Static and Dynamic Processor Allocation for Higher-Order Concurrent Languages*

Hanne Riis Nielson, Flemming Nielson

Computer Science Department, Aarhus University, Denmark.
e-mail:{hrnielson,fnielson}@daimi.aau.dk

Abstract. Starting from the process algebra for Concurrent ML we develop two program analyses that facilitate the intelligent placement of processes on processors. Both analyses are obtained by augmenting an inference system for counting the number of channels created, the number of input and output operations performed, and the number of processes spawned by the execution of a Concurrent ML program. One analysis provides information useful for making a static decision about processor allocation; to this end it accumulates the communication cost for all processes with the same label. The other analysis provides information useful for making a dynamic decision about processor allocation; to this end it determines the maximum communication cost among processes with the same label. We prove the soundness of the inference system and the two analyses and demonstrate how to implement them; the latter amounts to transforming the syntax-directed inference problems to instances of syntax-free equation solving problems.

1 Introduction

Higher-order concurrent languages as CML [16] and FACILE [5] offer primitives for the dynamic creation of processes and channels. A distributed implementation of these languages immediately raises the problem of processor allocation. The efficiency of the implementation will depend upon how well the *network configuration* matches the *communication topology* of the program – and here it is important which processes reside on which processors. When deciding this it will be useful to know:

- Which channels will be used by the process for input and output operations and how many times will the operations be performed?
- Which channels and processes will be created by the process and how many instances will be generated?

As an example, two processes that frequently communicate with one another should be allocated on processors in the network so as to ensure a low communication overhead.

*This is a preliminary version of a paper accepted for TAPSOFT'95; the full version appears as DAIMI-PB 483 and is also available on http://www.daimi.aau.dk/~bra8130/LOMAPS.html

In CML and FACILE processes and channels are created dynamically and this leads naturally to a distinction between two different processor allocation schemes:

- *Static processor allocation*: At compile-time it is decided where all instances of a process will reside at run-time.
- *Dynamic processor allocation*: At run-time it is decided where the individual instances of a process will reside.

The first scheme is the simpler one and it is used in the current distributed implementation of FACILE; finer grain control over parallelism may be achieved using the second scheme [18].

What has been accomplished. In this paper we present *analyses providing information for static and dynamic processor allocation* of CML programs. We shall follow the approach of [13] and develop the analyses in two stages:

- Extract the communication behaviour of the CML program.
- Analyse the behaviour.

The two analyses only differ in the second stage.

The *first stage* follows [13] in developing a type and behaviour inference system for expressing the communication capabilities of programs in CML. This formulation takes full account of the polymorphism present in ML and algorithms for the automatic extraction of behaviours from CML programs are developed in [14, 1]. As was already indicated in [12] the behaviours may be regarded as terms in a process algebra (like CCS or CSP); however the process algebra of behaviours is specifically designed so as to capture those aspects of communication that are relevant for the efficient implementation of programs in CML.

The *second stage* of the two analyses are developed in detail in the present paper. To prepare for this we first develop an analysis that uses simple ideas from abstract interpretation to count for *each behaviour* the number of channels created, the number of input and output operations performed and the number of processes spawned. To provide information for static and dynamic processor allocation we then differentiate the information with respect to *labels* associated with the `fork` operations of the CML program; these labels will identify all instances of a given process and *for each label* we count the number of channels created, the number of input and output operations performed and the number of processes spawned. The central observation is now that for the *static* allocation scheme we *accumulate* the requirements of the individual instances whereas for the *dynamic* allocation scheme we take the *maximum* of the individual instance requirements.

In this paper we prove the *correctness* of the second stage of the analysis. The analyses are specified as inference systems and the correctness proof is based on a structural operational semantics for behaviours and an appropriate abstraction of the non-negative natural numbers. The correctness of the complete analysis then follows from the subject reduction result of [13] that allows us to *lift* safety (as opposed to liveness) results from the behaviours to safety results for CML programs.

We also address the *implementation* of the second stage of the analysis. Here the idea is to transform the problem as specified by the syntax-directed inference system into a syntax-free equation solving problem where standard techniques from data flow analysis can be used to obtain fast implementations. (As already mentioned the implementation of the first stage is the topic of [14, 1].)

Comparison with other work. First we want to stress that our approach to processor allocation is that of *static program analysis* rather than, say, heuristics based on profiling as is often found in the literature on implementation of concurrent languages.

In the literature there are only few program analyses for combined functional and concurrent languages. An extension of SML with Linda communication primitives is studied in [3] and, based on the corresponding process algebra, an analysis is presented that provides useful information for the placement of processes on a finite number of processors. A functional language with communication via shared variables is studied in [9] and its communication patterns are analysed, again with the goal of producing useful information for processor (and storage) allocation. Also a couple of program analyses have been developed for concurrent languages with an imperative facet. The papers [4, 8, 15] all present reachability analyses for concurrent programs with a statically determined communication topology; only [15] shows how this restriction can be lifted to allow communication in the style of the π-calculus. Finally, [11] presents an analysis determining the number of communications on each channel connecting two processes in a CSP-like language.

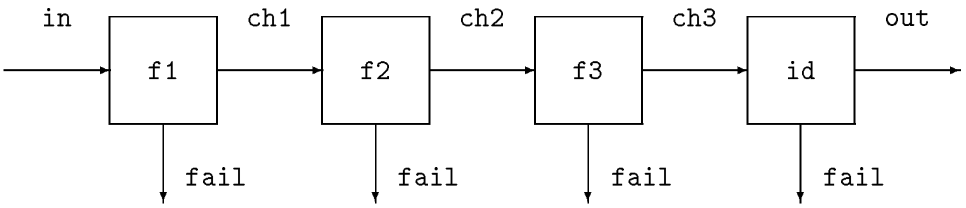
As mentioned our analysis is specified in two stages. The *first stage* is formalised in [13, 14]; similar considerations were carried out by Havelund and Larsen leading to a comparable process algebra [6] but with *no* formal study of the link to CML *nor* with any algorithm for automatically extracting behaviours. The same overall idea is present in [3] but again with *no* formal study of the link between the process algebra and the programming language.

The *second stage* of the analysis extracts much more detailed information from the behaviours and this leads to a much more complex notion of correctness than in [13]. Furthermore, the analysis is parameterised on the choice of value space thereby incorporating ideas from abstract interpretation.

2 Behaviours

Full details of the syntax of CML are not necessary for the developments of the present paper. It will suffice to introduce a running example and to use it to motivate the process algebra of CML.

Example 2.1 *Suppose we want to define a program* `pipe [f1,f2,f3] in out` *that constructs a pipeline of processes: the sequence of inputs is taken over channel* `in`*, the sequence of outputs is produced over channel* `out` *and the functions* `f1`*,* `f2`*,* `f3` *(and the identity function* `id` *defined by* `fn x => x`*) are applied in turn. To achieve concurrency we want separate processes for each of the functions* `f1`*,* `f2`*,* `f3` *(and* `id`*). This system might be depicted graphically as follows:*



Here `ch1`*,* `ch2`*, and* `ch3` *are new internal channels for interconnecting the processes; and* `fail` *is a channel over which failure of operation may be reported.*

Taking the second process as an example it may be created by the CML expression `node f2 ch1 ch2` *where the function* `node` *is given by*

```
fn f => fn in => fn out =>
fork_π (rec loop d =>
      sync (choose [wrap (receive in,
                          fn x => sync (send (out, f x));
                                  loop d),
                    send(fail,())]))
```

Here `f` *is the function to be applied,* `in` *is the input channel and* `out` *is the output channel. The function* $\texttt{fork}_\pi$ *creates a new process labelled* π *that performs as described by the recursive function* `loop` *that takes the dummy parameter* `d`*. In each recursive call the function may either report failure by* `send(fail,())` *or it may perform one step of the processing: receive the input by means of* `receive in`*, take the value* `x` *received and transmit the modified value* `f x` *by means of* `send(out,f x)` *after which the process repeats itself by means of* `loop d`*. The primitive* `choose` *allows to perform an unspecified choice between the two communication possibilities and* `wrap` *allows to modify a communication by postprocessing the value received or transmitted. The* `sync` *primitive enforces synchronisation at the right points and we refer to [16] for a discussion of the language design issues involved in this; once we have arrived at the process algebra such considerations will be of little importance to us.*

The overall construction of the network of processes is then the task of the `pipe` *function defined by*

```
rec pipe fs => fn in => fn out =>
if isnil fs
then node (fn x => x) in out
else let ch = channel ()
     in (node (hd fs) in ch; pipe (tl fs) ch out)
```

Here `fs` *is the list of functions to be applied,* `in` *is the input channel, and* `out` *is the output channel. If the list of functions is empty we connect* `in` *and* `out` *by means of a process that applies the identity function; otherwise we create a new internal channel by means of* `channel ()` *and then we create the process for the first function in the list and then recurse on the remainder of the list.*

The process algebra of CML [13] allows to give succinct representations of the communications taking place in CML programs. The terms of the process algebra are called *behaviours*, denoted $b \in \mathbf{Beh}$, and are given by

$$b \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t\ \mathrm{CHAN}_L \mid \beta \mid \mathrm{FORK}_L\ b \mid b_1; b_2 \mid b_1 + b_2 \mid \mathrm{REC}\beta.\, b$$

where $L \subseteq \mathbf{Labels}$ is a non-empty and finite set of program labels. The behaviour ϵ is associated with the pure functional computations of CML. The behaviours $L!t$ and $L?t$ are associated with sending and receiving values of type t over channels with label in L, the behaviour $t\ \mathrm{CHAN}_L$ is associated with creating a new channel with label in L and over which values of type t can be communicated, and the behaviour $\mathrm{FORK}_L\ b$ is associated with creating a new process with behaviour b and with label in L. Together these behaviours constitute the *atomic behaviours*, denoted $p \in \mathbf{ABeh}$, as may be expressed by setting

$$p \quad ::= \quad \epsilon \mid L!t \mid L?t \mid t\ \mathrm{CHAN}_L \mid \mathrm{FORK}_L\ b$$

Finally, behaviours may be composed by sequencing (as in $b_1; b_2$) and internal choice (as in $b_1 + b_2$) and we use *behaviour variables* together with an explicit REC construct to express recursive behaviours.

The structure of the types, denoted $t \in \mathbf{Typ}$, shall be of little concern to us in this paper and we shall therefore leave it mostly unspecified (but see [13]); however, we need to state that $\alpha\ \mathtt{chan}_L$ is the type of a channel with label in L over which elements of type α may be communicated. Since types might conceivably contain behaviours the notion of free variables needs to be replaced by a notion of exposed variables: we shall say that a behaviour variable β is *exposed* in a behaviour b if it has a free occurrence that is not a subterm of any type mentioned in b.

Example 2.2 *Assuming that* `fail` *is a channel of type* $\mathtt{unit\ chan}_L$ *the type inference system of [13] can be used to prove that* `pipe` *has type*

$$(\alpha \rightarrow^{\beta} \alpha)\ \mathtt{list} \rightarrow^{\epsilon} \alpha\ \mathtt{chan}_{L_1} \rightarrow^{\epsilon} \alpha\ \mathtt{chan}_{L_2} \rightarrow^{b} \mathtt{unit}$$

where b is

$$\begin{aligned}&\mathrm{REC}\beta'.(\mathrm{FORK}_\pi(\mathrm{REC}\beta''.(L_1?\alpha; \epsilon; L_2!\alpha; \beta'' + L!\mathtt{unit}))\\ &\qquad +\alpha\ \mathrm{CHAN}_{L_1}; \mathrm{FORK}_\pi(\mathrm{REC}\beta''.(L_1?\alpha; \beta; L_2!\alpha; \beta'' + L!\mathtt{unit})); \beta')\end{aligned}$$

Thus the behaviour expresses directly that the `pipe` *function is recursively defined and that it either spawns a single process or creates a channel, spawns a process and recurses. The spawned processes will all be recursive and they will either report failure over a channel in L and terminate, or else input over a channel in L_1, do something (as expressed by ϵ and β), output over a channel in L_2 and recurse.*

The *semantics* of behaviours is defined by a transition relation of the form

$$PB \Longrightarrow^{a}_{ps} PB'$$

where PB and PB' are mappings from process identifiers to *closed* behaviours and the special symbol $\surd$ denoting termination. Furthermore, a is an action that takes place and ps is a list of the processes that take part in the action. The *actions* rather closely correspond to atomic behaviours and are given by

$$a ::= \epsilon \mid L!t?L \mid t\ \text{CHAN}_L \mid \text{FORK}_L\ b$$

If the transition $PB \Longrightarrow^{a}_{ps} PB'$ has $a = \epsilon$ this means that *one* of the behaviours in PB performed some internal computation that *did not* involve communication; in other words it performed the atomic behaviour ϵ. If $a = L!t?L$ this means that *two* disctinct behaviours performed a communication: one performed the atomic behaviour $L!t$ and the other the atomic behaviour $L?t$. Finally if $a = \text{CHAN}_L$ or $a = \text{FORK}_L$ this means that *one* of the behaviours in PB allocated a new channel or forked a new process. Since we have covered all possibilities of atomic behaviours we have also covered all possibilities of actions. We refer to [13] for the precise details of the semantics as these are of little importance for the development of the analyses.

3 Value Spaces

In the analyses we want to predict the number of times certain events may happen. The precision as well as the complexity of the analyses will depend upon how we count so we shall parameterise the formulation of the analyses on our notion of counting.

This amounts to abstracting the non-negative integers $\mathbf{N}$ by a complete lattice $(\mathbf{Abs}, \sqsubseteq)$. As usual we write $\perp$ for the least element, $\top$ for the greatest element, $\bigsqcup$ and $\sqcup$ for least upper bounds by a function and $\sqcap$ for greatest lower bounds. The abstraction is expressed

$\mathcal{R} : \mathbf{N} \rightarrow_m \mathbf{Abs}$ that is strict (has $\mathcal{R}(0) = \perp$) and monotone (has $\mathcal{R}(n_1) \sqsubseteq \mathcal{R}(n_2)$ whenever $n_1 \leq n_2$); hence the ordering on the natural numbers is reflected in the abstract values. Three elements of $\mathbf{Abs}$ are of particular interest and we shall introduce special syntax for them:

$$\text{O} = \mathcal{R}(0) = \perp \qquad\qquad \text{I} = \mathcal{R}(1) \qquad\qquad \text{M} = \top$$

We cannot expect our notion of counting to be precisely reflected by $\mathbf{Abs}$; indeed it is likely that we shall allow to identify for example $\mathcal{R}(2)$ and $\mathcal{R}(3)$ and perhaps even $\mathcal{R}(1)$ and $\mathcal{R}(2)$. However, we shall ensure throughout that no identifications involve $\mathcal{R}(0)$ by demanding that $\mathcal{R}^{-1}(\text{O}) = \{0\}$ so that O really represents "did not happen".

We shall be interested in two binary operations on the non-negative integers. One is the operation of *maximum*: $\max\{n_1, n_2\}$ is the larger of n_1 and n_2. In $\mathbf{Abs}$ we shall use the binary least upper bound operation to express the maximum operation. Indeed $\mathcal{R}(\max\{n_1, n_2\}) = \mathcal{R}(n_1) \sqcup \mathcal{R}(n_2)$ holds by monotonicity of $\mathcal{R}$ as do the laws $n_1 \sqsubseteq n_1 \sqcup n_2$, $n_2 \sqsubseteq n_1 \sqcup n_2$ and $n \sqcup n = n$. As a consequence $n_1 \sqcup n_2 = \text{O}$ iff both n_1 and n_2 equal O.

The other operation is *addition*: $n_1 + n_2$ is the sum of n_1 and n_2. In **Abs** we shall have to define a function $\oplus$ and demand that (**Abs**, $\oplus$, O) is an Abelian *monoid* with $\oplus$ monotone. This ensures that we have the associative law $n_1 \oplus (n_2 \oplus n_3) = (n_1 \oplus n_2) \oplus n_3$, the absorption laws $n \oplus \text{O} = \text{O} \oplus n = n$, the commutative law $n_1 \oplus n_2 = n_2 \oplus n_1$ and by monotonicity we have also the laws $n_1 \sqsubseteq n_1 \oplus n_2$ and $n_2 \sqsubseteq n_1 \oplus n_2$. As a consequence $n_1 \oplus n_2 = \text{O}$ iff both n_1 and n_2 equal O. To ensure that $\oplus$ models addition on the integers we impose the condition $\forall n_1, n_2.\ \mathcal{R}(n_1 + n_2) \sqsubseteq \mathcal{R}(n_1) \oplus \mathcal{R}(n_2)$ that is common in abstract interpretation.

Definition 3.1 *A* value space *is a structure* (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) *as detailed above. It is an* atomic value space *if* I *is an atom (that is* $\text{O} \sqsubseteq n \sqsubseteq \text{I}$ *implies that* $\text{O} = n$ *or* $\text{I} = n$*).*

Example 3.2 *One possibility is to use* **A3** $= \{\text{O}, \text{I}, \text{M}\}$ *and define* $\sqsubseteq$ *by* $\text{O} \sqsubseteq \text{I} \sqsubseteq \text{M}$. *The abstraction function* $\mathcal{R}$ *will then map 0 to* O, *1 to* I *and all other numbers to* M. *The operations* $\sqcup$ *and* $\oplus$ *can then be given by the following tables:*

$\sqcup$	O	I	M
O	O	I	M
I	I	I	M
M	M	M	M

$\oplus$	O	I	M
O	O	I	M
I	I	M	M
M	M	M	M

This defines an atomic value space.

For two value spaces (**Abs**$'$, $\sqsubseteq'$, O', I', M', $\oplus'$, $\mathcal{R}'$) and (**Abs**$''$, $\sqsubseteq''$, O'', I'', M'', $\oplus''$, $\mathcal{R}''$) we may construct their *cartesian product* (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) by setting **Abs** $=$ **Abs**$'$ $\times$ **Abs**$''$ and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space but it is not atomic even if **Abs**$'$ and **Abs**$''$ both are. As a consequence $\text{I} = (\text{I}', \text{I}'')$ will be of no concern to us; instead we use (O', I'') and (I', O'') as appropriate.

For a value space (**Abs**$'$, $\sqsubseteq'$, O', I', M', $\oplus'$, $\mathcal{R}'$) and a non-empty set E of *events* we may construct the *indexed value space* (or function space) (**Abs**, $\sqsubseteq$, O, I, M, $\oplus$, $\mathcal{R}$) by setting **Abs** $= E \to$ **Abs**$'$ (the set of total functions from E to **Abs**$'$) and by defining $\sqsubseteq$, O, I, M, $\oplus$ and $\mathcal{R}$ componentwise. This defines a value space that is almost never atomic; as a consequence $\text{I} = \lambda e.\text{I}'$ will be of no concern to us.

For indexed value spaces we may represent ($f \in E \to$ **Abs**) by ($\text{rep}(f) \in E \hookrightarrow$ **Abs**$\backslash\{\text{O}\}$) where $E \hookrightarrow$ **Abs**$\backslash\{\text{O}\}$ denotes the set of partial functions from E to **Abs**$\backslash\{\text{O}\}$; here $\text{rep}(f)$ maps e to n iff $f(e) = n$ and $n \neq \text{O}$. In practice we want to restrict E to be a finite set in order to obtain finite representations; we write ($f \in E \to_f$ **Abs**) to indicate that f is O on all but a finite number of arguments so that such a representation is possible.

4 Counting the Behaviours

For a given behaviour b and value space **Abs** we may ask the following four questions:

$$benv \vdash \epsilon : [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{I},\mathrm{O})] \qquad benv \vdash L?t : [L \mapsto (\mathrm{O},\mathrm{I},\mathrm{O},\mathrm{O})]$$

$$benv \vdash t\ \textsc{chan}_L : [L \mapsto (\mathrm{I},\mathrm{O},\mathrm{O},\mathrm{O})] \qquad \frac{benv \vdash b : A}{benv \vdash \textsc{fork}_L\ b : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})] \oplus A}$$

$$\frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2} \qquad \frac{benv \vdash b_1 : A_1 \quad benv \vdash b_2 : A_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A}{benv \vdash \textsc{rec}\ \beta.\ b : A} \qquad benv \vdash \beta : A \quad \text{if } benv(\beta) = A$$

Table 1: Analysis of behaviours

how many times are channels labelled by L created?

how many times do channels labelled by L participate in input?

how many times do channels labelled by L participate in output?

and how many times are processes labelled by L generated?

To answer these questions we define an inference system with formulae

$$benv \vdash b : A$$

where **LabSet** $= \mathcal{P}_f(\textbf{Labels})$ is the set of finite and non-empty subsets of **Labels** and $A \in \textbf{LabSet} \to_f \textbf{Abs}$ records the required information.

In this section we shall define the inference system for answering all four questions simultaneously. Hence we let **Abs** be the four-fold cartesian product $\textbf{Ab}^4$ of an atomic value space **Ab**; we shall leave the formulation parameterised on the choice of **Ab** but a useful candidate is the three-element value space **A3** of Example 3.2 and this will be the choice in all examples.

The idea is that $A(L) = (n_c, n_i, n_o, n_f)$ means that channels labelled by L are created at most n_c times, that channels labelled by L participate in at most n_i input operations, that channels labelled by L participate in at most n_o output operations, and that processes labelled by L are generated at most n_f times. The behaviour environment *benv* then associates each behaviour variable with an element of **LabSet** $\to_f$ **Abs**.

The analysis is defined in Table 1. We use $[\,]$ as a shorthand for $\lambda L.(\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{O})$ and $[L \mapsto \vec{n}]$ as a shorthand for $\lambda L'.\left\{\begin{array}{ll} (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{O}) & \text{if } L' \neq L \\ \vec{n} & \text{if } L' = L \end{array}\right\}$. Note that I denotes the designated "one"-element in each copy of **Ab** since it is the atoms $(\mathrm{I},\mathrm{O},\mathrm{O},\mathrm{O})$, $(\mathrm{O},\mathrm{I},\mathrm{O},\mathrm{O})$, $(\mathrm{O},\mathrm{O},\mathrm{I},\mathrm{O})$, and $(\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})$ that are useful for increasing the count. In the rule for $\textsc{fork}_L$ we are deliberately incorporating the effects of the forked process; to avoid doing so simply remove the "$\oplus A$" component. The rules for sequencing, choice, and behaviour variables are straightforward given the developments of the previous section.

Note that the rule for recursion expresses a fixed point property and so allows some slackness; it would be inelegant to specify a least (or greatest) fixed point property whereas a post-fixed point[1] could easily be accomodated by incorporating a notion of subsumption into the rule. We decided not to incorporate a general subsumption rule and to aim for specifying as unique results as the rule for recursion allows.

Example 4.1 *For the* `pipe` *function of Examples 2.1 and 2.2 the analysis will give the following information (read "*M*" as "many"):*

L_1:	M *channels created and* M *inputs performed*
L_2:	M *outputs performed*
L:	M *outputs performed*
π:	M *processes created*

While this is evidently correct it also seems pretty uninformative; yet we shall see that this simple analysis suffices for developing more informative analyses for static and dynamic processor allocation.

To formally express the correctness of the analysis we need a few definitions. Given a list X of actions define:

$$\mathrm{COUNT}(X) = \lambda L.(CC(X,L), CI(X,L), CO(X,L), CF(X,L))$$

$CC(X,L)$:	the number of elements of the form t $\mathbf{CHAN}_L$ in X,
$CI(X,L)$:	the number of elements of the form $L'!t?L$ in X,
$CO(X,L)$:	the number of elements of the form $L!t?L'$ in X, and
$CF(X,L)$:	the number of elements of the form $\mathbf{FORK}_L$ b in X.

The formal version of our explanations above about the intentions with the analysis then amounts to the following soundness result:

Theorem 4.2 *If* $\emptyset \vdash b : A$ *and* $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ *then we have*

$$\mathcal{R}^*(COUNT[a_1, \cdots, a_k]) \sqsubseteq A.$$

where $\mathcal{R}^*(C)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ *if* $C(L) = (c, i, o, f)$.

5 Implementation

It is well-known that compositional specifications of program analyses (whether as abstract interpretations or annotated type systems) are not the most efficient way of obtaining the actual solutions. We therefore demonstrate how the inference problem may be transformed to an equation solving problem that is independent of the syntax of our process algebra and where standard algorithmic techniques may be applied. This approach also carries over to the inference systems for processor allocation developed subsequently.

The first step is to generate the set of equations. To show that this does not affect the set of solutions we shall be careful to avoid undesirable "cross-over" between equations generated from disjoint syntactic components of the behaviour. One possible cause for such "cross-over" is that behaviour variables

[1]We take a post-fixed point of a function f to be an argument n such that $f(n) \sqsubseteq n$.

$\mathcal{E}[\![B : \varpi : \epsilon]\!] = \{\langle\varpi\rangle = [\,]\}$
$\mathcal{E}[\![B : \varpi : L!t]\!] = \{\langle\varpi\rangle = [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{I},\mathrm{O})]\}$
$\mathcal{E}[\![B : \varpi : L?t]\!] = \{\langle\varpi\rangle = [L \mapsto (\mathrm{O},\mathrm{I},\mathrm{O},\mathrm{O})]\}$
$\mathcal{E}[\![B : \varpi : t\ \mathbf{CHAN}_L]\!] = \{\langle\varpi\rangle = [L \mapsto (\mathrm{I},\mathrm{O},\mathrm{O},\mathrm{O})]\}$
$\mathcal{E}[\![B : \varpi : \mathbf{FORK}_L\ b]\!] = \{\langle\varpi\rangle = [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})] \oplus \langle\varpi 1\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b]\!]$
$\mathcal{E}[\![B : \varpi : b_1; b_2]\!] = \{\langle\varpi\rangle = \langle\varpi 1\rangle \oplus \langle\varpi 2\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b_1]\!] \cup \mathcal{E}[\![B : \varpi 2 : b_2]\!]$
$\mathcal{E}[\![B : \varpi : b_1 + b_2]\!] = \{\langle\varpi\rangle = \langle\varpi 1\rangle \sqcup \langle\varpi 2\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b_1]\!] \cup \mathcal{E}[\![B : \varpi 2 : b_2]\!]$
$\mathcal{E}[\![B : \varpi : \beta]\!] = \{\langle\varpi\rangle = \langle\beta\rangle\}$
$\mathcal{E}[\![B : \varpi : \mathbf{REC}\ \beta.\ b]\!] = \mathrm{CLOSE}^{\varpi}_{\beta}(\ \{\langle\varpi\rangle = \langle\varpi 1\rangle,\ \langle\varpi\rangle = \langle\beta\rangle\} \cup \mathcal{E}[\![B : \varpi 1 : b]\!]\ )$

Table 2: Constructing the equation system

may be bound in more than one **REC**; one classical solution to this is to require that the overall behaviour be alpha-renamed such that this does not occur; the solution we adopt avoids this requirement by suitable modification of the equation system. Another possible cause for "cross-over" is that disjoint syntactic components of the overall behaviour may nonetheless have components that syntactically appear the same; we avoid this problem by the standard use of tree-addresses (denoted ϖ).

The function $\mathcal{E}$ for generating the equations for the overall behaviour B achieves this by the call $\mathcal{E}[\![B : \varepsilon : b]\!]$ where ε denotes the empty tree-address. In general $B : \varpi : b$ indicates that the subtree of B rooted at ϖ is of the form b and the result of $\mathcal{E}[\![B : \varpi : b]\!]$ is the set of equations produced for b. The formal definition is given in Table 2.

The key idea is that $\mathcal{E}[\![B : \varpi : b]\!]$ operates with *flow variables* of the form $\langle\varpi'\rangle$ and $\langle\beta'\rangle$. We maintain the invariant that all ϖ' occurring in $\mathcal{E}[\![B : \varpi : b]\!]$ are (possibly empty) prolongations of ϖ and that all β' occurring in $\mathcal{E}[\![B : \varpi : b]\!]$ are exposed in b. To maintain this invariant in the case of recursion we define

$$\mathrm{CLOSE}^{\varpi}_{\beta}(\mathbf{E}) = \{\ (L[\langle\varpi\rangle/\langle\beta\rangle] = R[\langle\varpi\rangle/\langle\beta\rangle])\ \mid\ (L = R) \in \mathbf{E}\}$$

(although it would actually suffice to apply the substitution $[\langle\varpi\rangle/\langle\beta\rangle]$ on the righthand sides of equations and it would be correct to remove the trivial equation produced).

Terms of the equations are formal terms over the flow variables (that range over the complete lattice $\mathbf{LabSet} \to \mathbf{Abs}$), the operations $\oplus$ and $\sqcup$ and the constants (that are elements of the complete lattice $\mathbf{LabSet} \to \mathbf{Abs}$). Thus all terms are monotonic in their free flow variables. A *solution* to a set $\mathbf{E}$ of equations is a partial function σ from flow variables to $\mathbf{LabSet} \to \mathbf{Abs}$ such that all flow variables in $\mathbf{E}$ are in the domain of σ and such that all equations $(L = R)$ of $\mathbf{E}$ have $\sigma(L) = \sigma(R)$ where σ is extended to formal terms in the obvious way. We write $\sigma \models \mathbf{E}$ whenever this is the case.

Theorem 5.1 $[\,] \vdash b : A$ *iff* $\exists\sigma.\ \sigma \models \mathcal{E}[\![b : \varepsilon : b]\!] \wedge \sigma(\langle\varepsilon\rangle) = A.$

Corollary 5.2 *The least (or greatest) A such that* $[\,] \vdash b : A$ *is* $\sigma(\langle\varepsilon\rangle)$ *for the least (or greatest)* σ *such that* $\sigma \models \mathcal{E}[\![b : \varepsilon : b]\!]$.

We have now transformed our inference problem to a form where the standard algorithmic techniques can be exploited. These include simplifications of the equation system, partitioning the equation system into strongly connected components processed in (reverse) topological order, widening to ensure convergence when **Abs** does not have finite height etc.; a good overview of useful techniques may be found in [2, 7, 10, 17]. Also the flow variables may be decomposed to families of flow variables over simpler value spaces.

6 Static Processor Allocation

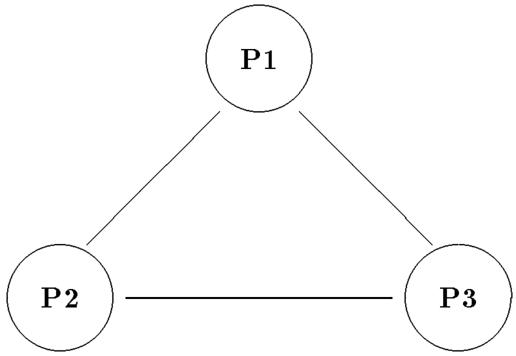
The idea behind the static processor allocation is that all processes with the *same* label will be placed on the *same* processor and we would therefore like to know what requirements this puts on the processor. To obtain such information we shall extend the simple counting analysis of Section 4 to associate information with the *process labels* mentioned in a given behaviour b. For each process label L_a we therefore ask the four questions of Section 4 accumulating the *total* information for all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 6.1 *Let us return to the* `pipe` *function of Examples 2.1 and 2.2 and suppose that we want to perform* static processor allocation. *This means that* all *instances of the processes labelled* π *will reside on the* same *processor. The analysis should therefore estimate the* total *requirements of these processes as follows:*

main:	L_1:	M *channels created*	π:	L_1:	M *inputs performed*
	π:	M *processes created*		L_2:	M *outputs performed*
				L:	M *outputs performed*

Note that even though each process labelled by π *can only communicate once over* L *we can generate many such processes and their combined behaviour is to communicate many times over* L. *It follows from this analysis that the main program does not in itself communicate over* L_2 *or* L *and that the processes do not by themselves spawn new processes.*

Now suppose we have a network of processors that may be explained graphically as follows:



One way to place our processes is to place the main program on **P1** *and all the processes labelled* π *on* **P2**. *This requires support for multitasking on* **P2** *and for multiplexing (over* L_1*) on* **P1** *and* **P2**.

The analysis (specified in Table 3) is obtained by modifying the inference system of Section 4 to have formulae

$$benv \vdash b : A \;\&\; P$$

where $A \in \mathbf{LabSet} \to_f \mathbf{Abs}$ as before and the new ingredient is

$$P : \mathbf{LabSet} \to_f (\mathbf{LabSet} \to_f \mathbf{Abs})$$

The idea is that if some process is labelled L_a then $P(L_a)$ describes the *total* requirements of all processes labelled by L_a. The behaviour environment *benv* is an extension of that of Section 4 in that it associates pairs A & P with the behaviour variables. Note that in the rule for $\texttt{FORK}_L$ we have removed the "$\oplus A$" component from the local effect; instead it is incorporated in the global effect for L.

To express the correctness of the analysis we need to keep track of the relationship between the process identifiers and the associated labels. So let *penv* be a mapping from process identifiers to elements L_a of **LabSet**. We shall say that *penv* respects the derivation sequence $PB \Longrightarrow^{a_1}_{ps_1} \ldots \Longrightarrow^{a_k}_{ps_k} PB'$ if whenever (a_i, ps_i) have the form $(\texttt{FORK}_L\ b, (pi_1, pi_2))$ then $penv(pi_2) = L$; this ensures that the newly created process (pi_2) indeed has a label (in L) as reported by the semantics.

We can now redefine the function COUNT of Section 4. Given a list $\mathcal{X}$ of pairs of actions and lists of process identifiers define

$$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.(CC_{L_a}(\mathcal{X}, L), CI_{L_a}(\mathcal{X}, L), CO_{L_a}(\mathcal{X}, L), CF_{L_a}(\mathcal{X}, L))$$

$CC_{L_a}(\mathcal{X}, L)$:	the number of elements of the form $(t\ \texttt{CHAN}_L, pi)$ in $\mathcal{X}$ where $penv(pi) = L_a$,
$CI_{L_a}(\mathcal{X}, L)$:	the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$, where $penv(pi) = L_a$,
$CO_{L_a}(\mathcal{X}, L)$:	the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, where $penv(pi) = L_a$, and
$CF_{L_a}(\mathcal{X}, L)$:	the number of elements of the form $(\texttt{FORK}_L b, (pi, pi'))$ in $\mathcal{X}$ where $penv(pi) = L_a$.

Soundness of the analysis then amounts to:

$$benv \vdash \epsilon : [\,] \ \& \ [\,]$$

$$benv \vdash L!t : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{I},\mathrm{O})] \ \& \ [\,]$$

$$benv \vdash L?t : [L \mapsto (\mathrm{O},\mathrm{I},\mathrm{O},\mathrm{O})] \ \& \ [\,]$$

$$benv \vdash t \ \mathbf{CHAN}_L : [L \mapsto (\mathrm{I},\mathrm{O},\mathrm{O},\mathrm{O})] \ \& \ [\,]$$

$$\frac{benv \vdash b : A \ \& \ P}{benv \vdash \mathbf{FORK}_L \ b : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})] \ \& \ ([L \mapsto A] \oplus P)}$$

$$\frac{benv \vdash b_1 : A_1 \ \& \ P_1 \quad benv \vdash b_2 : A_2 \ \& \ P_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2 \ \& \ P_1 \oplus P_2}$$

$$\frac{benv \vdash b_1 : A_1 \ \& \ P_1 \quad benv \vdash b_2 : A_2 \ \& \ P_2}{benv \vdash b_1 + b_2 : A_1 \sqcup A_2 \ \& \ P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A \ \& \ P] \vdash b : A \ \& \ P}{benv \vdash \mathbf{REC} \ \beta.\ b : A \ \& \ P}$$

$$benv \vdash \beta : A \ \& \ P \qquad \text{if } benv(\beta) = A \ \& \ P$$

Table 3: Analysis for static process allocation

Theorem 6.2 *Assume that* $\emptyset \vdash b : A \ \& \ P$ *and* $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ *and let penv be a mapping from process identifiers to elements of* **LabSet** *respecting the above derivation sequence and such that* $penv(pi_0) = L_0$*. We then have*

$$\mathcal{R}^*(COUNT^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \oplus [L_0 \mapsto A])$$

where $\mathcal{R}^*(C)(L_a)(L) = (\mathcal{R}(c), \mathcal{R}(i), \mathcal{R}(o), \mathcal{R}(f))$ *if* $C(L_a)(L) = (c, i, o, f)$.

Note that the lefthand side of the inequality counts the number of operations for all processes whose labels is given (by L_a); hence our information is useful for static processor allocation.

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system. This is hardly any different from the approach of Section 5 except that by now there is even greater scope for decomposing the flow variables into families of flow variables over simpler value spaces.

7 Dynamic Processor Allocation

The idea behind the dynamic processor allocation is that the decision of how to place processes on processors is taken dynamically. Again we will be interested in knowing which requirements this puts on the processor but in contrast to the previous section we are only concerned with a single process rather than all processes with a given label. We shall now modify the analysis of Section 6 to associate *worst-case* information with the *process labels* rather than accumulating the total information. For each process label L_a we therefore ask the four

$$\frac{benv \vdash b : A \;\&\; P}{benv \vdash \texttt{FORK}_L\ b : [L \mapsto (\mathrm{O},\mathrm{O},\mathrm{O},\mathrm{I})] \;\&\; ([L \mapsto A] \sqcup P)}$$

$$\frac{benv \vdash b_1 : A_1 \;\&\; P_1 \quad benv \vdash b_2 : A_2 \;\&\; P_2}{benv \vdash b_1; b_2 : A_1 \oplus A_2 \;\&\; P_1 \sqcup P_2}$$

$$\frac{benv[\beta \mapsto A] \vdash b : A \;\&\; P}{benv \vdash \texttt{REC}\ \beta.\ b : A \;\&\; P}$$

$$benv \vdash \beta : A \;\&\; [\,] \qquad \text{if } benv(\beta) = A$$

Table 4: Analysis for dynamic process allocation

questions of Section 4 taking the *maximum* information over all processes with label L_a: how many times are channels labelled by L created, how many times do channels labelled by L participate in input, how many times do channels labelled by L participate in output, and how many times are processes labelled by L generated?

Example 7.1 *Let us return to the* `pipe` *function of Examples 2.1 and 2.2 and suppose that we want to perform* dynamic processor allocation. *This means that all the processes labelled π* need not *reside on the same processor. The analysis should therefore estimate the* maximal *requirements of the instances of these processes as follows:*

main:	L_1:	M *channels created*	π:	L_1:	M *inputs performed*
	π:	M *processes created*		L_2:	M *outputs performed*
				L:	I *output performed*

Note that now we do record that each individual process labelled by π actually only communicates over L at most once.

Returning to the processor network of Example 6.1 we may allocate the main program on **P1** *and the remaining processes on* **P2** *and* **P3** *(and possibly* **P1** *as well): say* `f1` *and* `f3` *on* **P2** *and* `f2` *and* `id` *on* **P3**. *Facilities for multitasking are needed on* **P2** *and* **P3** *and facilities for multiplexing on all of* **P1**, **P2** *and* **P3**.

The inference system still has formulae

$$benv \vdash b : A \;\&\; P$$

where A and P are as in Section 6 and now *benv* is as in Section 4: it does not incorporate the P component[2]. Most of the axioms and rules are as in Table 3; the modifications are listed in Table 4.

A difference from Section 6 is that now we need to keep track of the individual process identifiers. We therefore redefine the function COUNT^{penv} as follows:

[2]It could be as in Section 6 as well because we now combine P components using $\sqcup$ rather than $\oplus$.

$\mathrm{COUNT}^{penv}(\mathcal{X}) = \lambda L_a.\lambda L.((CC_{PI}(\mathcal{X},L), CI_{PI}(\mathcal{X},L), CO_{PI}(\mathcal{X},L), CF_{PI}(\mathcal{X},L))$
where $PI = penv^{-1}(L_a))$

$CC_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(t\ \mathbf{CHAN}_L, pi)$ in $\mathcal{X}$,
$CI_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(L'!t?L, (pi', pi))$ in $\mathcal{X}$,
$CO_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(L!t?L', (pi, pi'))$ in $\mathcal{X}$, and
$CF_{PI}(\mathcal{X},L)$:	the maximum over all $pi \in PI$ of the number of elements of the form $(\mathbf{FORK}_L b, (pi, pi'))$ in $\mathcal{X}$.

Soundness of the analysis then amounts to:

Theorem 7.2 *Assume that $\emptyset \vdash b : A \ \&\ P$ and $[pi_0 \mapsto b] \Longrightarrow^{a_1}_{ps_1} \cdots \Longrightarrow^{a_k}_{ps_k} PB$ and let penv be a mapping from process identifiers to elements of* **LabSet** *respecting the above derivation sequence and such that $penv(pi_0) = L_0$. We then have*

$$\mathcal{R}^*(COUNT^{penv}[(a_1, ps_1), \cdots, (a_k, ps_k)]) \sqsubseteq (P \sqcup [L_0 \mapsto A])$$

where $\mathcal{R}^*$ is as in Theorem 6.2.

Note that the lefthand side of the inequality gives the maximum number of operations over all processes with a given label; hence our information is useful for dynamic processor allocation.

To obtain an efficient implementation of the analysis it is once more profitable to generate an equation system and the remarks at the end of the previous section still apply.

8 Conclusion

The specifications of the analyses for static and dynamic allocation have much in common; the major difference of course being that for static processor allocation we *accumulate* the total numbers whereas for dynamic processor allocation we calculate the *maximum*; a minor difference being that for the static analysis it was crucial to let behaviour environments include the P component whereas for the dynamic analysis this was hardly of any importance.

This difference in approach is reminiscent of the difference between the formulation of MFP-style and MOP-style analyses: in the former the effects of paths (corresponding to process identifiers with the same label set) are merged along the way whereas in the latter the paths (corresponding to the process identifiers) have to be kept separate and their effects can only be merged when the propagation of effects has taken place.

Acknowledgements. We would like to thank Torben Amtoft for many interesting discussions. This research has been funded in part by the LOMAPS (ESPRIT BRA) and DART (Danish Science Research Council) projects.

References

[1] T.Amtoft, F.Nielson, H.R.Nielson: Type and behaviour reconstruction for higher-order concurrent programs. This proceedings.

[2] J.Cai, R.Paige: Program Derivation by Fixed Point Computation. *Science of Computer Programming* **11**, pp. 197–261, 1989.

[3] R. Cridlig, E.Goubault: Semantics and analysis of Linda-based languages. *Proc. Static Analysis*, Springer Lecture Notes in Computer Science **724**, 1993.

[4] C.E.McDowell: A practical algorithm for static analysis of parallel programs. *Journal of parallel and distributed computing* **6**, 1989.

[5] A.Giacalone, P.Mishra, S.Prasad: Operational and Algebraic Semantics for Facile: a Symmetric Integration of Concurrent and Functional Programming. *Proc. ICALP'90*, Springer Lecture Notes in Computer Science **443**, 1990.

[6] K.Havelund, K.G.Larsen: The Fork Calculus. *Proc. ICALP'93*, Springer Lecture Notes in Computer Science **700**, 1993.

[7] M.S.Hecht: *Flow Analysis of Computer Programs*, North-Holland, 1977.

[8] Y.-C.Hung, G.-H.Chen: Reverse reachability analysis: a new technique for deadlock detection on communicating finite state machines. *Software — Practice and Experience* **23**, 1993.

[9] S.Jagannathan, S.Week: Analysing stores and references in a parallel symbolic language. *Proc. L&FP*, 1994.

[10] M.Jourdan, D.Parigot: Techniques for Improving Grammar Flow Analysis. *Proc. ESOP'90*, Springer Lecture Notes in Computer Science **432**, pp. 240–255, 1990.

[11] N. Mercouroff: An algorithm for analysing communicating processes. *Proc. of MFPS*, Springer Lecture Notes in Computer Science **598**, 1992.

[12] F.Nielson, H.R.Nielson: From CML to Process Algebras. *Proc. CONCUR'93*, Springer Lecture Notes in Computer Science **715**, 1993.

[13] H.R.Nielson, F.Nielson: Higher-Order Concurrent Programs with Finite Communication Topology. *Proc. POPL'94*, pp. 84–97, ACM Press, 1994.

[14] F.Nielson, H.R.Nielson: Constraints for Polymorphic Behaviours for Concurrent ML. *Proc. CCL'94*, Springer Lecture Notes in Computer Science **845**, 1994.

[15] J.H.Reif, S.A.Smolka: Dataflow analysis of distributed communicating processes. *International Journal of Parallel Programs* **19**, 1990.

[16] J.R.Reppy: Concurrent ML: Design, Application and Semantics. Springer Lecture Notes in Computer Science **693**, pp. 165–198, 1993.

[17] R.Tarjan: Iterative Algorithms for Global Flow Analysis. In J.Traub (ed.), *Algorithms and Complexity*, pp. 91–102, Academic Press, 1976.

[18] B.Thomsen. Personal communication, May 1994.