Partial-Evaluation Techniques for Concurrent Programs

Mihnea Marinescu Benjamin Goldberg
Department of Computer Science, New York University
e-mail: {marinesc,goldberg}@cs.nyu.edu

Abstract

This paper presents an application of *partial evaluation* (program specialization) techniques to *concurrent programs*. The language chosen for this investigation is a very simple CSP-like language. A standard binding-time analysis for imperative languages is extended in order to deal with the basic concurrent constructs (synchronous communication and nondeterministic choice). Based on the binding-time annotations, a specialization transformation is defined and proved correct. In order to maintain a simple and clear presentation, the specialization algorithm addresses only the data transfer component of the communication; partial evaluation, the way it is defined here, always generates residual synchronizations. However, a simple approximate analysis for detecting and removing redundant synchronizations from the residual program (i.e. synchronizations whose removal does not increase the nondeterminism of a program) can be performed. The paper also addresses pragmatic concerns such as improving the binding-time analysis, controlling loop unrolling and the consequences of lifting nondeterminism from run-time to specialization-time. Finally, the power of the newly developed technique is shown in several examples.

Keywords: Partial evaluation, binding-time analysis, concurrency, CSP, nondeterminism.

1 Introduction

1.1 Motivation

Semantic methods have proven to be an effective tool in optimizing deterministic sequential languages. One of these methods, *partial evaluation* (PE) is a source-to-source program transformation, often presented as follows:

> Given a program and part of its input data, generate a residual program that, when running on the rest of the input data, behaves as the original program running on the whole input.

PEPM '97 Amsterdam, ND

The idea behind standard PE is to reduce the number of computation steps performed at run-time by doing as many computation steps as possible at the specialization time (i.e. *before* run-time).

Our aim is to reduce the number of *communication* and computation steps performed at run-time by doing as many communication and computation steps as possible at the specialization time (without altering the meaning of the program).

The motivation for doing this is the performance improvement that could be obtained when specializing a wide range of concurrent systems ranging from operating systems to numerical computing modules and artificial intelligence programs. Research centered around using PE for efficiency gains has been conducted for all the domains mentioned above. Applications of PE for operating systems are described in papers by Pu, Consel, et al. [CPW93, PC+95]; they suggest that *incremental* specialization techniques need to be developed and used. A paper discussing an application of PE for numerical computing (deterministic algorithms) is [Ber90]. However, none of these studies examines PE for a *concurrent* system.

Finally, we believe that there is a significant class of concurrent algorithms that have a *static communication topology*, where the number of communicating channels and the association of the channels to threads, are fixed (or has a narrow bounded variation). As Pepper showed in [Pep93], several concurrent programs can be obtained by program transformations directly from specifications of an algorithm. The communication topology of these programs reflects the data dependencies corresponding to the data structures used in the algorithm specification; for most algorithms these dependencies are *static* or can be *statically bounded*.

1.2 Background

While we are not aware of studies concerning the PE of *concurrent* programs, there is a long history of research conducted (separately) in:

- the PE of deterministic sequential languages (both functional and imperative) and
- the understanding and formalization of the concepts of concurrency and communication.

Papers, such as [FOF88], on partial evaluation for concurrent logic programs, are not significant for this research.

Our paper uses many of the standard PE results. Consel and Danvy's tutorial [CD93] concisely presents the central ideas of the domain. The book by Jones et. al. [JGS93] presents the typical PE framework and addresses many pragmatic concerns for a wide variety of language constructs. The presentation of a taxonomy of program transformers given by Glück and Sørensen in [GS96a] is also relevant for concurrent programs.

Not considering PE of Prolog, Turchin's paper [Tur93] is probably the only reference to the study of PE of languages with nondeterministic constructs. It shows a transformation that has as an intermediate result a program that is apparently nondeterministic; however, the nondeterminism can be further specialized and the final result is deterministic. The lesson to be learned is that PE techniques can also be used to transform a (falsely) nondeterministic construct into a deterministic one. One step further is the one that we propose: to use specialization (whenever possible) to lift the (actual) nondeterminism from run-time to specialization-time.

Since the concurrent language studied has imperative features, we refer to results presented in papers by Meyer [Mey91], Nirkhe and Pugh [NP92] and Jones et. al. [JGS93].

For concurrency the focus is on the very basic concepts of communication, synchronization and nondeterminism. Excellent references include Hoare's seminal paper on Communicating Sequential Processes (CSP) [Hoa78], his book [Hoa85], which offers intuition and formalizes semantics for CSP-like constructs, and Milner's book [Mil89] on the Calculus of Communicating Systems (CCS). For the formal specification of the operational semantics of a CSP-like language Plotkin's paper [Plo83] is a standard reference.

Among the papers that describe static analysis algorithms for concurrent programs, Reif and Smolka's [RS90] dataflow analysis is particularly relevant. Even more relevant for our research is the work of Mercouroff [Mer91] that presents an abstract interpretation algorithm for statically computing send-receive matches for a CSP-like program.

An Unfold/Fold transformation strategy for CCS programs is studied in a recent paper by de Francesco and Santone [dFS96]. Their work is somewhat complementary to ours, since they consider a transformational framework for CCS, a concurrent language with *no values*, while we study an imperative CSP-like language and we concentrate on the binding-time analysis and on lifting computation, communication and nondeterminism safely from run-time to specialization-time.

A paper describing the analysis and removal of redundant synchronizations was presented by Gupta and Schonberg [GS96b], but their analysis is done in a different setting, for data parallel programs and is not related to PE.

1.3 Outline

The purpose of this paper is to show how the PE framework can be extended in order to include all basic concurrent language constructs. We show how to specialize the data transfer (message passing) component of the communication and how to lift the nondeterministic choice from run-time to specialization time. However, we stop short of specializing synchronization and instead we show a post-specialization analysis for synchronization removal.

The next section describes the syntax of the language and its operational semantics using a labeled transition system (LTS) specification. Then a standard PE framework is presented. It includes the specification of the binding-time analysis which is extended to cope with the basic concurrent language constructs and of the specialization rules using a LTS with actions. The central result, the correctness of the algorithm, is stated and a proof sketch is given. This result generalizes the correctness of the PE of a deterministic sequential program and is closely related to (strong) bisimulation, the equivalence relation on transition systems[1]. More practical issues, concerning improvements of the quality of the partial evaluator as well as its termination properties are then briefly discussed. A special section is dedicated to synchronization analysis and removal.

2 The Language

The language is very similar to the CSP-language kernel (see [Hoa85]) and hence to the programming language Occam (see [May83, Ltd84]). Its simple imperative skeleton and well-understood concurrent language primitives allow for a clear presentation of a partial evaluator.

2.1 Syntax and Comments

The specification of the syntax of the language is given in Fig. 1.

Expressions have the usual syntax and are side-effect free (this is important because the guards may include boolean expressions). For simplicity there is no aliasing and no user-defined functions in the language.

A specific aspect of the language presented here is the introduction of the domain of *threads*. We have chosen this word rather then the word *process* because Milner's notion of a process is different; the threads here are more like the processes in [Hoa85] and [Plo83]. A thread denotes a *sequential* unit of execution and there is a unique thread identifier associated with each thread body. Because of this one-to-one correspondence between threads and thread identifiers we will often abuse notation and identify these two domains. For all programs, a special thread with the identifier $MAIN$ exists. The body of any thread consists of (local) declarations of variables and commands.

The parallel composition command enforces the identification of the threads that are to be executed in parallel.

Recursive threads are not allowed, the reason being to guarantee the existence of a finite number of threads[2].

Communication is performed via synchronized unidirectional message passing on channels. Channel declarations, which associate a channel with a unique pair

[1] Introduced in [Par81]. See also Milner's work [Mil89].

[2] Hoare for instance forbids process generation inside a recursion.

Programs:	P	$\in$	$PROG$
Threads:	t	$\in$	$THREAD$
Thread Identifiers:	tid	$\in$	TID
Declarations of Threads:	$tdecl$	$\in$	$TDECL$
Declarations of Channels:	$chdecl$	$\in$	$CHDECL$
Declarations of Variables:	d	$\in$	$DECL$
Channel names:	α	$\in$	$CHAN$
Expressions:	a, b, X	$\in$	$EXPR$
Locations:	X	$\in$	LOC
Boolean expressions:	b	$\in$	$BEXP$
Program Points:	i	$\in$	PP
Commands:	c	$\in$	COM
Guarded Commands:	gc	$\in$	$GCOM$

$$
\begin{array}{lcl}
P & ::= & chdecl;\ tdecl \\
chdecl & ::= & \mathbf{channel}\ \alpha : (tid_0, tid_1) \\
 & | & chdecl_0; chdecl_1 \mid \varepsilon \\
tdecl & ::= & \mathbf{thread}\ tid\ \mathbf{is}\ d;\ c\ \mathbf{end} \mid tdecl; tdecl \mid \varepsilon \\
d & ::= & \mathbf{var}\ x \mid d_0; d_1 \mid \varepsilon \\
c & ::= & \mathbf{skip} \mid i : X := a \mid i : \alpha?X \mid i : \alpha!a \\
 & | & c_0; c_1 \mid tid_0 \| tid_1 \\
 & | & i : \mathbf{alt}\ gc\ \mathbf{end\ alt} \mid i : \mathbf{do}\ gc\ \mathbf{end\ do} \\
gc & ::= & b \to c \mid b \wedge i_0 : \alpha?X \to i_1 : c \\
 & | & b \wedge i_0 : \alpha!a \to i_1 : c \mid gc_0 [\!] gc_1
\end{array}
$$

Figure 1: Syntax of a CSP-language

of threads, are used to enforce a *static topology of the communication.*

There is *no multicommunication.* This doesn't reduce the power of the language; one can define multiplexor/demultiplexor threads to simulate multicommunication.

As in the standard CSP, we *disallow the use of global variables.* Allowing read-only global variables will not change the results in this paper since they are *not* a source of nondeterminism. Again there is no loss of power: global variables can be implemented by defining, for each of them, a thread whose body is an iterated nondeterministic choice.

The idea is to *isolate only one syntactic construct* that has *nondeterministic semantics*: the *choice.* By imposing the two restrictions above, no multicommunication and no global variables, we have eliminated the other two sources of nondeterminism.

CSP has an explicit channel hiding operation. We have chosen to omit it from the source language. We will address this operation when we introduce *observability*, a semantic notion which will be discussed in the next section.

Finally a minor detail that becomes important for PE: we explicitly associate program points with (most of) the commands in a program, as specified in Fig. 1.

2.2 Semantics

This section presents an operational semantics of the language. We have chosen the operational view because, as Mosses writes in [Mos90] "...it is debatable whether the denotational treatment of concurrency is satisfactory ...in contrast Structural Operational Semantics extends easily from sequential languages to concurrency".

Nevertheless, since the semantics of the declarations and expressions is functional (because expressions are side-effect free), the part of the semantics associated with them has a denotational flavor.

In order to keep the presentation clear we distinguish three types of environments:

- for channels (denoted by $\omega \in CHENV$)
- for threads (denoted by $\tau \in TENV$)
- for variables (denoted by $\sigma \in ENV$)

The semantics of the commands is described in terms of a Labeled Transition System (LTS) similar to the one in [Win93] or in [Plo83].

For the ease of presentation, the specification of the LTS omits the channel environment and the thread environment. Notice that once the declarations have been processed these environments do not change.

The LTS specified in Fig. 2 appears quite standard. However, it differs by considering the transition relation as a *relation between sets of configurations* rather then between configurations. This is achieved by distributing the environment to each thread rather than keeping one environment and considering commands executed in parallel.

We write a transition in the form:

$$\Delta \xrightarrow{\lambda} \Delta'$$

where Δ and Δ' are (the initial and the final) sets of configurations and λ is the communication label associated with this transition.

Each Δ is either **fail** or has the form:

$$< c_0, \sigma_0 > \| < c_1, \sigma_1 > \| \ldots$$

where c_0, c_1 are commands and σ_0, σ_1 are the local environments.

There are several non-standard transitions:

- (SPAWN): notice the *asymmetry* of the parallel composition: thread tid_0 inherits the environment σ and the communication capabilities of the parent thread (we have chosen this solution for simplicity); the *associativity* of the parallel composition still holds;
- (JOIN): the environment of the second thread (i.e. tid_1) is discarded.

Based on the specified LTS, we can define the meaning of a thread.

For *deterministic sequential programs*, the operational meaning is usually defined as a function from an initial state to a final state.

For a *concurrent system*, meaning is in general a linear or branching structure of events[3] (where these events are associated with the communication).

We define a *hybrid meaning* for the threads: a pair *(trace, final_state)* where:

- *trace* is an ordered, possibly infinite, sequence of events; these events will be of the form: $\alpha!x$ with $\alpha \in CHAN$ such that $\mathcal{O}(\alpha)$ (i.e. the communication on that channel is *observable*) and $x \in VAL$.

[3] In his book Hoare [Hoa85] gives semantics in terms of traces, refusals and failures of events.

VAR the domain of variables
VAL the domain of values
$ENV = VAR \to VAL$ the environment of variables
$TENV = TID \to THREAD$ the environment of threads
$LABELS = \{\alpha!x \mid \alpha \in CHAN \text{ and } x \in VAL\} \cup \{\alpha?x \mid \alpha \in CHAN \text{ and } x \in VAL\}$
$CONF = COM \times ENV$ the configurations (as usual, a configuration with an empty command is represented only by the environment)
$\mathcal{E}_d : DECL \to ENV$ the denotational specification of the declarations
$\mathcal{E}_e : EXPR \to ENV \to VAL$ the denotational specification of the expressions
$\to_t = THREAD \to CONF$ start a thread
$\to \subset CONF^N \times LABELS \times (CONF^M \cup \{\mathbf{fail}\})$

$$\frac{\mathcal{E}_d[\![d]\!] = \sigma}{< \textbf{thread } tid \textbf{ is } d;\ c \textbf{ end} > \to_t < c, \sigma >} \quad (START)$$

$$< \textbf{skip}, \sigma > \to \sigma \quad (SKIP)$$

$$\frac{\mathcal{E}_e[\![a]\!]\sigma = x}{< X := a, \sigma > \to \sigma[x/X]} \quad (ASSIGN)$$

$$< \alpha?X, \sigma > \xrightarrow{\alpha?x} \sigma[x/X] \quad (RECEIVE)$$

$$\frac{\mathcal{E}_e[\![a]\!]\sigma = x}{< \alpha!a, \sigma > \xrightarrow{\alpha!x} \sigma} \quad (SEND)$$

$$\frac{< c_0, \sigma > \xrightarrow{\lambda} < c_0', \sigma' >}{< c_0; c_1, \sigma > \xrightarrow{\lambda} < c_0'; c_1, \sigma' >} \quad (SEQ)$$

$$\frac{< gc, \sigma > \xrightarrow{\lambda} < c, \sigma' >}{< \textbf{alt } gc \textbf{ end alt}, \sigma > \xrightarrow{\lambda} < c, \sigma' >} \quad (ALT)$$

$$\frac{< gc, \sigma > \xrightarrow{\lambda} < c, \sigma' >}{< \textbf{do } gc \textbf{ end do}, \sigma > \xrightarrow{\lambda} < c; \textbf{do } gc \textbf{ end do}, \sigma' >} \quad (UNROLL)$$

$$\frac{< gc, \sigma > \xrightarrow{\lambda} \textbf{fail}}{< \textbf{do } gc \textbf{ end do}, \sigma > \xrightarrow{\lambda} \sigma} \quad (EXIT\ LOOP)$$

$$\frac{t_0 = \tau(tid_0) \quad t_0 \to_t < c_0, \sigma_0 > \quad t_1 = \tau(tid_1) \quad t_1 \to_t < c_1, \sigma_1 >}{< tid_0 \parallel tid_1, \sigma > \to < (c_0, \sigma \oplus \sigma_0) \parallel (c_1, \sigma_1) >} \quad (SPAWN)$$

where $\oplus : ENV \times ENV \to ENV$ *denotes environment composition*

$$\frac{< c_0, \sigma_0 > \xrightarrow{\lambda} < c_0', \sigma_0' >}{< (c_0, \sigma_0) \parallel (c_1, \sigma_1) > \xrightarrow{\lambda} < (c_0, \sigma_0') \parallel (c_1, \sigma_1) >} \quad (INTERLEAVE) \quad \textit{and its dual}$$

$$\frac{< c_0, \sigma_0 > \xrightarrow{\alpha?x} < c_0', \sigma_0' > \quad < c_1, \sigma_1 > \xrightarrow{\alpha!x} < c_1', \sigma_1 >}{< (c_0, \sigma_0) \parallel (c_1, \sigma_1) > \to < (c_0', \sigma_0') \parallel (c_1', \sigma_1) >} \quad (COMM) \quad \textit{and its dual}$$

$$< \sigma_0 \parallel \sigma_1 > \to \sigma_0 \quad (JOIN)$$

$$\frac{\mathcal{E}_e[\![b]\!]\sigma = \textbf{true}}{< b \to c, \sigma > \to < c, \sigma >}$$

$$\frac{\mathcal{E}_e[\![b]\!]\sigma = \textbf{true}}{< b \wedge \alpha?X \to c, \sigma > \xrightarrow{\alpha?x} < c, \sigma[x/X] >}$$

$$\frac{\mathcal{E}_e[\![b]\!]\sigma = \textbf{true} \quad \mathcal{E}_e[\![a]\!]\sigma = x}{< b \wedge \alpha!a \to c, \sigma > \xrightarrow{\alpha!x} < c, \sigma >}$$

$$\frac{\mathcal{E}_e[\![b]\!]\sigma = \textbf{false}}{< b \ /\ b \wedge \alpha?X \ /\ b \wedge \alpha!a \ \to c, \sigma > \to \textbf{fail}} \quad \textit{three rules}$$

$$\frac{< gc_0, \sigma > \xrightarrow{\lambda} < c, \sigma' >}{< gc_0 [\!] gc_1, \sigma > \xrightarrow{\lambda} < c, \sigma' >} \quad \textit{and its dual}$$

$$\frac{< gc_0, \sigma > \to \textbf{fail} \quad < gc_1, \sigma > \to \textbf{fail}}{< gc_0 [\!] gc_1, \sigma > \to \textbf{fail}}$$

Figure 2: Semantics of the CSP-like language

- *final_state* is introduced to distinguish programs having the same observable behavior but different termination properties; the *final_state* is set to:
  - ∞ if no final configuration exists;
  - δ if no transition is possible (the system deadlocks because of circular waiting or because it fails in a nondeterministic choice);
  - the environment σ from the final configuration.

Following the spirit of the CCS (see [Mil89]) we argue that a specification of a concurrent system should include, apart from the text of the program that describes the system, a partition of the communication channels into *observable* and *non-observable*. We use the predicate:

$$\mathcal{O} : CHAN \rightarrow \{\mathbf{true}, \mathbf{false}\}$$

to denote this partition. The intuition is that, since a concurrent system is an open system, its semantics reflects the interaction with the environment. Thus, the meaning of a concurrent system should not include internal (non-observable) communication.

$m.trace$ and $m.\sigma$ are used to refer to the two components of the meaning m.

Notice that $m.trace$ is updated on the transitions associated with rules (COMM).

The meaning of a program P denoted by $\mathcal{M}(P)$ is set to be the meaning of the *MAIN* thread running in an environment where all channels and threads are defined i.e. all top-level declarations have been processed.

3 Partial Evaluation

3.1 Preliminary Comments

Our idea is to specialize a concurrent program by concurrently specializing its component threads. During this process of specialization we perform some data transfer and commit to particular choices.

The idea behind CSP is to *combine data transfer with synchronization*. Our idea is to **decouple** them for PE. We have developed both an *on-line* approach (in which the synchronization analysis is part of an extended binding-time analysis) and an *off-line* approach (in which redundant-synchronization analysis and elimination are done after PE).

We present in this paper only the *off-line* view; the advantages are a simpler binding-time analysis, an easier proof of the **correctness** theorem and also a simpler (and probably more powerful) synchronization elimination analysis. On the other hand, we suspect that the overall result may be weaker because PE cannot exploit the binding-time improvements derived from the synchronization removal.

The specialization of the communication relies on always residualizing its synchronization component. We can consider the channels as being streams of records:

```
type channel = stream of record {
                 in, out: Program_Point;
                 val: Value_Type
                 }
```

In order to lift a data transfer to the specialization time we must statically know not only the value `val` but also the program points of the send and the receive (`out` and `in`); more precisely we must know how to statically associate the correct *send* with each particular *receive*.

As for the *nondeterminism*, notice the simple setting: by disallowing global variables and multicommunication, we keep only the basic construct, the *choice*, clearly isolated in the syntax. For static choices (i.e. choices with static guards) we commit at specialization time[4], hence we define the specialization process to be nondeterministic. Dynamic choices are residualized, hence the residual program is also nondeterministic. We do *not* guarantee that our specialization preserves fairness (our formal semantics does not model fairness).

3.2 Conservative Binding-Time Analysis

This section defines a safe binding-time analysis (BTA) based on the principles stated above.

First, to clarify the terminology: from now on *static* denotes *specialization-time* while *off-line* and *on-line* refer to the moment of the binding-time analysis (i.e. before PE, respectively during PE).

In order to keep the BTA specification as simple as possible, we present a conservative off-line approach to PE. At the end of this section we comment on the possible advantages of using an on-line approach. We do not discuss standard binding-time improvements in this section.

Let us consider the simplest binding-time domain, i.e. $BT = \{S, D\}$ with $S < D$ (S standing for *static* and D for *dynamic*).

As we said before, we consider channels to be streams of records. In the spirit of simplicity, we assign only one binding-time value to a channel.

Usually a binding-time function is defined as:
$\mathcal{B}$: `SYNT_CATEG` $\rightarrow$ `BTENV` $\rightarrow$ `BTENV`

We choose to define a function for each syntactic category and to keep the binding-time environment `BTENV` parameter implicit:

- for expressions: $\mathcal{B}_e : EXPR \rightarrow BT$
- for channels: $\mathcal{B}_{CH} : CHAN \rightarrow BT$
- for program points: $\mathcal{B}_{PP} : PP \rightarrow BT$
- for (some of) the commands: $\mathcal{B}_c : COM \rightarrow BT$
- for guarded commands: $\mathcal{B}_{gc} : GCOM \rightarrow BT$

Notice that the binding-time domain BT described above is finite and hence the monotonicity of the binding-time functions is sufficient in order to guarantee the convergence of an iterative abstract interpretation algorithm.

The BTA for sequential programs assumes that binding times for the input variables are given. In a concurrent setting, we assume that binding times for input channels are given; also we *tag all observable channels as dynamic*.

[4] We are aware that heuristics can be developed but we haven't pursued this line of investigation.

The BTA for expressions is defined as usual; the only extension involves values read from a channel (assuming pointwise divisions[5] for these variables, while for channels we consider uniform divisions[6]):

$$\mathcal{B}_e[\![X \; at \; i \mid i \in PP, i : \alpha?X]\!] = \mathcal{B}_{CH}[\![\alpha]\!]$$

The BTA for channels is defined as:

$$\mathcal{B}_{CH}[\![\alpha]\!] = (\bigsqcup_{x \mid i:\alpha!x} \mathcal{B}_e[\![x]\!]) \bigsqcup$$

$$(\bigsqcup_{i \in PP \mid i:\alpha!x} \mathcal{B}_{PP}[\![i]\!]) \bigsqcup (\bigsqcup_{j \in PP \mid j:\alpha?X} \mathcal{B}_{PP}[\![j]\!])$$

where the binding time of a program point is given by:

$$\mathcal{B}_{PP}[\![i \mid i : c]\!] = \bigsqcup_{c' \in CTRL^*(c)} \mathcal{B}_c[\![c']\!]$$

Here $CTRL : COM \rightarrow COM$ is a partial function that associates with each command the control command (i.e. `alt` or `do`) that "surrounds" it. $CTRL^*$ is the transitive closure of $CTRL$ (so it gives the set of commands that control the execution of a given command).

Clearly, the binding time of a program point depends only on control commands; in order to compute binding times for these commands:

$$\mathcal{B}_c[\![\textbf{alt } gc \textbf{ end alt}]\!] = \mathcal{B}_{gc}[\![gc]\!]$$

$$\mathcal{B}_c[\![\textbf{do } gc \textbf{ end do}]\!] = \mathcal{B}_{gc}[\![gc]\!]$$

where binding times for guarded commands are (conservatively) given by:

$$\mathcal{B}_{gc}[\![b \rightarrow c]\!] = \mathcal{B}_e[\![b]\!]$$

$$\mathcal{B}_{gc}[\![b \wedge \alpha?X \rightarrow c]\!] = \mathbf{D}$$

$$\mathcal{B}_{gc}[\![b \wedge \alpha!x \rightarrow c]\!] = \mathbf{D}$$

$$\mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathcal{B}_{gc}[\![gc_0]\!] \bigsqcup \mathcal{B}_{gc}[\![gc_1]\!]$$

For the communication commands:

$$\mathcal{B}_c[\![\alpha?X]\!] = \mathcal{B}_{CH}[\![\alpha]\!]$$

$$\mathcal{B}_c[\![\alpha!x]\!] = \mathcal{B}_{CH}[\![\alpha]\!]$$

The key observation is that during specialization one can define a total order on the set of static program points. These points are under static control so, since static control is specialized, they can be assumed to be in straight-line code. This result can be extended for the static loops controlled by an always *true* condition; this extension involves using a controlled-unrolling technique described in section 4.2.

The BTA for channels is correct because we can statically perform the data transfer when we know the value and when the pairing of the program points of the send and the receive is static. We use the total order on the sets of program points to match each receive with the corresponding send.

Finally, the BTA for guarded commands is defined as above because all synchronizations are residual (hence dynamic).

Let us illustrate the conservative BTA with a short example:

```
ALT
  x > 0 & ch ! x  ->  a ! 1
  y > 0   ->  b ! 2
end ALT
```

Assume that both `x` and `y` are static. Nevertheless, since `ch!x` is residualized, the guards of the `alt` are dynamic, so the `alt` command is dynamic and hence channels `a` and `b` are dynamic - they are under dynamic control.

The use of on-line PE would lead to a more precise BTA of expressions with *mixed* binding times; in particular consider the BTA of the guard `cond & ch!x` where $\mathcal{B}_e[\![cond]\!] = \mathbf{S}$ and `cond` = **false**. Also, when using on-line PE, we can define a *speculative* BTA which is presented and commented in section 3.5.

We end with observation that hints at a finer BTA: a degenerated nondeterministic choice (with only one alternative) is in fact a *deterministic* choice. We can decouple the decision and the nondeterminism and assign them different binding times. From this perspective, the effects of improving the BTA of guards, as mentioned above, can be significant.

3.3 Specialization

The specialization of a program amounts to the specialization of the thread $MAIN$ in an environment in which the declarations of the channels and the threads are already processed.

The domains that are relevant for this specialization are shown in Fig. 3. The specialization corresponding to the denotational fragment of the semantics is standard and we omit it. The only non-standard aspect is the specialization of the top-level declarations for channels tagged as static. The channel is entered into the channel environment of the specializer and residual code defining that channel is generated. Therefore, communication on that channel will take place both at specialization time, when the data transfer is performed, and at run-time, because of the residual pure communication.

The following specialization primitives are standard (see [NP92] for instance), so we omit their definitions:

- for expressions:
  - evaluate: $\mathcal{S}_e : EXPR \rightarrow SENV \rightarrow VAL$
  - residualize: $\mathcal{S}_r : EXPR \rightarrow SENV \rightarrow CODE$
- for declarations of variables:
  $\mathcal{S}_d : DECL \rightarrow (SENV \times CODE)$

What is non-standard is the specification of the specialization using a LTS with actions. It is defined by the transition relation $\rightarrow_S$ (see Fig. 3). Note that most of the transition rules are guarded by *binding-time conditions* that are used to distinguish the *static* and the *dynamic* commands.

[5] i.e. allowing the possibility of assigning different binding-time values at different program points.

[6] i.e. one division for all program points.

We write a specializer transition in the form:

$$\Theta \xrightarrow{\lambda}_S (\Theta', action)$$

where Θ and Θ' are (the initial and the final) sets of extended configurations, λ is the label associated with this transition and *action* describes the residual code generated during this specialization step. Note that *action* is a tuple of pairs (*thread_identifier, residual_code*).

A set of extended configurations Θ is of the form:

$$< tid_0, \Delta_0 > || < tid_1, \Delta_1 > || \ldots$$

where tid_0, tid_1 are thread identifiers and Δ_0, Δ_1 are configurations (as described in section 2.2), with the observation that σ_0 and σ_1 in Δ_0 and Δ_1 denote the *static* environments. We use *thread identifiers* to *distinguish the residual code* for the various threads that are specialized in parallel.

The specification of the specialization given in Fig. 3 mirrors the LTS given in Fig. 2. Several comments need to be made:

1. Synchronization is always residualized. For static communication, i.e. $\mathcal{B}_c[\![\alpha?X]\!] = \mathbf{S}$ or $\mathcal{B}_c[\![\alpha!x]\!] = \mathbf{S}$, the residual code generated is $\alpha?DUMMY$ or $\alpha!dummy$ respectively (denoting communication with no data transfer). Note that, if the static communications lead to deadlock, then the specialization process deadlocks.

2. In the rule corresponding to the spawning of the threads, τ denotes the thread environment. Since all threads are statically created, one can get $\tau(tid)$ at specialization time for any thread identifier *tid*.

3. The rules corresponding to parallel execution describe the *interleaving* and the synchronization (by *static communication*). There is also a rule that describes the *joining* of two threads whose bodies were specialized.

4. The specialization of static **alt** and **do** choices is self-explanatory. Notice that the specialization process may *deadlock* when all guards of a static **alt** do `fail` or it may *not terminate* because of the unrolling rule for the static **do** iterations.

5. The specialization process is nondeterministic. This is reflected by the rules that describe the specialization of a compound static guarded command (i.e. **if** $\mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{S}$). If the guards of both gc_0 and gc_1 are *static* and *true*, then the nondeterminism is lifted from run-time to specialization-time.

6. During the specialization of a compound dynamic guarded command (i.e. **if** $\mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{D}$) the transitions of the specializer are *never* labeled because no static communication can take place under dynamic control; Θ used in that rule is *the same* both for the transition corresponding to gc_0 and for the one corresponding to gc_1 because the static environment cannot be updated under dynamic control.[7]

The meaning of the specialization of a program P is denoted by $\mathcal{M}_S(P)$ and is now a pair:

(eval_meaning, residual_code),

where *eval_meaning* is as before a pair: $(trace, \sigma)$ and the *residual_code* is generated as a result of the actions.

Several specialization examples are given in the Appendix.

[7] If we perform BTA using the bounded static variation improvement, then the environment σ in Θ should be computed as the least upper bound of the environments corresponding to the (bounded number of) choices.

3.4 Correctness of the Specialization

We start by defining a relation $\equiv$ (call it *equivalence* although it is neither transitive nor symmetric) between a meaning and an ordered pair of meanings (remember that a meaning is a pair $(trace, \sigma)$):

$m \equiv (m_S, m_D)$ **iff**
$m.trace = m_D.trace$ **and** $m.\sigma = m_S.\sigma \sqcup m_D.\sigma$
where: $x \sqcup \infty = \infty \sqcup y = \infty \quad (\forall x)\ (\forall y \neq \delta)$
and $x \sqcup \delta = \delta \sqcup y = \delta \quad (\forall x)\ (\forall y \neq \infty)$
and $\sigma_S \sqcup \sigma_D = \sigma_S \oplus \sigma_D$ (because they are disjoint)

Let P be a concurrent program; the **correctness** of the specialization algorithm consists of two parts:

- **soundness**: $\forall (m_S, res\_code) \in \mathcal{M}_S(P)\ \forall m_D \in \mathcal{M}(res\_code)$, $\exists m \in \mathcal{M}(P)$ s.t. $m \equiv (m_S, m_D)$
- **completeness**: $\forall m \in \mathcal{M}(P)$, $\exists (m_S, res\_code) \in \mathcal{M}_S(P)$, $\exists m_D \in \mathcal{M}(res\_code)$ s.t. $m \equiv (m_S, m_D)$

Notice that $m_S.trace$ does not occur anywhere above! This is because $m_S.trace$ is *empty*; all observable communication is residualized so there is only non-observable (internal) communication taken place at specialization time and this communication is not reflected in the trace meaning.

Theorem: Under the *conservative* BTA, the PE algorithm is correct.

Proof sketch: The idea is to state and prove a stronger result. We can define an *extended* meaning (we call the previously defined meaning the *restricted* meaning) of a concurrent program as being a trace of transitions, a transition being the triple: (initial_environment, event_label, final_environment). We can define a function that recovers our restricted meaning from the extended meaning, hence proving that equivalence of extended meanings implies equivalence of restricted meanings. For the stronger version of the theorem, that considers the extended meaning, the proof shows the equivalence between the trace of the original program and the merging of the traces of the specialization and residual program. The proof involves a layer of of mathematical induction (on the length of the traces) and one of structural induction (on the possible transitions under the specialization semantics and the transitions under the ordinary semantics).

There are several interesting **corollaries** of this theorem. We mention only:

- If (a run of) the specialization of the program P deadlocks, then there exists a run of the program P that either deadlocks or does not terminate.
- If (a run of) a specialized version of program P deadlocks, then there exists a run of the program P that deadlocks.

3.5 Speculative Binding-Time Analysis

The BTA of guarded commands described in section 3.2 is too conservative for some applications. In this section we comment on the possibilities to modify the BTA and on the implications of these modifications on the correctness of the specialization algorithm.

Let us consider the *on-line* PE, where the BTA is done during the PE. We can choose as the BT domain

$STATICVAR$ the domain of static variables VAL the domain of values
$SENV = STATICVAR \to VAL$ the environment of static variables
$XCONF = TID \times (COM \times SENV)$ the extended configurations
$LABELS = \{\alpha!x \mid \alpha \in CHAN \text{ and } x \in VAL\} \cup \{\alpha?x \mid \alpha \in CHAN \text{ and } x \in VAL\}$
$CODE$ the domain of source programs; we can concatenate code using the infix operator $+ : CODE \times CODE \to CODE$
$ACTIONS = (TID \times CODE)^N$ the domain of actions
$\to_{S_t} = THREAD \to (XCONF \times ACTIONS)$ start the specialization of a thread
$\to_S \subset XCONF^N \times LABELS \times ((XCONF^M \cup \{\textbf{fail}\}) \times ACTIONS)$

$$\frac{\mathcal{S}_d[\![d]\!] = (\sigma, decl\_code)}{< \textbf{thread } tid \textbf{ is } d;\ c \textbf{ end} > \to_{S_t} (< tid, < c, \sigma >>, (tid, \text{“}\textbf{thread } tid \textbf{ is}\text{”} + decl\_code))}$$

$$< tid, < \textbf{skip}, \sigma >> \to_S (< tid, \sigma >, null)$$

$$\textbf{if } \mathcal{B}_c[\![X := a]\!] = \textbf{S} \quad \frac{\mathcal{S}_e[\![a]\!]\sigma = x}{< tid, < X := a, \sigma >> \to_S (< tid, \sigma[x/X] >, (tid, null))}$$

$$\textbf{if } \mathcal{B}_c[\![X := a]\!] = \textbf{D} \quad \frac{\mathcal{S}_r[\![a]\!]\sigma = res\_a}{< tid, < X := a, \sigma >> \to_S (< tid, \sigma >, (tid, \text{“}X := \text{”} + res\_a))}$$

$$\textbf{if } \mathcal{B}_c[\![\alpha?X]\!] = \textbf{S} \quad < tid, < \alpha?X, \sigma >> \overset{\alpha?x}{\to}_S (< tid, \sigma[x/X] >, (tid, \text{“}\alpha?DUMMY\text{”})$$

$$\textbf{if } \mathcal{B}_c[\![\alpha?X]\!] = \textbf{D} \quad < tid, < \alpha?X, \sigma >> \to_S (< tid, \sigma >, (tid, \text{“}\alpha?X\text{”}))$$

$$\textbf{if } \mathcal{B}_c[\![\alpha!a]\!] = \textbf{S} \quad \frac{\mathcal{S}_e[\![a]\!]\sigma = x}{< tid, < \alpha!a, \sigma >> \overset{\alpha!x}{\to}_S (< tid, \sigma >, (tid, \text{“}\alpha!dummy\text{”}))}$$

$$\textbf{if } \mathcal{B}_c[\![\alpha!a]\!] = \textbf{D} \quad \frac{\mathcal{S}_r[\![a]\!]\sigma = res\_a}{< tid, < \alpha!a, \sigma >> \to_S (< tid, \sigma >, (tid, \text{“}\alpha!\text{”} + res\_a))}$$

$$\frac{< tid, < c_0, \sigma >> \overset{\lambda}{\to}_S (< \_, < c_0', \sigma' >>, act)}{< tid, < c_0; c_1, \sigma >> \overset{\lambda}{\to}_S (< tid, < c_0'; c_1, \sigma' >>, act)}$$

$$\frac{t_0 = \tau(tid_0) \quad t_1 = \tau(tid_1) \quad t_0 \to_{S_t} (\Theta_0, (tid_0, decl_0)) \quad t_1 \to_{S_t} (\Theta_1, (tid_1, decl_1))}{< tid, < tid_0 \parallel tid_1, \sigma >> \to_S ((\Theta_0' \parallel \Theta_1),\ ((tid, \text{“}tid_0 \parallel tid_1\text{”}), (tid_0, decl_0), (tid_1, decl_1)))}$$

where $\Theta_0 =< tid_0, < c_0, \sigma_0 >>$ **and** $\Theta_0' =< tid_0, < c_0, \sigma \oplus \sigma_0 >>$ **and** $\Theta_1 =< tid_1, < c_1, \sigma_1 >>$

$$\frac{< tid_0, < c_0, \sigma_0 >> \overset{\lambda}{\to}_S (\Theta_0, (tid_0, code))}{(< tid_0, < c_0, \sigma_0 >>) \parallel (< tid_1, < c_1, \sigma_1 >>) \overset{\lambda}{\to}_S ((\Theta_0 \parallel (< tid_1, < c_1, \sigma_1 >>)),\ (tid_0, code))} \quad \textit{and its dual}$$

where $\Theta_0 =< tid_0, < c_0', \sigma_0' >>$

$$\frac{< tid_0, < c_0, \sigma_0 >> \overset{\alpha?x}{\to}_S (\Theta_0, (tid_0, code_0)) \quad < tid_1, < c_1, \sigma_1 >> \overset{\alpha!x}{\to}_S (\Theta_1, (tid_1, code_1))}{(< tid_0, < c_0, \sigma_0 >>) \parallel (< tid_1, < c_1, \sigma_1 >>) \to_S ((\Theta_0 \parallel \Theta_1),\ ((tid_0, code_0), (tid_1, code_1)))} \quad \textit{and its dual}$$

where $\Theta_0 =< tid_0, < c_0', \sigma_0' >>$ **and** $\Theta_1 =< tid_0, < c_1', \sigma_1 >>$

$$(< tid_0, \sigma_0) \parallel (< tid_1, \sigma_1 >) \to_S (< tid_0, \sigma_0 >, null)$$

$$\textbf{if } \mathcal{B}_{gc}[\![gc]\!] = \textbf{S} \quad \frac{< tid, < gc, \sigma >> \overset{\lambda}{\to}_S (< tid, < c, \sigma' >>, (tid, code))}{< tid, < \textbf{alt } gc \textbf{ end alt}, \sigma >> \overset{\lambda}{\to}_S (< tid, < c, \sigma' >>, (tid, code))}$$

$$\textbf{if } \mathcal{B}_{gc}[\![gc]\!] = \textbf{D} \quad \frac{< tid, < gc, \sigma >> \to_S (< tid, \sigma' >, (tid, code))}{< tid, < \textbf{alt } gc \textbf{ end alt}, \sigma >> \to_S (< tid, \sigma' >, (tid, \text{“}\textbf{alt}\text{”} + code + \text{“}\textbf{end alt}\text{”}))}$$

$$\textbf{if } \mathcal{B}_{gc}[\![gc]\!] = \textbf{S} \quad \frac{< tid, < gc, \sigma >> \overset{\lambda}{\to}_S (< tid, < c, \sigma' >>, (tid, code))}{< tid, < \textbf{do } gc \textbf{ end do}, \sigma >> \overset{\lambda}{\to}_S (< tid, < c; \textbf{do } gc \textbf{ end do}, \sigma' >>, (tid, code))}$$

$$\textbf{if } \mathcal{B}_{gc}[\![gc]\!] = \textbf{S} \quad \frac{< tid, < gc, \sigma >> \to_S < tid, \textbf{fail} >}{< tid, < \textbf{do } gc \textbf{ end do}, \sigma >> \to_S (< tid, \sigma >, null)}$$

$$\textbf{if } \mathcal{B}_{gc}[\![gc]\!] = \textbf{D} \quad \frac{< tid, < gc, \sigma >> \to_S (< tid, \sigma >, (tid, code))}{< tid, < \textbf{do } gc \textbf{ end do}, \sigma >> \to_S (< tid, \sigma >, (tid, \text{“}\textbf{do}\text{”} + code + \text{“}\textbf{end do}\text{”}))}$$

Figure 3: Specification of the specialization

$$\text{if } \mathcal{B}_e[\![b]\!] = \mathbf{S} \quad \frac{\mathcal{S}_e[\![b]\!]\sigma = \mathbf{false}}{< tid, < b \,/\, b \wedge \alpha?X \,/\, b \wedge \alpha!a \rightarrow c, \sigma >> \rightarrow_S < tid, \mathbf{fail} >}$$

$$\text{if } \mathcal{B}_e[\![b]\!] = \mathbf{S} \quad \frac{\mathcal{S}_e[\![b]\!]\sigma = \mathbf{true}}{< tid, < b \rightarrow c, \sigma >> \rightarrow_S (< tid, < c, \sigma >>, null)}$$

$$\text{if } \mathcal{B}_e[\![b]\!] = \mathbf{D} \quad \frac{\mathcal{S}_r[\![b]\!]\sigma = res\_b}{< tid, < b \rightarrow c, \sigma >> \rightarrow_S (< tid, < c, \sigma >>, (tid, res\_b + \text{" } \rightarrow \text{ "}))}$$

$$\frac{\mathcal{S}_r[\![b]\!]\sigma = res\_b}{< tid, < b \wedge \alpha?X \rightarrow c, \sigma > \rightarrow_S (< tid, < c, \sigma >>, (tid, res\_b + \text{" } \wedge \alpha?X \rightarrow \text{ "}))}$$

$$\frac{\mathcal{S}_r[\![b]\!]\sigma = res\_b \quad \mathcal{S}_r[\![a]\!]\sigma = res\_a}{< tid, < b \wedge \alpha!a \rightarrow c, \sigma > \rightarrow_S (< tid, < c, \sigma >>, (tid, res\_b + \text{" } \wedge \alpha!\text{"} + res\_a + \text{" } \rightarrow \text{ "}))}$$

$$\text{if } \mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{S} \quad \frac{< tid, < gc_0, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code)}{< tid, < gc_0 [\!] gc_1, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))} \quad \textit{and its dual}$$

$$\text{if } \mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{D} \quad \frac{< tid, < gc_0, \sigma >> \rightarrow_S (\Theta, (tid, code_0)) \quad < tid, < gc_1, \sigma >> \rightarrow_S (\Theta, (tid, code_1))}{< tid, < gc_0 [\!] gc_1, \sigma >> \rightarrow_S (\Theta, (tid, code_0 \text{ "[]" } code_1))}$$

$$\textbf{where} \quad \Theta = < tid, \sigma >$$

$$\frac{< tid, < gc_0, \sigma >> \rightarrow_S < tid, \mathbf{fail} > \quad < tid, < gc_1, \sigma >> \rightarrow_S < tid, \mathbf{fail} >}{< tid, < gc_0 [\!] gc_1, \sigma >> \rightarrow_S < tid, \mathbf{fail} >}$$

Figure 3: Specification of the specialization (continued)

for boolean expressions the 3-element domain $\{F, T, D\}$ where F denotes a static *false* value, T denotes a static *true* value and D denotes a dynamic value. The partial order for this BT domain is given by: $F < D$, $T < D$.

First, let us consider the BT rule:

$$\mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathcal{B}_{gc}[\![gc_0]\!] \bigsqcup \mathcal{B}_{gc}[\![gc_1]\!]$$

We can replace it by:

$$\mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \begin{cases} F & if\ (\mathcal{B}_{gc}[\![gc_0]\!] = F) \wedge (\mathcal{B}_{gc}[\![gc_1]\!] = F) \\ T & if\ (\mathcal{B}_{gc}[\![gc_0]\!] = T) \vee (\mathcal{B}_{gc}[\![gc_1]\!] = T) \\ D & else \end{cases}$$

The intuition is the following:

- if all guards are static and *false* then the BT is obviously F;
- if some of the guards are static and *true* then we will *speculatively* choose one of the corresponding guarded commands and *commit at specialization time*; hence BT is T;
- if all static guards are false and there is also at least one dynamic guard, then the choice is dynamic so the BT is D.

The modifications of the specialization specification are given in Fig. 4.

The consequence of committing on a specific branch of a choice at specialization time, although some of the guards of that choice are dynamic, is that the specialization algorithm is *no longer complete*. This happens because some of the nondeterminism of the initial program is lost during this speculative specialization. Notice that we can still prove the *soundness* part of the correctness theorem, but clearly the *completeness does not hold anymore*. This type of BTA is useful for programs that exhibit a *don't care* type of nondeterminism (see such a specialization example in Appendix B).

There appears to be also a second possible modification of the BTA of guards. The rules: $\mathcal{B}_{gc}[\![b \wedge \alpha?X \rightarrow c]\!] = \mathbf{D}$ and $\mathcal{B}_{gc}[\![b \wedge \alpha!x \rightarrow c]\!] = \mathbf{D}$ seem overly conservative. However, this are the only sound rules for BTA. In order to illustrate this, let us consider the example given in Fig. 5.

Assume that the expressions `cond1, cond2, cond3` and `cond4` are all dynamic and that the threads are synchronized at program points `start0, start1` and `start2`. According to the conservative BTA, channels **a** and **b** are dynamic because some communication on these channels is under the control of the **alt** in `thread0` which is tagged as dynamic because of the residual synchronizations that correspond exactly to channels **a** and **b**. It appears that we can do better and tag these two channels as static! If we do this, then the **if** statements from threads 1 and 2 are dynamic and hence residualized; the **alt** in `thread0` is static, the specialization is nondeterministic. Let us assume, w.l.o.g. that it commits to the first choice. This may be *unsound* because `cond1` and `cond4` may be true while `cond2` and `cond3` may be false and the commitment on the first choice would be impossible in the original program because of the synchronization of the threads 1 and 2 on communication channel **c**. A similar argument shows that committing to the second choice of the **alt** is equally unsound.

It is clear that, for guards that include communication, only an analysis of the synchronization patterns can lead to a correct assignment of their binding time. In this paper we choose to residualize all synchronizations and conservatively associate binding-time *dynamic* for all guards that include communication operations. Clearly an analysis of redundant synchronizations done *on-line*, during BTA, would improve the BTA. On the

$$\text{if } \mathcal{B}_{gc}[\![gc]\!] = \mathbf{T} \quad \frac{< tid, < gc, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))}{< tid, < \textbf{alt } gc \textbf{ end alt}, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))}$$

$$\text{if } \mathcal{B}_{gc}[\![gc]\!] = \mathbf{T} \quad \frac{< tid, < gc, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))}{< tid, < \textbf{do } gc \textbf{ end do}, \sigma >> \xrightarrow{\lambda}_S (< tid, < c; \textbf{do } gc \textbf{ end do}, \sigma' >>, (tid, code))}$$

$$\text{if } \mathcal{B}_{gc}[\![gc]\!] = \mathbf{F} \quad \frac{< tid, < gc, \sigma >> \xrightarrow{\lambda}_S < tid, \textbf{fail} >}{< tid, < \textbf{do } gc \textbf{ end do}, \sigma >> \xrightarrow{\lambda}_S (< tid, \sigma >, null)}$$

$$\text{if } \mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{T} \quad \frac{< tid, < gc_0, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))}{< tid, < gc_0 [\!] gc_1, \sigma >> \xrightarrow{\lambda}_S (< tid, < c, \sigma' >>, (tid, code))} \quad \textit{and its dual}$$

$$\text{if } \mathcal{B}_{gc}[\![gc_0 [\!] gc_1]\!] = \mathbf{F} \quad \frac{< tid, < gc_0, \sigma >> \rightarrow_S < tid, \textbf{fail} > \quad < tid, < gc_1, \sigma >> \rightarrow_S < tid, \textbf{fail} >}{< tid, < gc_0 [\!] gc_1, \sigma >> \rightarrow_S < tid, \textbf{fail} >}$$

Figure 4: Specialization under speculative BTA

```
thread 0:                    thread 1:                    thread 2:
 start0:                      start1:                      start2:
   ALT                          IF cond1 c!7 end IF          IF cond2 d!9 end IF
     true & a?X -> cont(X)      a!5                          b?y
     true & b!1 -> smthing      IF cond3 d?m end IF          IF cond4 c?n end IF
   end ALT                      ...                          ...
   ...
```

Figure 5: Example: Assigning binding times to guarded commands

other hand this redundant-synchronization analysis (RSA) would be approximate since the dependencies on which the RSA is based cannot be exactly computed before the whole residual program is available.

4 Pragmatic Concerns

4.1 Language Extensions

The CSP-language presented is complete in the sense that standard commands such as the deterministic decision **if** and the iteration **while** can be expressed in terms of the non-iterated choice **alt** and the iterated choice **do**. However, adding the **if** and the **while** constructs is more than syntactic sugar since it allows us to *identify determinism syntactically.* There is no problem in extending our framework to include the **if** and the **while** commands; they are control commands, so the BTA of their conditions will affect the BTA of the program points in their bodies, and their specialization is standard (see [NP92]).

Another useful language extension is allowing data structures such as arrays and structures. The problem is the binding-time separation and it is discussed in [JGS93]; see also Mogensen [Mog88] and Romanenko [Rom90].

Finally, we may use commands executing in parallel without including them in threads as long as we can statically assign unique identifiers to these commands.

4.2 Controlling Loop Unrolling

This section does not address the problem of analyzing and improving the termination properties of our specializer. This is a very hard problem and research on this topic is described by Jones in [JGS93] and in a recent paper [Jon96].

We study here a problem that is specific to concurrent programs. In a sequential language the specialized program point captures all the information needed for folding (i.e. limiting the unrolling of) a loop. In a concurrent language, because the communication topology is not captured into the specialized program point, the folding is a little tricky. This folding must ensure the proper pairing of the *static* send and receive communications that are in the body of a statically controlled loop.

We'll use counting arguments. The same idea was used by Mercouroff [Mer91] in his work. The difference is that our counting is more restricted, targeting only static communication and is precise. Mercouroff's analysis is used to detect deadlock and is approximate.

Before going further, we make three observations:

- the only static loop we have to deal with is the *while true* loop; all other static loops are unrolled completely;
- if we have several levels of *while true* loops, only the innermost level counts; the control will never leave the innermost level so the other static loops can be ignored.
- static communication cannot occur under dynamic control (see the BTA section 3.2).

Let us consider an arbitrary unrolling transformation of the loop:

while true *body* **end while**

we get:

$body_1 \ldots body_m$ **while true** $body_1 \ldots body_n$ **end while**

The pair of integers (m, n) (with $0 \leq m$ and $1 \leq n$) *uniquely* characterizes this unrolling.

Now, let us consider the following example (program points `p0` and `p1` are synchronized and the only communication is carried on channel `CH`):

```
-- thread T0                -- thread T1
p0:                         p1:
   while true                  CH ? X
      ...                      while true
      CH ! 100                    ...
      ...                         CH ? Y
      CH ! 11                     ...
      ...                      end while
   end while
```

We can write a set of integer equations. The pair of equations for each static channel reflects the count of the communications that take place:

$$A_0 + B_0 * m_0 = A_1 + B_1 * m_1$$

$$B_0 * n_0 = B_1 * n_1$$

where $A_0 = 0$ is the number of communications on CH occurring in `T0` before the *while true* loop; $B_0 = 2$ is the number of communications in `T0` on CH occurring inside the *while true* loop; similarly for thread `T1`: $A_1 = 1$ and $B_1 = 1$.

This arises because of the desire to associate each static receive statement with a single static send statement and vice-versa.

We compute: (m_0, n_0) that characterize the unrolling of the loop in thread `T0`; similarly (m_1, n_1) for thread `T1`. Notice that the two systems of equations (in m and n) are *independent*.

If the system of equations doesn't have solutions then we can prove *deadlock* (circular waiting); else we pick the minimal solution. In our example the minimal solution is $m_0 = 1; n_0 = 1; m_1 = 1; n_1 = 2$, leading to:

```
-- thread T0                -- thread T1
p0:                         p1:
   ...                         CH ? X
   CH ! 100                    ...
   ...                         CH ? Y
   CH ! 11                     ...
   ...                         while true
   while true                     ...
      ...                         CH ? Y
      CH ! 100                    ...
      ...                         ...
      CH ! 11                     CH ? Y
      ...                         ...
   end while                   end while
```

4.3 Binding-Time Improvements

Several standard methods can be used to improve the BTA. In some cases, we can exploit the bounded static variation of the matching between the program points of the sender and the receiver. In other cases, splitting the channel into a static stream and a dynamic stream can be useful.

We comment in some detail on one binding-time improvement that we find interesting.

Assume that the data transferred on a channel `CH` is of type `T`, a type which has a finite (and small) set of values; refer to them as: `val_0` ... `val_N`.

We can use "the trick" for both the send and the receive (the **alt** plays the role of the **case** from the deterministic setting); the transformation of the receive is more interesting:

```
x: SOME_TYPE(val_0,...,val_N)
CH ? x
continuation(x)
```

yields something like:

```
ALT
  CH ? val_0 -> continuation(val_0)
     ...
  CH ? val_N -> continuation(val_N)
end ALT
```

We can go further and replace channel `CH` with several channels - let's call them `CH_0` ... `CH_N` - and do the transformations:

```
CH ! val_0   ===>   CH_0 ! NO_VALUE
CH ? val_0   ===>   CH_0 ? NO_VALUE
...
CH ! val_N   ===>   CH_N ! NO_VALUE
CH ? val_N   ===>   CH_N ? NO_VALUE
```

This sequence of transformations is strikingly similar to the ones used by Milner (see [Mil89]) to translate the general CCS into *pure* CCS (having only pure communication with no exchange of values).

Notice that here the transformation is used for a more pragmatic reason than Milner's: *improving the binding-time properties* and *effectively specializing* the continuation with respect to the concrete values.

An example of specialization that uses the transformation described above in given in Appendix A.2.

5 Removing Redundant Synchronizations

The relationship between synchronization and nondeterminism is central when studying concurrent systems. Synchronizations are the means of reducing the nondeterminism. A synchronization is called *redundant* if its removal does not increase the nondeterminism of the program.

The problem of correctly removing synchronizations during or after PE is complex; blindly lifting synchronizations to the specialization time may increase the nondeterminism and is therefore incorrect.

Research on removing synchronizations has been conducted for improving the performance of data-parallel languages [GS96b]. In these languages the nondeterminism is a result of the use of global variables. In this paper, the problem is specified at the basic level: the relation between *synchronization* and *choice*.

This section proceeds as follows: we characterize the dependencies between program points of different

threads, then we present an analysis that tags redundant synchronizations (we may call them *pure communications* borrowing a term from Milner [Mil89]) and finally we sketch an algorithm for synchronization removal.

Let's consider an example:

```
--thread T0        --thread T1    --thread T2
start0:            start1:        start2:
  ALT              i1:...         i2:...
    a?X -> ..      j1:a!0         j2:b!1
    b?Y -> ..      k1:...         k2:...
  end ALT
```

Assuming that `i1` and `k2` are synchronized, we can infer that thread `T0` is deterministic (`b?1` must be chosen). Removing this synchronization will change the meaning of the program by increasing its nondeterminism (because `a?0` can now be chosen also).

The key observation is that there are dependencies between program points in threads `T1` and `T2` because some of the communications of these two threads are complementary[8] to communications occurring in guards of a choice that is executed in a parallel thread.

Hence, a simple conservative analysis can collect the set of all pairs of communications that occur in the guards of all `alt` and `do` commands. Using this set of pairs one can characterize the dependency between any program point `p1` from a thread `T1` and program point `p2` from thread `T2`.

This characterization is *safe* (captures all dependencies between program points), but is *approximate* for several reasons such as: the guards may have boolean conditions attached or it may be the case that, although apparently executing in parallel, in an actual run, the choice that was used as a witness of the dependency will never be executed in parallel with the commands corresponding to the two program points that are tagged as dependent.

We extend this point-to-point dependency relation to sets of program points. Two sets of program points S_0 and S_1 are dependent if $\exists p_0 \in S_0, \exists p_1 \in S_1$ such that p_0 and p_1 are dependent.

FACT: Synchronizations that are targeted for removal are *not* under dynamic control because they resulted after specialization. Therefore they are: either at top level or under a **while true** control, because this is the only static control that remains after PE. There are two types of dependencies that are associated with these **while true** loops: loop independent and loop carried; to capture both of them, in the absence of arrays, it is sufficient to do a transformation that replaces a **while true** *body* **end while** with the sequence *body; body.*

So, without the loss of generality, we assume that we are dealing with straight line code. Therefore for each thread, the set of program points associated with pure communications is totally ordered; each synchronization is characterized by a pair of such program points.

Let us consider synchronization $\Sigma = \{i_0, i_1\}$. Tag the synchronization as redundant or essential based on the dependencies between the sets of program points:

$(before(i_0) - before(i_1))$ and $(after(i_1) - after(i_0))$ and

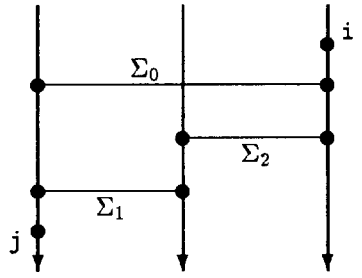
$(before(i_1) - before(i_0))$ and $(after(i_0) - after(i_1))$

[8]The send $\alpha!x$ is complementary to the receive $\alpha?X$ and vice-versa.

The computation of the sets *before* and *after* takes into account the other synchronizations that are in the program, so the removal of synchronization changes these sets and hence the tagging of the other synchronizations as residual or essential.

Note that if we lift this analysis to the specialization time we have to approximate the program point sets *after*; for trace-based abstract interpretations see [Col95, CL96].

The simplest synchronization removal algorithm is a naive greedy algorithm that removes one redundant synchronization at a time and updates the tagging of the remaining synchronizations. It computes a maximal set of synchronizations to be removed. However, it fails in finding the best solution to the problem as one can see from the following example. Note that the vertical lines represent threads, as straight-line code, while the horizontal lines represent synchronizations.



Assume that there is a dependency between program points `i` and `j`. The simple analysis presented here tags both Σ_0 and Σ_1 as redundant. If we remove Σ_0 then both Σ_1 and Σ_2 are essential. On the other hand, if we remove Σ_1, then Σ_2 is redundant and a better solution is obtained.

6 Conclusions and Future Work

We have presented the specification of a partial evaluator for a simple concurrent language. The examples given have shown that it is possible to automatically specialize concurrent programs of a reasonable complexity. We believe that the results presented here are relevant for a wide variety of concurrent languages; for instance we think our methods can be applied to object oriented concurrent languages like POOL (see [AR89]).

The performance of the post-specialization synchronization removal can be improved. On the other hand, we'd like to integrate an analysis of the synchronizations into the BTA; this involves a less precise synchronization analysis but the overall quality may be higher because a more powerful specializer may result.

We are also investigating the possibility of extending our framework for languages that support dynamic communication topologies. An analysis of the communication topology of a rather complex concurrent language such as CML (see [Rep91]), was presented by Nielson and Nielson in [NN94] and we are looking into integrating this analysis into the binding-time analysis and exploiting the potential of bounded-static-variation-based techniques.

Acknowledgements: We are grateful to Olivier Danvy and the anonymous reviewers for their comments.

We also like to thank Bob Paige for the discussions on programming language semantics and program transformations.

References

[AR89] P. America and J. Rutten. A parallel object oriented language: Design and semantic foundations. In J. W. de Bakker, editor, *Languages for Parallel Architectures*. John Willey, 1989.

[Ber90] A. Berlin. Partial evaluation applied to numerical computation. In *The 1990 ACM Conference on LISP and Functional Programming*, pages 139–150, 1990.

[CD93] C. Consel and O. Danvy. Tutorial notes in partial evaluation. In *The 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, 1993.

[CL96] C. Colby and P. Lee. Trace-based program analysis. In *The 23th Annual ACM Symposium on Principles of Programming Languages*, pages 195–207, 1996.

[Col95] C. Colby. Analyzing the communication topology of concurrent programs. In *The ACM Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 202–214, 1995.

[CPW93] C. Consel, C. Pu, and J. Walpole. Incremental specialization: The key to high performance, modularity and portability in operating systems. In *The ACM Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, pages 44–46, 1993.

[dFS96] N. de Francesco and A. Santone. Unfold/Fold Transformations of Concurrent Processes. In *LNCS nr. 1140*, pages 167–181. Springer Verlag, 1996.

[FOF88] H. Fujita, A. Okamura, and K. Furukawa. Partial evaluation of GHC programs based on the UR-set with constraints. In *Logic Programming 5th International Conference and Symposium*, pages 924–941, 1988.

[GS96a] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In *Partial Evaluation, LNCS nr. 1110*, pages 137–160. Springer Verlag, 1996.

[GS96b] M. Gupta and E. Schonberg. Static analysis to reduce synchronization costs in data-parallel programs. In *The 23th Annual ACM Symposium on Principles of Programming Languages*, pages 322–332, 1996.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[Jon96] N. Jones. What not to do when writing an interpreter for specialization. In *Partial Evaluation, LNCS nr. 1110*, pages 216–237. Springer Verlag, 1996.

[Ltd84] (INMOS Ltd.). *Occam Programming Manual*. Prentice Hall, 1984.

[May83] D. May. Occam. *Sigplan Notices*, 13(4), 1983.

[Mer91] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics, LNCS nr. 598*, pages 312–325. Springer Verlag, 1991.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 94–105, 1991.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North Holland, 1988.

[Mos90] P.D. Mosses. Denotational semantics. In van Leeuwen, editor, *Handbook of Theoretical Computer Science*. M.I.T. Press, 1990.

[NN94] H.R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *The 21st Annual ACM Symposium on Principles of Programming Languages*, pages 84–97, 1994.

[NP92] V. Nirkhe and W. Pugh. Partial evaluation of high-level imperative languages, with applications in hard real-time systems. In *The 19th Annual ACM Symposium on Principles of Programming Languages*, pages 269–280, 1992.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *LNCS nr. 104*, pages 167–173. Springer Verlag, 1981.

[PC+95] C. Pu, C. Consel, et al. Optimistic incremental specialization: Steamlining a commercial operating system. In *ACM Symposium on Operating Systems Principles*, 1995.

[Pep93] P. Pepper. Deductive Derivation of Parallel Programs. In *Parallel Algorithm Derivation and Program Transformation*, pages 1–53. Kluwer Academic Publishers, 1993.

[Plo83] G.D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 199–225. North Holland, 1983.

[Rep91] J.H. Reppy. CML: A Higher-Order Concurrent Language. In *ACM Symposium Programming Language Design and Implementation*, 1991.

[Rom90] S. Romanenko. Arity raiser and its use in program specialization. In N. Jones, editor, *ESOP'90, LNCS nr. 432*, pages 341–360. Springer Verlag, 1990.

[RS90] J. Reif and S. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, 1990.

[Tur93] V.F. Turchin. Program transformations with metasystem transition. *Journal of Functional Programming*, 3(3):283–313, 1993.

[Win93] G Winskel. *The Formal Semantics of Programming Languages*, pages 297–336. MIT Press, 1993.

A Example: Matrix Multiplication

A simple concurrent program illustrates the PE techniques developed in this paper. The program is given in Fig. 6 and it is written in Occam (suitably extended to accommodate declarations needed for PE and to improve readability).

This program implements the multiplication of matrices `a[M,N]` and `b[N,Q]` by:

- associating thread `P[i,j]` with element `a[i,j]`
- piping the line `j` of the matrix `b` (prefixed by a suitably chosen number of 0's) to each thread `P[i,j]`
- each line `i` of the product matrix is piped out on channel `CH[i,N]` .

Two possible specializations are described.

A.1 Triangular Matrices

Let's assume that matrix **a** from thread `InputA` is partly static. For instance, consider specializing the multiplication algorithm in the case when matrix **a** is superior triangular, i.e.

$$\forall i > j,\ a[i,j] = 0$$

A fully automatic, *on-line* PE, using techniques described in this paper, goes as follows:

- the BTA for expressions leads to (the static environment is denoted by σ):

  $\forall i > j$ in `thread P[i,j]`

  $\mathcal{B}_e[\![s]\!] = \mathcal{B}_e[\![a]\!] = \mathbf{S}$; and $\sigma(s) = \sigma(a) = 0$

  also $\forall i$ in `thread P[i,i]`

  $\mathcal{B}_e[\![s]\!] = \mathbf{S}$ and $\sigma(s) = 0$

- a simple dataflow analysis (because there are no globals) spots the useless variables (such as `b`) and channels (such as $B[i,j]$) $\forall i > j$

```
INT M IS ...
INT N IS ...
INT Q IS ...
-- matrix a of M lines N columns;
-- matrix b of N lines Q columns
-- Q may be the total number of columns
--   of several matrices b that are pipelined

[M,N] REAL CHANNEL A:
-- for distributing the elements of matrix a
[M,N] REAL CHANNEL B:
-- pipeline the j-th line of matrix b
--         through channel B[_,j]
[M,N] REAL CHANNEL CH:

INT i, j:
DECLARE i := 1 FOR M
   DECLARE j := 1 FOR N
      CHANNEL A[i,j] = (InputA, P[i,j])
      CHANNEL B[i,j] = (InputB, P[i,j])
   end DECLARE
   DECLARE j := 0 FOR N-1
      CHANNEL CH[i,j] = (P[i,j], P[i,j+1])
   end DECLARE
   CHANNEL CH[i,N] = (P[i,N], Output)
end DECLARE

-- the program consists of
--       the parallel execution of threads
--       InputA, InputB, MAIN and Output

-- the result: line i of matrix product
-- is pipelined out on channel CH[i,N]
-- therefore, the observable communication
-- is on  CH[i,N]  forall 0 < i <= M

THREAD MAIN IS
   INT i, j:
   PAR i := 1 FOR M
      PAR j := 0 FOR N
         P[i,j]
      end PAR
   end PAR
end THREAD

DECLARE i := 1 FOR M
   DECLARE j := 1 FOR N
      THREAD P[i,j] IS
         REAL a, b, s:
         A[i,j] ? a
         WHILE true
            SEQ
               CH[i,j-1] ? s
               B[i,j] ? b
               CH[i,j] ! s + a * b
            end SEQ
         end WHILE
      end THREAD
   end DECLARE
end DECLARE
```

Figure 6: Matrix Multiplication Program

```
DECLARE i := 1 FOR M
   THREAD P[i,0] IS
      WHILE true
         CH[i,0] ! 0
      end WHILE
   end THREAD
end DECLARE

THREAD InputA IS
   INT i, j:
   [M,N] REAL a: -- matrix a
   PAR i := 1 FOR M
      PAR j := 1 FOR N
         A[i,j] ! a[i,j]
      end PAR
   end PAR
end THREAD

THREAD InputB IS
   INT i, j, k:
   [N,Q] REAL b: -- matrix b
   SEQ k := 1 FOR Q
      PAR i := 1 FOR N
         PAR j := 1 FOR M
            IF k+1-j <= 0
               B[i,j] ! 0
            ELSE
               B[i,j] ! b[j,k+1-j]
            end IF
         end PAR
      end PAR
   end SEQ
end THREAD
```

Figure 6: Matrix Multiplication Program (continued)

- BTA tags all program points as static while for channels: $\forall i > j$
  $\mathcal{B}_{CH}[\![CH[i,j]]\!] = \mathcal{B}_{CH}[\![A[i,j]]\!] = \mathbf{S}$;
- as a result of the specialization of the data transfer, the communication on the static and useless channels is residualized as pure communication;
- all pure communications are removed because they are redundant;
- after the clean-up phase (in which empty threads are discarded), a significant drop in the number of threads and communication channels is achieved; the residual program uses only:
  - threads `P[i,j]` with `i` $\leq$ `j`
  - channels `CH[i,j]` (with `i` $\leq$ `j`) and input on `A[i,j]` and `B[i,j]` (with `i` $\leq$ `j`).

A.2 Boolean Matrices

This second PE example illustrates the specialization of the general matrix multiplication algorithm for boolean matrices. We can use PE techniques although no static input data is given because of the *bounded static variation* of the values of the elements of the matrices.

The residual code for threads `P` and `InputB` is given in Fig. 7 and is based on the transformation that was presented in the Binding-Time Improvements section.

Notice that during the specialization only *some* of the channels `B[i,j]` are tagged as useless (the ones corresponding to elements `a[i,j] = 0`); therefore, if we want to remove this channels, we need to insert the decision `if a[i,j]=1` in the code of thread `InputB` (see Fig. 7). In order to do this transformation we should either consider matrix **a** as global (OK since it is read-only data) or explicitly pass it to thread `InputB`.

What is remarkable about this transformation is that it can be interpreted as *trading communication for computation*. A naive complexity analysis shows how difficult it is to accurately evaluate the performance of the specializer: some communication was eliminated as well as some computation (additions and multiplications); however, a decision was inserted.

B Example: Nondeterministic Sorting

We use an example to illustrate the specialization of the nondeterministic choice. The focus is on *nondeterminism* so, for clarity reasons, we consider a program with no communication. The program (see Fig. 8) has only one thread which implements the sorting of 4 numbers.

We can assign two distinct meanings to this program:

- if we are interested only in obtaining the 4 numbers in non-increasing order, then the only meaningful communication is on channel `outnr`; the original program doesn't need to include any communication on channel `outstr`; consequently the nondeterministic choice is a *don't care* type of choice;
- if we are interested in the particular order in which the swaps of the numbers was done then the communication on channel `outstr` is meaningful because if offers the log of the sorting algorithm.

Now, assume that **x1**, **x2** and **x3** are static with:
`x1 = 1;  x2 = 2;  x3 = 3`
and **x4** is dynamic.

The residual programs showed in Fig. 9 can be automatically obtained by using the *speculative* BTA for the guards described in section 3.5. Notice that:

1. The PE is *nondeterministic*; there are two possible residual programs that can be generated depending on the commitment on a specific choice at specialization time;

2. The PE is *sound* but *not complete*; it is correct only if we consider that the meaning of the program is *just* the communication on channel `outnr`; otherwise it is clear that, using this specialization, we loose some of the nondeterminism of the initial program; for example if `x4 = 4` then, by using the specialization described above, we are *not* able to obtain a trace of execution in which the first swap is the one between **x3** and **x4**.

A final remark: in order to perform this specialization we need to use *constraint-based* information propagation (see [GS96a]) instead of *constant propagation*. This is necessary in order to tag the condition `2 < x3` as *static* and *true* on the true branch of the conditional `if 2 < x3`.

```
-- a and b are to be redeclared as BOOLEAN

DECLARE i := 1 FOR M
DECLARE j := 1 FOR N
THREAD P[i,j] IS
  ALT
    A[i,j] ? 0
      WHILE true
        ALT
          CH[i,j-1] ? 0 -- B[i,j] ? dummy
            CH[i,j] ! 0 -- is eliminated
          CH[i,j-1] ? 1 -- B[i,j] ? dummy
            CH[i,j] ! 1 -- is eliminated
        end ALT
      end WHILE
    A[i,j] ? 1
      WHILE true
        ALT
          CH[i,j-1] ? 0
            ALT
              B[i,j] ? 0
                CH[i,j] ! 0
              B[i,j] ? 1
                CH[i,j] ! 1
            end ALT
          CH[i,j-1] ? 1
            ALT
              B[i,j] ? 0
                CH[i,j] ! 1
              B[i,j] ? 1
                CH[i,j] ! 0
            end ALT
        end ALT
      end WHILE
  end ALT
end THREAD
end DECLARE
end DECLARE


THREAD InputB IS
  INT i, j, k:
  SEQ k := 1 FOR Q
    PAR i := 1 FOR N
      PAR j := 1 FOR M
        IF a[i,j] = 1
        -- this decision must be inserted!
          if k+1-j <= 0
            B[i,j] ! 0
          else
            B[i,j] ! b[j,k+1-j]
          end if
        ELSE  -- do nothing
        end IF
      end PAR
    end PAR
  end SEQ
end THREAD.
```

Figure 7: Boolean Matrices Specialization

```
THREAD MAIN IS
DO
  x1 < x2  ->  swap(x1, x2);  outstr!"A"
  x2 < x3  ->  swap(x2, x3);  outstr!"B"
  x3 < x4  ->  swap(x3, x4);  outstr!"C"
end DO
outnr!x1
outnr!x2
outnr!x3
outnr!x4
end THREAD

PROCEDURE swap(x,y) IS
aux := x
x := y
y := aux
end PROCEDURE
```

Figure 8: Nondeterministic Sorting

```
THREAD MAIN IS
outstr!"A"          //  outstr!"B"
outstr!"B"          //  outstr!"A"
outstr!"A"          //  outstr!"B"
IF 1 < x4
  x3 := x4
  outstr!"C"
  IF 2 < x3
    x2 := x3
    outstr!"B"
    IF 3 < x2
      x1 := x2
      outstr!"A"
      outnr!x1
      outnr!3
      outnr!2
      outnr!1
    ELSE
      outnr!3
      outnr!x2
      outnr!2
      outnr!1
    end IF
  ELSE
    outnr!3
    outnr!2
    outnr!x3
    outnr!1
  end IF
ELSE
  outnr!3
  outnr!2
  outnr!1
  outnr!x4
end IF
end THREAD
```

Figure 9: Nondeterministic Sorting Specialization