text{chan } r \\
  & \mid \text{com } b
\end{align*}
\]

We have type variables \( t v \in \text{TyVar} \) (e.g. \( \tau, \tau', \tau_1 \)) and for functions we use a superscript behaviour \( b \in \text{Beh} \) for indicating the communication that will take place when the function is executed. Much as in CML we have a type for channels over which values of a given type may be communicated. To allow some separation among the identity of channels we indicate the specific region where the channel is allocated. A region will describe a non-empty set of “program points”. For regions \( r \in \text{Reg} \) we take:

\[
\begin{align*}
  r & ::= i \mid r_1 + r_2 \mid rv
\end{align*}
\]

and we shall occasionally need region variables \( rv \in \text{RegVar} \) (e.g. \( \rho, \rho', \rho_1 \)).

Also as in CML we have a type for a “delayed” communication yielding a result of a certain type; unlike CML we have added a behaviour for indicating the communication that will take place when the “delayed” communication is enacted.

Finally, behaviours \( b \in \text{Beh} \) are given by:
The behaviours include primitive constructs for describing “no communication”,
sending a value of some type over a channel allocated in a certain region, receiving
a value, allocating a channel, and forking a new process of a given type and
with a given behaviour when executed. We use semi-colon to express that one
behaviour takes place before another and we use plus to express that either the
first behaviour takes place or the second does. For recursive functions we need
a behaviour REC br.b for expressing a behaviour that is as given by b provided
that recursive calls are as given by bv ∈ BehVar (e.g. β,β',β1).

Well-typing

We shall say that an expression e has type t and behaviour b, written tenv ⊨ e | t & b, whenever the type of e is t in the usual sense and evaluation of e gives rise
to the communication behaviour b. As usual tenv is a type environment, i.e. a
finite list of pairs of identifiers and types, giving the types of free variables; since
CML is an eager language there is no effect associated with accessing an identifier
and therefore the type environment does not contain any behaviour component
(except embedded within the types). For constants our syntax prescribes an
explicit monotype to be given; we use the polytypes of Figure 1 to restrict the
decision of monotypes. Only three primitives involve functions with a non-trivial
behaviour: sync for enacting a “delayed” computation, fork for forking a new
process and channel for allocating a new channel.

The details of the type inference for expressions are given by the axioms and rules
of Figure 2. For function abstraction the resulting type and behaviour indicate
that no communication takes place when constructing the function abstraction
but only when the function is executed. For application the overall behaviour
expresses eager left-to-right evaluation. We do not require equality between the
type of the actual parameter and the type of the formal parameter but merely
that the type of the actual parameter is a sub-type of the type of the formal
parameter. As illustrated in [12] this is useful for allowing a function expressing
mild restrictions on the argument, e.g. that it only communicates over channels
in certain regions, to be applied to a concrete argument with a very specific
communication behaviour. The rule for recursive functions is much as the rule
for function abstraction except that we need to extend the type environment
with assumptions about the recursive function and we only require the type and
behaviour of the body to be sub-types and sub-behaviours of the corresponding
parts of the assumptions. Finally, the rule for conditional allows the types of the
branches to be dissimilar and only requires them to be sub-types of a common
type. To require equality would invalidate the subject reduction property proved
in Section 4.

Fact (Unique Typing) If tenv ⊨ e | t1 & b1 and tenv ⊨ e | t2 & b2 then t1 = t2
and b1 = b2. □
Sub-typing

Since types involve regions as well as behaviours the sub-typing relation must involve a sub-region relation and a sub-behaviour relation. These relations may be defined by axioms and inference rules and have some important similarities (as well as important differences). To save repetition and to help demonstrating that they constitute the “right” collection we shall organize their presentation with diligence.

We begin with regions. Intuitively, \( r_1 \leq r_2 \) is to mean that the set of identifiers
\[ \leq \text{ is a preorder:} \quad r \leq r \quad \frac{r_1 \leq r_2 \quad r_2 \leq r_3}{r_1 \leq r_3} \]

\[ \equiv \text{ is the associated equivalence:} \quad \frac{r_1 \leq r_2 \quad r_2 \leq r_1 \quad r_1 \equiv r_2 \quad r_2 \leq r_1}{r_1 \equiv r_2} \]

structural rule with polarity \( \oplus \oplus \): \quad \frac{r_1 \leq r_1^+ \quad r_2 \leq r_2^+}{r_1 + r_2 \leq r_1^+ + r_2^+}

\( + \text{ is join:} \quad r_1 \leq r_1 + r_2 \quad r_2 \leq r_1 + r_2 \quad r \equiv r + r \)

Figure 3: Coercion Rules for Regions

\[ \leq \text{ is a preorder and} \quad \equiv \text{ is the associated equivalence} \]

structural rules with polarities \( \oplus \times \oplus, \oplus \text{ list}, \oplus \rightarrow \oplus, \oplus \text{ chan}, \oplus \text{ com} \oplus \)

\[ \text{e.g.} \quad \frac{t_1^- \leq t_1 \quad b \leq b^+ \quad t_2 \leq t_2^+}{t_1 \rightarrow b \quad t_2 \leq t_1^+ + t_2^+} \quad \frac{t \equiv t' \quad r \leq r^+}{t \text{ chan} \quad r \leq t' \text{ chan} \quad r^+} \]

Figure 4: Coercion Rules for Types

listed in \( r_1 \) is a subset of those listed in \( r_2 \). Formally, this may be axiomatized as shown in Figure 3. The first 5 axioms and rules simply state that \( \leq \) is a preorder and that \( \equiv \) is the associated equivalence. The last 4 axioms and rules state that \( + \) is a least upper bound operator (modulo the equivalence). The notion of polarity is explained below.

Turning to types we once more need to state that \( \leq \) is a preorder and \( \equiv \) is the associated equivalence. The details of this are as for regions and are therefore not repeated in Figure 4. Next comes a structural rule for each type constructor. To summarize these succinctly we use the notion of polarity. There are three polarities: \( \oplus \) for a covariant or monotonic position, \( \ominus \) for a contravariant or antimonic position and \( \ominus \) for a mixed co- and contravariant position. The examples given in Figures 3 and 4 should make the intention clear (or see [12]). The definitions are in good accord with the literature on sub-typing.

Many of the rules and axioms for behaviours in Figure 5 follow the pattern seen already. On top of this we have distribution laws for ‘+’, and for ‘;’ we have an associative law and two axioms stating that \( \odot \) is a neutral element. For recursion we have axioms for a \( \alpha \)-conversion, a one-level unfolding and a simple structural rule.

3 Dynamic Semantics of CML

We now present a structural operational semantics for the eager left-to-right evaluation of CML. The formulation is close in spirit to [15] but some differences
≤ is a preorder and ∎ is the associated equivalence
structural rules with polarities ⊕ ∎, ⊕ ? ∎, ⊕ chan ∎, ⊕ fork ∎, ⊕ ; ∎, ⊕ + ∎
+ is join
distribution laws over + for ⊕ ! ∎, ⊕ ? ∎, ⊕ chan ∎, ⊕ ; ∎, ⊕ h ∎
e.g. \((r_1 + r_2)!t \equiv (r_1)!t + (r_2)!t\) \((h_1 + h_2) \equiv (h_1) + (h_2)\)
; and ∎ constitute a monoid (modulo the equivalence)
\(b_1; (b_2; b_3) \equiv (b_1; b_2); b_3\) \(k; ∎ \equiv ∎\) \(ε; ∎ \equiv ∎\)
rules for recursion
\[
\begin{align*}
\text{REC } b, b \equiv & \text{ REC } b', b[bv \mapsto b'] & \text{ where } bv' \notin FV(b) \\
\text{REC } b, b \equiv & \text{ REC } b[bv \mapsto (∖ \text{ REC } b, b)] \\
& \equiv b' \\
\end{align*}
\]

Figure 5: Coercion Rules for Behaviours

include the treatment of δ-reduction and the choice of book-keeping details.

Sequential evaluation

We begin with the sequential evaluation of expressions. This encompasses all features of CML except the channel, fork and sync primitives; these were the primitives listed in Figure 1 that did not have an ε-behaviour associated with the function space. The definition of the transition relation is given in Figure 6. A central concept is that of an evaluation context \(E\). It may be defined inductively by:

\[
E ::= [] \mid E \varepsilon \mid w E \mid \text{let } i = E \text{ in } ε \mid \text{if } E \text{ then } ε_1 \text{ else } ε_2 : t
\]

Here [] is an empty context or a “hole”; so in general \(E\) describes an expression with precisely one hole in it. We then write \(E[ε]\) for the expression that is like \(E\) except that the hole is replaced by \(ε\). The definition of \(E\) is responsible for enforcing the eager left-to-right evaluation.

Most of the axioms of Figure 6 are now straightforward. The final axiom describes the δ-reductions for the primitive constructs of CML. The details are listed in Figure 7. To record the piecemeal evaluation of constants, as in the intended reduction sequence
Most of the out that we deviate from in not requiring stuck configurations. It may be defined by (situation and instead introduce a new set of constants for characterizing the dynamically stuck configurations. It may be defined by (3+true) that should have been caught by the type system and (hd nil) that cannot be expected to be caught by any decidable type system. Alternatively, one could mask the dynamically stuck configurations using non-termination, e.g., to impose (hd: t, nil, hd: t nil) ∈ δ_−; this is essentially the approach of [11, Chapter 6].
Concurrent evaluation

The transition relation for concurrent evaluation is given in Figure 8. Configurations have the form \texttt{cent}, \texttt{PP} where \texttt{cent} is a channel environment and \texttt{PP} is a process pool. More precisely, a process pool \texttt{PP} is a partial function from process identifiers \(pi \in \texttt{PIdent}\) (e.g. \texttt{p-0}, \texttt{p-1},...) to the expression residing there. When writing a process pool \texttt{PP}' in the form \(PP[p_i_1 \leftarrow e_1] \ldots [p_i_n \leftarrow e_n]\) we take it for granted that all of \(\text{dom}(PP), \{p_i_1\}, \ldots, \{p_i_n\}\) are mutually disjoint. The channel environment \texttt{cent} is much like the type environment and so associates channel identifiers \(ci \in \texttt{CIdent}\) (e.g. \texttt{c-0}, \texttt{c-1},...) with the type of values that may be communicated over the channel. We assume that the sets \texttt{Ident, PIdent} and \texttt{CIdent} are mutually disjoint. The fact that we use a channel environment rather than just a set of previously allocated channels, is an example of the book-keeping details present in the semantics.

The first axiom embeds sequential evaluation within concurrent evaluation. For book-keeping purposes the transition relation is labelled with the process executing and an indication of the communication behaviour; this will be useful in formulating the results of the next section. Next we have axioms for those primitives of Figure 1 that were not dealt with in the definition of sequential evaluation. For channel allocation we use the channel environment to make sure that we do not re-allocate an already allocated channel. To record the allocation the channel environment is extended; for book-keeping purposes it turns out to be helpful for the next section that also the type is recorded and we do this by means of the channel environment. The third axiom deals with process creation and is rather similar in spirit to the axiom for channel creation. The fourth axiom takes care of communication among different processes. The formulation makes use of a transition system for expressing when two "delayed" communications match and for calculating the respective outcomes as well as indications of the communication behaviour.
4 Deriving a Process Algebra from CML

We now show to which extent the types and behaviours are preserved or modified in the course of computation.

Sequential Correctness

It is natural to restrict the attention to closed expressions, i.e. expressions with no free identifiers, because the definition of evaluation context is such that we never pass inside the scope of any defining occurrence for identifiers. However, we will have to allow that the expressions include channel identifiers that have been
allocated in previous concurrent transitions. So we shall regard an expression \( e \) as being closed when \( cenv \vdash e \mid t \& b \) for some \( cenv, t \) and \( b \).

**Proposition 4.1** If \( cenv \vdash e \mid t \& b \) and \( e \rightarrow e' \) then there exists \( t^- \leq t \) and \( b^- \leq b \) such that \( cenv \vdash e \mid t^- \& b^- \). \( \square \)

It may be instructive to demonstrate why it would be too demanding to require that \( t^- = t \) or \( b^- = b \). For types suppose that \( t^- < t \) is given and that \( c : t^- \) is a constant; then \( (\text{fn } x : t \Rightarrow x)(c : t^-) \) has type \( t \) but it evaluates to \( c : t^- \) that has type \( t^- \). For behaviours simply note that if \text{true} then \( e_1 \) else \( e_2 \) has behaviour \( e_1 ; (b_1 + b_2) \) and that it evaluates to \( e_1 \) that has behaviour \( b_1 \).

The proof is by induction on the inference \( e \rightarrow e' \) [12].

**Matching Correctness**

The transition relation for concurrent evaluation utilizes the transition relations for sequential evaluation and for matching. It is therefore convenient to formulate the correctness of matching before considering the correctness of concurrent evaluation.

**Proposition 4.2** If \( cenv \vdash w_1 \mid (t_{b_1} \com b_{0_1}) \& e, cenv \vdash w_2 \mid (t_{b_2} \com b_{0_2}) \& e \) and \( (w_1, w_2) \rightsquigarrow (e_1, e_2) : (b_1, b_2) \) then there exists \( t_{b_1}' \leq t_{b_1} \) and \( t_{b_2}' \leq t_{b_2} \) such that

\[

cenv \vdash e_1 \mid t_{b_1}' \& b'_{0_1} \text{ with } b_1 ; b'_1 \leq b_{0_1} \\
cenv \vdash e_2 \mid t_{b_2}' \& b'_{0_2} \text{ with } b_2 ; b'_2 \leq b_{0_2}
\]

and where \( t_{b_1}' \leq t_{b_1} \) and \( t_{b_2}' \leq t_{b_2} \).
Furthermore, one of \( b_1 \) and \( b_2 \) may be written \( r_1 t_1 \) and the other \( r_2 t_2 \) where \( t_1 \equiv t_2 \) and \( r_1 \) and \( r_2 \) have a lower bound, i.e. \( \exists r_0 : r_0 \leq r_1 \land r_0 \leq r_2 \). \( \square \)

The proof is by induction on the transition relation for matching [12].

**Concurrent Correctness**

So far we have not extended the notion of well-typing to the configurations of the concurrent transition relation and our first task is to remedy this. To this end we shall need a partial function \( PT \) of process types: it maps process identifiers \( \pi \in \mathbf{PIdent} \) to types. Similarly, we shall need a partial function \( PB \) of process behaviours: it maps process identifiers to behaviours. Intuitively, a process pool \( PP \) is correct with respect to \( PT \) and \( PB \) if each process, \( PP(\pi) \), has type and behaviour given by \( PT(\pi) \) and \( PB(\pi) \), respectively.

Formally, the correctness of \( PP \) with respect to \( PT \) and \( PB \) is written \( \vdash \mathit{cenv}, \mathit{PP} \mid PT \& PB \) and is given by

\[
\begin{align*}
\text{dom}(PP) &= \text{dom}(PT) \land \\
\forall \pi i \in \text{dom}(PP) : cenv \vdash PP(\pi i) \mid PT(\pi i) \& PB(\pi i)
\end{align*}
\]

Our main result about concurrent evaluation is the following proposition that gives information about the evolution of types and behaviours. A concise formulation requires some additional notation. We allow writing \( \tilde{b} \) for \( b \) as well as \( b_1, b_2 \) and similarly \( \tilde{\pi i} \) for \( \pi i \) as well as \( \pi i_1, \pi i_2 \). When writing \( \{\tilde{\pi i}\} \) this then stands for \( \{\pi i\} \) or \( \{\pi i_1, \pi i_2\} \), respectively. When \( P \) is a partial function from process identifiers we write \( P \{\tilde{\pi i}\} \) for the restriction \( P \{\text{dom}(P) \backslash \{\tilde{\pi i}\}\} \) of \( P \) to the subset \( \text{dom}(P) \backslash \{\tilde{\pi i}\} \) of \( \text{dom}(P) \). This notation applies to process pools, process types and process behaviours. For process types \( PT \) and \( PT' \) we write

\[
PT'[\tilde{\pi i}] \leq PT[\tilde{\pi i}]
\]

for \( \{\tilde{\pi i}\} \subseteq \text{dom}(PT') \land \forall \pi i \in \{\tilde{\pi i}\} \cap \text{dom}(PT) : PT'(\pi i) \leq PT(\pi i) \). This takes care of the situation where new processes are created.

**Proposition 4.3** If \( \vdash \mathit{cenv}, \mathit{PP} \mid PT \& PB \) and \( \mathit{cenv}, \mathit{PP} \Rightarrow \tilde{\pi i} \mathit{cenv}' \), \( PP' \) then there exists \( PT' \) and \( PB' \) such that

- if \( \tilde{b} = t_0 \mathsf{chan} i_0 \) then \( \mathit{cenv}' = \mathit{cenv}[ci \mapsto t_0 \mathsf{chan} i_0] \) for some \( ci \notin \text{dom}(\mathit{cenv}) \); otherwise \( \mathit{cenv}' = \mathit{cenv} \),
- if \( \tilde{b} = t_0 \mathsf{fork} k_0 \) and \( \tilde{\pi i} = \pi i_1, \pi i_2 \) then \( PT'(\pi i_2) \leq t_0 \),
- \( PP' \{\tilde{\pi i}\} = PP \{\tilde{\pi i}\} \),
- \( PT' \{\tilde{\pi i}\} = PT \{\tilde{\pi i}\} \) and \( PT'[\tilde{\pi i}] \leq PT[\tilde{\pi i}] \),
\[ PB' \setminus \{ \tilde{p}_i \} = PB' \setminus \{ \tilde{p}_i \} \]
as well as \[ \vdash \text{conv}', PP' | PT' \& PB'. \]

The proof is by cases on the rule used for the concurrent transition [12].

**Process Algebras**

The statement of Proposition 4.3 (as opposed to its proof) does not convey much information about the relationship between \( PB[\tilde{p}_i] \), \( \tilde{b} \) and \( PB'[\tilde{p}_i] \). This will be rectified now and our main tools will be two transition relations: one for the evolution of individual behaviours and one for the evolution of process behaviours.

The transition relation for individual behaviours takes the form \( b_1 \xrightarrow{a} b_2 \) and says that the behaviour \( b_1 \in \text{Beh} \) evolves to \( b_2 \in \text{Beh} \) while performing the action \( a \). It is possible to identify actions and behaviours, i.e. to use \( a \in \text{Beh} \), but it may be more informative to be more restrictive. To this end we define actions \( a \in \text{Act} \) by:

\[
a := \epsilon | r^1 | r^2 | t \text{chan} | t \text{fork} b
\]

The details of the transition system are given in Figure 10. The first axiom simply records the effect of performing an individual action. Then we have a rule that allows evolution of actions to take place in more elaborate contexts. The next rule is patterned after a structural rule

\[
\frac{b_1 \equiv b'_1 \quad b'_1 \xrightarrow{a} b'_2 \quad b'_2 \equiv b_2}{b_1 \xrightarrow{a} b_2}
\]

as might be found in the \( \pi \)-calculus [9]. However, because of our use of subtyping we find that we need a stronger rule and to obtain this we replace \( \equiv \) by \( \xrightarrow{\cdot} \) and add three more axioms. The first says that \( \equiv \) is contained in \( \xrightarrow{\cdot} \) and the final two allow to discard possible behaviours. (Actually \( b_1 + b_2 \xrightarrow{\cdot} b_2 \) is derivable from the remaining axioms and rules.)

**Lemma 4.4** The statement \( b_1 \xrightarrow{a} b_2 \) is equivalent to \( a; b_2 \leq b_1 \). \( \Box \)

The transition relation for process behaviours takes the form \( PB \xrightarrow{\tilde{p}_i} P B' \) and says that the process behaviour \( PB \) evolves to the process behaviour \( PB' \). Regarding process behaviours as a process algebra this transition relation then gives the operational semantics of terms in the process algebra. The details of the transition system are given in Figure 11 and make use of the transition relation for individual behaviours.

**Theorem 4.5** The statement of Proposition 4.3 may be extended with the following condition:
\[ a \leftarrow^* \epsilon \]
\[ b_1 \leftarrow^a b_2 \]
\[ b_1; b \leftarrow^* b_3; b \]
\[ b_1 \leftarrow^* b'_1 \quad b'_1 \leftarrow^a b'_2 \quad b'_2 \leftarrow^* b_2 \]
\[ b \leftarrow^* b' \quad \text{if } b' \equiv b \]
\[ b_1 + b_2 \leftarrow^* b_1 \]
\[ b_1 + b_2 \leftarrow^* b_2 \]

Figure 10: Evolution of Behaviours

\[ \begin{align*}
    b \leftarrow^* \ell' \\
    PB[p_i \mapsto b] & \implies^*_{PB} PB[p_i \mapsto b'] \\
    b \leftarrow^* \text{CHAN} \ell' \\
    PB[p_i \mapsto b] & \implies^*_{PB} PB[p_i \mapsto b'] \\
    b \leftarrow^* \text{FORK}b_0 \ell' \\
    PB[p_i \mapsto b] & \implies^*_{PB} PB[p_i \mapsto \ell'[p'] \mapsto b'] \\
    b_j \leftarrow^* \ell_{1:j}, \ \ell'_j \\
    b_{3-j} \leftarrow^* \ell_{4:j}, \ell'_{3-j} \\
    t_1 \equiv t_2 \\
    \exists r_0 : r_1 \geq r_0 \leq r_2 \\
    PB[p_i \mapsto b_1][p_i \mapsto b_2] & \implies^*_{PB} PB[p_i \mapsto b'_1][p_i \mapsto b'_2] 
\end{align*} \]

Figure 11: Evolution of Process Behaviours

- \[ PB \implies^*_{PB} PB' \]

(assuming the notation of Proposition 4.3).

The proof simply amounts to inspecting the proof of Proposition 4.3 and checking that the process behaviour \( PB' \) constructed there satisfies the new claim.

## 5 Conclusion

We started our work with an existing programming language. The first step was to “extend the type system” with additional information about some of the phenomena that take place during execution; our approach was to define the syntax of behaviours based on the notion of effect systems [8]. The second
step was to (re-)define an operational semantics in such a way that the newly added information does not influence the semantics, yet enough information is retained that it meaningfully describes the result of one step of evaluation. The third step was to prove this formally in the form of “subject reduction” and related results. In the course of this development the behaviours took on a life of their own: they were equipped with an operational semantics designed to make “subject reduction” both informative and provable, and the operational semantics was very close to the semantics of process algebras.

We believe that the main impact of this approach is not confined to the study of CML or similar languages. Rather one may ask in general for a programming language (with communication): how does the associated process algebra look. And conversely for an existing process algebra one may ask: for what kind of languages is this an appropriate process algebra. Questions like this may provide further insights into the role of operators in process algebras, e.g. the possibility of implementing an operator like ‘+’ of CCS, because the perspective is beyond that of merely translating between the syntax of a programming language and the syntax of a process algebra. Also studies of the process algebra may provide valuable information when reasoning about programs in the language. In particular “negative” information can be carried over: we may for example conclude that a program definitely deadlocks whenever its behaviour has this property.

In our future work we hope to perform a deeper study of the relationship between the semantics of the programming language and the semantics (and syntax) of the process algebra. Our work will be guided by the following slogans: (1) a process algebra is an abstract interpretation of (the effect system of) a programming language with communication; and (2) the “propositions as types” correspondence generalizes to a “processes as behaviours” correspondence. We believe that we have already demonstrated that a process algebra is an abstraction of a programming language; whether this is describable as an abstract interpretation remains to be seen.

References


CONCUR’93, SLNCS