Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-90-25

1990-06-28

A Mathematical Model and Refinement Relation for a CSP-Like Language

Authors: Luming Lai and Jeff Sanders

In this paper we present a mathematical model for CSP-like language. This model handles both safety and liveness properties of "purely parallel" CSP processes, as well as CSP processes with internal machine states. A refinement order is defined in this model which is a combination of the refinement order in CSP's failure model and the refinement order for sequential programs. Finally, related work and applications are discussed.

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research

Recommended Citation

Lai, Luming and Sanders, Jeff, "A Mathematical Model and Refinement Relation for a CSP-Like Language" Report Number: WUCS-90-25 (1990). *All Computer Science and Engineering Research.*
http://openscholarship.wustl.edu/cse_research/700

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

A MATHEMATICAL MODEL AND REFINEMENT RELATION FOR A CSP-LIKE LANGUAGE

Luming Lai and Jeff Sanders

WUCS-90-25

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A Mathematical Model and Refinement Relation for a CSP-like Language

Luming Lai *
Computer Science Department
Washington University
Campus Box 1045
St. Louis, MO 63130

Jeff Sanders
Computing Laboratory
Oxford University
8-11 Keble Rd, Oxford OX1 3QD
England

June 28, 1990

Abstract

In this paper we present a mathematical model for a CSP-like language. This model handles both safety and liveness properties of "purely parallel" CSP processes, as well as CSP processes with internal machine states. A refinement order is defined in this model which is a combination of the refinement order in CSP's failure model and the refinement order for sequential programs. Finally, related work and applications are discussed.

1 Introduction

In the past decade, there has been a substantial increase in the understanding of the nature of sequential and parallel languages. This increase is very much due to the study of the underlying mathematical models.

A number of models have already existed for CSP-like languages [Ros84], [So84] and [Zw88], etc.. None of them is quite satisfactory, as far as unifying the techniques for sequential and parallel languages is concerned. Some of them do not handle liveness properties or nondeterminism [So84] and [Zw88], and some of them do not model sequential programs appropriately [Ros84]. But a most salient feature is that they all support the so-called compositional and text-independent development and verification method. This means that the specification of an abstract parallel system (or program) can be deduced from the specifications of its subsystems independent of their implementation. This gives us the hope to develop a top-down development method for a CSP-like parallel programming language, like the method developed for "purely parallel" CSP in [LS87].

The most successful models for those "purely parallel" languages are Hoare's CSP and Milner's CCS. By "purely parallel", we mean that the machine states of a communicating process are hidden.

The fact that CSP does not facilitate modelling of machine states may not, however, be advantageous when we handle CSP-like languages, like OCCAM.

*Supported in part by a gift from the southwestern Bell Foundation

In the failures model of CSP, the safety properties of a process are specified by its communication traces (tr) which are finite sequences of communications that it can commit on its input and output channels, and liveness properties can be specified by means of failures (tr, ref), where ref called refusal, is a set set of channels on which the process can not send or receive any messages after having performed trace tr. Therefore the observation set of a process in CSP is a set whose elements are of the form (tr, ref), called failures and denoted fr.

To describe a process with machine states, this kind of observation is certainly not enough. In contrast to "purely parallel" CSP, the mathematical model taken in this thesis aims to make machine states explicit.

An observation of a process in our model is a quadruple (s_0, tr, ref, s), or a triple (s_0, fr, s). The interpretation is that a process (or a computation), starting in an initial state s_0, after having performed communications in trace tr, if it does not diverge, e.g. $s \neq \bot$, nor deadlock, i.e. $\surd \notin ref$, then it will terminate in a final state s. A process may be able to do many of this kind of computations. Therefore the observation set of a process is a set of all these computations performed by it.

In order to give a simpler denotational semantics to our CSPL language in this paper, we separate the failure set and the internal machine state transitions, and define a process as a pair

$$(\, Fr,\ Tr\,)$$

where Fr is a total function which takes a machine state and produces a failure set and Tr is a total function which takes a machine state and a trace and produces a set of machine states.

The advantages of the above definition are that the first element $Fr(s_0)$ is, in fact, the failure set of a process in "purely parallel" CSP and has all the properties proved in [BHR84], and that the second element defines the internal machine state transitions and can be used to model sequential programs. Therefore the only thing we have to do is to establish the relationship between these two models.

It is easy to extend our CSPL language to include a so-called specification statement in our model which is of the form

$$Sp \ :: \ I_\alpha,\, \vec{w} \ : \ [\,pre,\ post\,],$$

where I_α specifies the failure set of Sp, pre and $post$ are predicates over trace-state paires, α is the set of channels Sp has and $\vec{w}$ is the set of variables that can be changed in Sp.

In Section 2 we give a formal method of describing processes with communications and define a refinement order on the process domain. The resulting process domain under the refinement order is a complete semi-lattice.

In Section 3 we extend the semantic domain to include specifications. Then we give denotational semantics to mixed terms which forms the mathematical foundation of refinement calculi and proofsystems.

Finally, related models and application areas are discussed.

2 The Description of Processes

The description of the behaviour of a process depends on the observations that can be made on the process while it is being executed.

2.1 Traces

In our model, a process can communicate with other processes or its environment via the channels linking them. Communications are the only way that one process can affect another. No shared program variables are allowed between processes. The atomic unit of communication of a process is called a communication event, denoted $c.v$, where c is a channel name and v is the message communicated on c. A trace tr is a finite sequence of communication events that have occurred during the execution of P.

The set of all finite traces is denoted by $Comm^*$ with $<>$ being the empty trace.

The following notational conventions are also used throughout this thesis:

$tr[i]$	the ith element in trace tr;
$\|tr\|$	is the length of the finite sequence, or trace, tr;
$tr_1 \leq tr_2$	tr_1 is a prefix of tr_2;
$first(tr) = a$	if $tr = < a >^\wedge tr'$;
$last(tr) = a$	if $tr = tr'^\wedge < a >$;
$tr \upharpoonright A$	is the trace obtained from tr by removing all the $c.v$ whose c is not in A. If A is a singleton, $tr \upharpoonright \{c\}$ is abbreviated as c which denotes the sequence of messages passed on channel c

αP is a finite set of channel names that P has. The set of channels is denoted by $Chan$.

Definition 2.1 *(Safety Property)*
The safety properties of a process are what the process is allowed to do, in other words, the process does not do anything wrong.

Examples of safety properties are: a two-place buffer can buffer at most two messages, a variable x is nonnegative, etc. We do not give a formal definition here. The safety properties of the communication behaviour of a process can be specified by means of its traces.

Example 2.1
The safety properties of a one-place buffer with $\alpha P = \{in, out\}$ can be specified by stating that output is a prefix of input which is at most one element shorter:

$$Buff_S \;\widehat{=}\; 0 \leq |\, in \,| - |\, out \,| \leq 1 \;\wedge\; out \leq in. \tag{1}$$

□

2.2 Failures

It is a well-known fact that, under the CSP's refinement order, the liveness properties of the communication behaviour of a process can not be captured solely by its traces.

Definition 2.2 *(Liveness Property)*
The liveness properties of a process are what the process is supposed to do, i.e. something good will eventually happen.

Therefore, in order to specify the liveness properties of a process, another observation element, called refusal and denoted ref, was introduced [BHR84].

Definition 2.3 *(Refusals)*
A refusal of a process P is a subset of $\alpha P_{\surd}$ on which P can refuse to communicate with its environment at the very beginning of its execution, where $A_{\surd} \;\widehat{=}\; A \cup \{\surd\}$ and $\surd$ indicates termination.

Putting tr and ref together, we obtain a new element (tr, ref), called a failure.

Definition 2.4 *(Failures)*
A failure fr of a process is a pair (tr, ref) where tr is a trace of P and ref is a refusal of the process after having performed the communication events in tr.

The meaning of a failure (tr, ref) of a process P is that, after having been engaged in trace tr, P may refuse to take part in any communication on all the channels in ref, even though the environment of P is prepared to do so. If $\surd$ is in ref, P may fail to terminate successfully. If P can not refuse $\{\surd\}$, then P must terminate in a **generalized state** (tr, s), and tr is called a terminating trace of P. If P cannot refuse $\{c\}$ after tr, then the communication event on c must take place after some finite time in spite of internal actions.

The domain of all failures, $Comm^* \times \mathcal{P}(Chan_{\surd})$, is denoted $Failures$.

Example 2.2
The liveness properties of the one-place buffer can be specified by its failures.

$$
\begin{aligned}
Buff_L \;\widehat{=}\; & \quad\; |\, in \,|=|\, out \,| \Longrightarrow in \notin ref \\
& \wedge \;|\, in \,|>|\, out \,| \Longrightarrow out \notin ref,
\end{aligned} \tag{2}
$$

where ref ranges over $\{in,\ out,\ \surd\}$.

(2.2) states that, when the buffer is empty, it is ready and able to take in the next message; and (2.3) states that, when the buffer has already buffered one message in it, it must output it on channel out before it takes another one. In fact, $Buffer_L$ does not say that the buffer can take at most one message. This property is specified in $Buff_S$. Therefore, the complete specification of a one-place buffer is $Buff_S \wedge Buff_L$.
□

2.3 Processes

In order to model divergence and internal machine state transitions, a third observable has to be introduced.

The domain of machine states is defined by

$$State \;:\; Var \rightharpoonup Sv^{+},$$

where *Var* is the domain of variables, *Sv* is the domain of values and $Sv^+ = Sv \cup \{\omega\}$. If a variable is assigned to ω, it is said to be undefined. In order to give a simpler and continuous semantics for the "hiding" operator, we postulate that both *var* and *Sv* are finite.

A state s ($\in State$) maps each variable onto its current value which belongs to *Sv* or an ω.

A process may be engaged in a computation which never reaches a final stable state s because of having been involved in an infinite sequence of internal actions. In this case, the process is said to have been diverging. A special state $\perp_{State}$ (or simply $\perp$) is introduced to indicate such a divergent state. Thus, if $(s_0, tr, ref, \perp)$ is an observation of a process P, then P diverges after *tr* which is called a divergence trace of P from the initial state s_0.

We adopt the conventions that $A \subseteq \perp$ and $A \cup \perp = \perp$ for all $A \subseteq State$.

Then a process P can be defined by a pair

$$(Fr,\ Tr),$$

where *Fr* is a total function which takes a state and produces a failure set, and *Tr* is a total function which takes a state and a trace and produces a set of states.

The interpretation of $(Fr,\ Tr)$ is as follows:

1). The first element *Fr* of a process P determines the failures of process P, started in some initial state s_0. If a set of channels are not a current refusal of P , then the communications on some of these channels must take place if the environment insists in doing so; If $\surd$ is a current refusal of P after *tr*, then, the process *may* fail to terminate successfully after *tr* even though there may be some final states possible for that trace *tr*. Termination *must* take place (if desired by the environment) only when the set $\{\surd\}$ cannot be refused after *tr*.

2). Termination *can* take place on any trace *tr* for which $Tr(s_0, tr)$ is nonempty. When $Tr(s_0, tr)$ contains more than one element, the choice of which final state occurs is nondeterministic.

3). If a $Tr(s_0, tr) = \perp$, then the process is considered to be broken after *tr*, started in the initial state s_0. In this case, we allow the possibility that it might diverge or do anything else.

The trace set of P, started in a state s_0, is denoted $traces(P_{s_0})$, and so is the failure set, $failure(P_{s_0})$.

Definition 2.5 *A process P is a pair*

$$P \ \widehat{=}\ (Fr,\ Tr),$$

where $Fr : State \rightarrow Comm^* \times \mathcal{P}(Chan_{\surd})$ *is a total function and so is* $Tr : State \times Comm^* \rightarrow \mathcal{P}(State) \cup \{\perp\}$. *Fr and Tr satisfy the following conditions, started in an initial state* $s_0 \in State$:

P_1). $traces(P_{s_0})$ *is nonempty and prefix-closed;*

P_2). $(tr, ref) \in Fr(s_0)\ \&\ X \subseteq ref \implies (tr, ref) \in Fr(s_0)$;

P_3). $(tr,\, ref) \in Fr(s_0)$ & $(\forall\, v.\ tr^\wedge < c.v > \notin traces(P_{s_0})) \Longrightarrow$
$(tr,\, ref \cup \{c\}) \in Fr(s_0)$;

S_1). $Tr(s_0,\, tr) \neq \phi \Longrightarrow tr \in traces(P_{s_0})$;

S_2). $(tr,\, ref) \in Fr(s_0)$ & $Tr(s_0,\, tr) = \phi \Longrightarrow (tr,\, ref \cup \{\surd\}) \in Fr(s_0)$;

S_3). $Tr(s_0,\, tr) = \perp$ & $tr' \in Comm^* \Longrightarrow Tr(s_0,\, tr^\wedge tr') = \perp$;

S_4). $Tr(s_0,\, tr) = \perp$ & $X \subseteq Chan_{\surd} \Longrightarrow (tr,\, X) \in Fr(s_0)$.

The space of all processes with alphabet *Chan* and state set *State* is denoted *Proc*.
We define a partial order $\sqsubseteq_r$, called refinement order on *Proc*.

Definition 2.6 *A process $P_1 = (Fr_1,\, Tr_1)$ is said to be refined by another process $P_2 = (Fr_2,\, Tr_2)$, denoted $P_1 \sqsubseteq_r P_2$, if and only if, for any $s_0 \in State$,*

1). $Fr_2(s_0) \subseteq Fr_1(s_0)$;

2). $Tr_2(s_0,\, tr) \subseteq Tr_1(s_0,\, tr)$, *for all* $tr \in Comm^*$

where $\alpha P_1 = \alpha P_2$.

The intuitive meaning of the refinement order $\sqsubseteq_r$ is as follows:
Condition 1) means that the communication behaviour of P_2 is more deterministic than P_1;
Condition 2) means that the final states of P_2 after trace *tr* are included in P_1's.

The least element $\perp_{Proc}$ (or simply $\perp$) on *Proc* under $\sqsubseteq_r$ is defined

$$\perp_{Proc} \ \widehat{=}\ (Fr_\perp,\ Tr_\perp),$$

where $Fr_\perp(s_0) = Failures$ and $Tr_\perp(s_0,\, tr) = \perp_{State}$, for any $s_0 \in State$ and $tr \in Comm^*$.

Then, we have the following theorem.

Theorem 2.1 *$(Proc,\, \sqsubseteq_r,\, \perp_{Proc})$ is a complete semi-lattice.*
□

Proof:
See Appendix.

3 The Denotational Semantics For CSPL

In this section, we give a denotational semantics to the CSP-like parallel language CSPL.

3.1 The Syntax of CSPL

In the following, x stands for a program variable, e for an expression and b for a boolean expression.

Definition 3.1 *(The syntax of CSPL)*
The syntax of CSPL is defined as follows:

$$\begin{array}{rcl} P & \widehat{=} & x := e \mid \textbf{div} \mid \textbf{skip} \mid \textbf{stop} \mid c?x \mid c!e \mid P_1 \parallel_A P_2 \mid \textbf{if } \Box_i\, G_i \textbf{ fi} \mid \\ & & \textbf{while } b \textbf{ do } P \textbf{ end} \mid P_1 ;\ P_2 \mid \textbf{chan } c \textbf{ in } P \textbf{ end} \mid \textbf{var } x \textbf{ in } P \textbf{ end}; \\ G & \widehat{=} & b \& \sigma \rightarrow P; \\ \sigma & \widehat{=} & c?x \mid c!e \mid \textbf{skip}. \end{array}$$

□

The syntactic domain of programs (or processes) is denoted *Prog*.
The intuitive meanings of these program constructs (or processes) are as follows:

- $x := e$ is the usual assignment statement in sequential languages;
- **div** is the most nondeterministic process which pursues an infinite sequence of internal events. These internal events might be either internal communications or nonterminating sequential loops;
- **skip** is a process which terminates successfully with its internal machine states unchanged;
- **stop** is a process which terminates unsuccessfully and no internal states can be observed even though there might exist some. Actually, **stop** describes a process which is deadlocked;
- $c?x$ is an input process which inputs a message on channel c and stores it in a program variable x;
- $c!e$ is an output process which outputs the value of e to its environment through channel c;
- $P_1 \parallel_A P_2$ is a process which runs two processes in parallel. As the only way in which one process can influnce another is via communications along their common channels, P_1 and P_2 do not share any program variables. Therefore, each process has a distinct portion of the state of $P_1 \parallel P_2$ and the state will be reconstructed at the termination of the parallel construct by the "distributed termination" property — a parallel process can only terminate when each of its component processes can terminate (and thus yield its own final state). If any of its component process diverges, the parallel process diverges.
- $P_1 ;\ P_2$ is the sequential composition of P_1 and P_2;
- The if-statemet implement deterministic choice, as well as nondeterministic choice. Whenever there are some boolean guards b_i in b_i&**skip** which are **true** at that moment, the if-statement nondeterministically picks up one of the processes following these guards and excute it. In this case, the guarded processes with input or output in their guards will never be pick up even though some of their boolean guards may be **true**. The purely sequential nondeterministic choice is a special case of the

if-statement when all the guards have **skip** in them and the guarded processes do not have any communications. Olny when all the boolean guards in b_i&**skip** are **false** and some of the boolean guards for input or output guards are **true**, the choice between these input or output guards are deterministic.

- The while-statement have the similar meaning as in sequential languages. It repeatedly excutes P while b is evaluated to **true**. When b is **false**, the while-statement skips.
- **chan c in P end** is a process which has channel c as its internal channel, i.e. all the communications of the form $c.v$ are hidden from the outside. This operator is used to introduce internal channels;
- **var** x **in** P **end** is a process with x being its local variable. Usually we use this operator to introduce local variables.

3.2 The Semantics

In this subsection we give a denotational semantics to our parallel language CSPL.

The main semantic function is

$$\Im : Prog \rightarrow Proc.$$

The intuitive meaning of $\Im$ is that, given a process or a piece of program, $\Im$ produces an element on *Proc*

$$(Fr,\ Tr).$$

We also define

$$f(\Im[\![P]\!]) = Fr_P \qquad \text{and} \qquad t(\Im[\![P]\!]) = Tr_P.$$

For expressions e and boolean expressions b, the value of e and b, evaluated in a state s, is denoted $\mathcal{E}[\![e]\!]s$, where $\mathcal{E} : Exp \rightarrow State \rightarrow Sv \cup Bool \cup \{error\}$, where Exp is the domain of all expressions and $Bool \mathrel{\hat=} \{\mathbf{true},\ \mathbf{false}\}$. If anything goes wrong, like a variable is undefined, etc, the value of $\mathcal{E}$ is *error*. Whenever this happens, the above semantic function produces a divergence process.

1). Primitive Processes

- The divergent process is the worst process. It is identified with the least element $\perp_{Proc}$ on *Proc*.

$$\Im[\![\mathbf{div}]\!] = \perp_{Proc},$$

  where $f(\perp_{Proc}) = Fr_\perp$ and $t(\perp_{Proc}) = Tr_\perp$ as defined in the previous subsection.

- The process **skip** is defined as follows: for any $s_0 \in State$,

$$f(\Im[\![\mathbf{skip}]\!])(s_0) = \{\, (<>,\ X) \mid X \subseteq Chan \,\},$$

  and

$$\begin{aligned} t(\Im[\![\mathbf{skip}]\!])(s_0,\ tr) &= \{s_0\}, \qquad \text{if } tr = <> \\ &= \phi \qquad \text{otherwise} \end{aligned}$$

skip is a process which terminates successfully with its final states unchanged and refuses any communication on any subset of the channels connected to it.

- The process **stop** is slightly different from **skip** in that it terminates unsuccessfully by some faults, such as power breakdown, etc.. Therefore, its final machine states can never be observed or may be lost. Its semantics is defined as follows: for any $s_0 \in State$,

$$f(\Im[\![\mathbf{stop}]\!])(s_0) \;=\; \{\, (<>,\, X) \mid X \subseteq Chan_{\surd} \,\},$$

and

$$t(\Im[\![\mathbf{stop}]\!])(s_0,\, tr) \;=\; \phi, \qquad \text{for all } tr \in Comm^*$$

- The assignment statement is, more or less, the same as that in sequential languages. In CSPL, processes do not share program variables. Consequently, there is no interference between them and the nontrivial interference-free test (see [AFR80]) is not needed in our method. For any $s_0 \in State$, we define

$$f(\Im[\![x \,:=\, e]\!])(s_0) \;=\; \{\, (<>,\, X) \mid X \subseteq Chan \,\},$$

and

$$\begin{aligned} t(\Im[\![x \,:=\, e]\!])(s_0,\, tr) &= \{s_0[\mathcal{E}[\![e]\!]s_0/x]\}, \qquad \text{if } tr = <> \\ &= \phi \qquad \text{otherwise} \\ &\qquad\qquad \text{provided that } \mathcal{E}[\![e]\!]s_0 \neq error. \end{aligned}$$

In the following, whenever $\mathcal{E}[\![e]\!]s \;=\; error$, the semantic function always produces $\perp_{Proc}$.

- The semantics of the output process is relatively simple because it does not change any machine state. We define, for any $s_0 \in State$,

$$\begin{aligned} f(\Im[\![c!e]\!])(s_0) \;=\; & \{\, (<>,\, X) \mid c \notin X \,\} \\ & \bigcup \{\, (< c.v >,\, X) \mid v \;=\; \mathcal{E}[\![e]\!]s_0 \;\&\; X \subseteq Chan \,\}, \end{aligned}$$

$$\begin{aligned} t(\Im[\![c!e]\!])(s_0,\, tr) &= \{s_0\}, \qquad \text{if } tr = < c.v > \text{ and } v \;=\; \mathcal{E}[\![e]\!]s_0 \\ &= \phi \qquad \text{otherwise} \end{aligned}$$

- The most important primitive process is the input process $c?x$. Different explanations can result in different verification and development methods. The semantics defined in the following is natural and supports a modular verification and development method, i.e. the verification and development of the component processes in a parallel process can be carried out independently.

  The definition of the failure set of $c?x$ is the same as that in the "purely parallel" CSP's failures model [BHR84]: for any $s_0 \in State$,

$$\begin{aligned} f(\Im[\![c?x]\!])(s_0) \;=\; & \{\, (<>,\, X) \mid c \notin X \,\} \\ & \bigcup \{(< c.v >,\, X) \mid v \in Sv \;\&\; X \subseteq Chan \,\}. \end{aligned}$$

For the machine state part, we take the explanation of $c?x$ as follows: a process P containing $c?x$ is going to input a message along channel c. There might be some constraints on the message received by P. But the process itself cannot control what kind of message it will actually receive on channel c from the environment. Therefore we have

$$\begin{aligned} t(\Im[\![c?x]\!])(s_0,\, tr) &= \{s_0[v/x]\}, \qquad \text{if } tr = < c.v > \ (v \in Sv) \\ &= \phi \qquad \text{otherwise} \end{aligned}$$

2). **Compositional Processes**

- The first compositional process we are going to define is the sequential composition construct ";". Because the composition of failure sets (not only relations about the machine states) is involved, the semantics of ";" becomes a bit more complex. We have to consider the termination of the first component and the catenation of the traces of the two components.

$$\begin{aligned} & f(\Im[\![P_1\, ;\, P_2]\!])(s_0) \\ = \quad & \{\, (\, tr,\, X\,) \mid (\, tr,\, X \cup \{\surd\}\,) \in f(\Im[\![P_1]\!])(s_0)\,\} \\ & \bigcup \{\, (\, tr_1^\wedge tr_2,\, X\,) \mid \exists\ s'.\ s' \in t(\Im[\![P_1]\!])(s_0,\, tr_1)\ \&\ (tr_2,\, X) \in f(\Im[\![P_2]\!])(s')\,\} \\ & \bigcup \{\, (\, tr_1^\wedge tr_2,\, X\,) \mid t(\Im[\![P_1]\!])(s_0,\, tr_1) = \perp\ \&\ tr_2 \in Comm^*\ \&\ X \in Chan\,\}, \end{aligned}$$

$$\begin{aligned} & t(\Im[\![P_1\, ;\, P_2]\!])(s_0,\, tr) \\ = \quad & \perp, \qquad \text{if } t(\Im[\![P_1]\!])(s_0,\, tr) = \perp, \text{ or if } tr = tr_1^\wedge tr_2, \text{ and there exists } s' \\ & \qquad\quad \text{such that } s' \in t(\Im[\![P_1]\!])(s_0,\, tr_1) \text{ and } t(\Im[\![P_2]\!])(s',\, tr_2) = \perp \\ = \quad & \bigcup \{\, t(\Im[\![P_2]\!])(s',\, tr_2) \mid \exists\, tr_1, tr_2.\ tr = tr_1^\wedge tr_2\ \&\ s' \in t(\Im[\![P_1]\!])(s_0,\, tr_1)\,\} \\ & \qquad\qquad \text{otherwise} \end{aligned}$$

  Note that P_1 cannot refuse a set X unless it can refuse $X \cup \{\surd\}$; otherwise it would be able to terminate (invisibly) and let P_2 take over.

- The difference between deterministic choice and nondeterministic choice lies in the beginning of their execution. In "purely parallel" CSP, the internal machine states are hidden (or omitted). Therefore, for the deterministic choice, its environment is able to make the choice among its guards at the beginning of its execution; but, for the nondeterministic choice, its environment is not able to do so, i.e. the choice is totally uncontrollable. In our CSPL language, the situation is a bit different. A guard (or guarded process) can be "disabled" by setting its boolean part b **false**. Therefore, for both deterministic and nondeterministic choices, the choice is machine-state related. For a deterministic choice, the environment can oly make choice among those guards whose boolean parts are evaluated to **true** in the machine state at that point in time during its execution. For a nondeterministic choice, the choice can only be controlled internally by the machine itself. This difference is reflected in the following definition of our general choice which combines the two kinds of choice constructs.

We first define a boolean function which distinguishes between internal guards whose communication parts are **skip** and external guards whose communication parts are either $c?x$ or $c!e$:

$$
\begin{aligned}
B[\![\, b\&c?x \;\rightarrow\; P \,]\!]s &= \textbf{false}, \\
B[\![\, b\&c!x \;\rightarrow\; P \,]\!]s &= \textbf{false}, \\
B[\![\, b\&\textbf{skip} \;\rightarrow\; P \,]\!]s &= \textbf{true}, \qquad \text{if } \mathcal{E}[\![b]\!]s \text{ is } \textbf{true} \\
&= \textbf{false}, \qquad \text{if } \mathcal{E}[\![b]\!]s \text{ is } \textbf{false} \\
&\qquad\quad \text{provided that } \mathcal{E}[\![b]\!]s \neq \mathit{error}, \\
B[\![\,\textbf{if } \Box_i G_i \textbf{ fi}\,]\!]s &= \bigvee_i B[\![\, G_i \,]\!]s,
\end{aligned}
$$

where $\bigvee$ is the *error*-strict version of the usual boolean $\vee$.

We extend our semantic function to guarded processes and define, for any $s_0 \in \mathit{State}$ and $tr \in \mathit{Comm}^*$,

$$
\begin{aligned}
f(\Im[\![b\&\sigma \;\rightarrow\; P]\!])(s_0) &= \{ (<>,\, X) \mid X \in \mathit{Chan}_{\surd} \}, \qquad \text{if } \mathcal{E}[\![b]\!]s_0 \text{ is } \textbf{false} \\
&= f(\Im[\![\sigma \,;\, P]\!])(s_0), \qquad \text{otherwise}, \\
t(\Im[\![b\&\sigma \;\rightarrow\; P]\!])(s_0, tr) &= \phi, \qquad \text{if } \mathcal{E}[\![b]\!]s_0 \text{ is } \textbf{false} \\
&= t(\Im[\![\sigma \,;\, P]\!])(s_0, tr), \qquad \text{otherwise}, \\
&= \qquad \text{provided that } \mathcal{E}[\![b]\!]s_0 \neq \mathit{error}
\end{aligned}
$$

Then the failure part of **if** $\Box_i G_i$ **fi** can be defined as follows: for any $s_0 \in \mathit{State}$,

$$
\begin{aligned}
& f(\Im[\![\textbf{if } \Box_i G_i \textbf{ fi}]\!])(s_0) \\
= \quad & \{ (<>,\, X) \mid \exists\, i.\; B[\![G_i]\!]s_0 \;\&\; (<>,\, X) \in f(\Im[\![G_i]\!])(s_0) \} \\
& \bigcup \{ (<>,\, X) \mid \forall\, i.\; \neg B[\![G_i]\!]s_0 \;\&\; (<>,\, X) \in f(\Im[\![G_i]\!])(s_0) \} \\
& \bigcup \{ (<>,\, X) \mid \exists\, i. t(\Im[\![G_i]\!])(s_0, <>) \;=\; \perp \} \\
& \bigcup \{ (tr,\, X) \mid tr \neq <> \;\&\; (tr,\, X) \in f(\Im[\![G_i]\!])(s_0) \}.
\end{aligned}
$$

The internal machine state transitions are defined by

$$
t(\Im[\![\textbf{if } \Box_i G_i \textbf{ fi}]\!])(s_0,\; tr) \;=\; \bigcup_i (t(\Im[\![G_i]\!])(s_0,\; tr)), \qquad \text{for any } tr \in \mathit{Comm}^*.
$$

Note that if none of the boolean guards is **true**, then the **if**-statement is equivalent to **stop**.

- To implement iteration, we introduce a **while**-construct. It has the usual meaning as in sequential languages except that it involves communications and may not terminate. The semantics of **while**-construct is similar to that of the sequential composition construct ";" except that it repeats the execution of P until the boolean guard b evaluates to **false**. Therefore we have

$$
\Im[\![\textbf{while } b \textbf{ do } P \textbf{ end}]\!] \;=\; (\bigsqcup_{n=0}^{\infty} H^n(\perp_{Proc})),
$$

where $H : Proc \rightarrow Proc$, is defined, for any $s_0 \in State$ and $tr \in Comm^*$,

$$\begin{aligned}
f(H(A))(s_0) &= f(\Im[\![P]\!]\,;\,A)(s_0), && \text{if } \mathcal{E}[\![b]\!]s_0 = \textbf{true} \\
&= \{(<>, X) \mid X \in Chan\}, && \text{if } \mathcal{E}[\![b]\!]s_0 = \textbf{false}, \\
t(H(A))(s_0, tr) &= t(\Im[\![P]\!]\,;\,A)(s_0, tr), && \text{if } \mathcal{E}[\![b]\!]s_0 = \textbf{true} \\
&= t(\Im[\![\textbf{skip}]\!])(s_0, tr) && \text{if } \mathcal{E}[\![b]\!]s_0 = \textbf{false},
\end{aligned}$$

provided that $\mathcal{E}[\![b]\!]s_0 \neq error$

- Now it is time to define the most important and difficult construct, the parallel composition $\|$. It is the key construct of any parallel languages.

  In our mathematical model, when a process runs in parallel with some other processes, it can only be affected in two ways: one is by communications and the other is by its own internal state changes. The communication with its environment or processes changes both its communication traces and its internal machine states by inputing a message and storing it into a program variable. Therefore, the machine state after an input command depends on the value input from its environment. To ensure that a process terminates in a required final state, we have to know the exact values input on its input channels. This can be done by looking up the communication traces committed by the process during its execution. If the input values satisfy the assumptions made by the process, all the results derived about the process's machine states hold. Otherwise, we can reach nowhere near the correct conclusions about the process's final machine states. We adopt the "distributed termination" properties that the parallel process can terminate only when all of its component processes terminate.

  The failure set of $P_1 \parallel P_2$ is defined as follows: for any $s_0 \in State$,

$$\begin{aligned}
& f(\Im[\![P_1 \parallel P_2]\!])(s_0) \\
= \quad & \{\, (tr,\, X) \mid \exists X_1,\, X_2.\, (tr \upharpoonright \alpha P_1,\, X_1) \in f(\Im[\![P_1]\!])(s_0) \\
& \qquad \&\ (tr \upharpoonright \alpha P_2,\, X_2) \in f(\Im[\![P_2]\!])(s_0)\ \&\ X = X_1 \cup X_2 \,\} \\
& \bigcup \{\, (tr^\wedge tr',\, X) \mid \exists i.\ t(\Im[\![P_i]\!])(s_0,\, tr \upharpoonright \alpha P_i) = \perp_{State} \\
& \qquad \&\ tr \upharpoonright \alpha P_j \in traces(P_{j s_0})\},
\end{aligned}$$

  where $i \neq j$ and $i, j \in \{1, 2\}$.

  The termination component function is defined, for any $s_0 \in State$ and $tr \in Comm^*$,

$$\begin{aligned}
& t(\Im[\![P_1 \parallel P_2]\!])(s_0,\, tr) \\
= \ & \perp_{State} \quad \text{if there exists } tr' \leq tr \text{ such that } tr' \upharpoonright \alpha P_i \in traces(P_{i s_0}) \text{ for all } i, \\
& \qquad \text{and there is some } j \text{ with } t(\Im[\![P_j]\!])(s_0,\, tr' \upharpoonright \alpha P_j) = \perp_{State}, \\
= \ & \{\, s_1 \oplus s_2 \mid s_1 \in t(\Im[\![P_1]\!])(s_0,\, tr \upharpoonright \alpha P_1)\ \&\ s_2 \in t(\Im[\![P_2]\!])(s_0,\, tr \upharpoonright \alpha P_2)\}, \\
& \qquad \text{otherwise}
\end{aligned}$$

  where $\oplus$ is defined as follows:

$$(\, s_1 \oplus s_2\,)(x)$$

$$\begin{aligned} &= s_1(x) \qquad \text{whenever } s_1(x) \neq \omega \\ &= s_2(x) \qquad \text{whenever } s_2(x) \neq \omega \\ &= \omega \qquad \text{otherwise} \end{aligned}$$

If s_1 and s_2 map x into different non-ω values, the parallel combination is broken from the point of this error. The disjointness constraints of $\|$ guarantee that this error cannot arise.

The parallel composition construct works by allowing each process to communicate only in its own alphabet, and only allowing a given communication occur when each process whose alphabet it belongs to agrees. Termination can only take place when all processes agree, the final state being formed by joining together the states of the individual processes. As soon as one process breaks, the while system is considered broken.

- To obtain good abstraction in CSPL, we introduce a hiding operator **chan** c **in** P **end**, which conceals the channel "c" in P. Its semantics is as follows:

$$\begin{aligned} & f(\Im[\![\textbf{chan } c \textbf{ in } P \textbf{ end}]\!])(s_0) \\ = \quad & \{ \, (tr \backslash c, \, X) \mid (tr, \, X \cup \{c\}) \in f(\Im[\![P]\!])(s_0) \, \} \\ & \bigcup \{ \, (tr, \, X) \mid \{ \, tr' \mid tr' \in traces(P_{s_0}) \,\&\, tr' \backslash c \leq tr \, \} \text{ is infinite} \, \}, \end{aligned}$$

where $tr \backslash c \; = \; tr \restriction (\alpha P \; - \; \{c\})$.

$$\begin{aligned} & t(\Im[\![\textbf{chan } c \textbf{ in } P \textbf{ end}]\!])(s_0, \, tr) \\ = \quad & \bot_{State} \qquad \text{if } \{ \, tr' \mid tr' \in traces(P_{s_0}) \,\&\, tr' \backslash c \leq tr \, \} \text{ is infinite} \\ = \quad & \bigcup \{ \, t(\Im[\![P]\!])(s_0, \, tr') \mid tr' \backslash c \; = \; tr \, \}. \end{aligned}$$

- The operator for introducing local variables is much simpler because it has nothing to do with a process's failure set. Its semantics is defined as

$$f(\Im[\![\textbf{var } x \textbf{ in } P \textbf{ end}]\!])(s_0, \, tr) \; = \; f(\Im[\![P]\!])(s_0),$$

and

$$\begin{aligned} & t(\Im[\![\textbf{var } x \textbf{ in } P \textbf{ end}]\!])(s_0, \, tr) \\ = \quad & \{ s \backslash x \mid s \in t(\Im[\![P]\!])(s_0, \, tr) \, \}, \qquad \text{for all } tr \in Comm^* \end{aligned}$$

where $s \backslash x$ is defined as follows:

$$\begin{aligned} (s \backslash x)(y) \; &= \; s(y), \qquad \text{if } y \not\equiv x \\ &= \; \omega, \qquad \text{otherwise} \end{aligned}$$

This completes the definitions of the semantics of all the constructs in CSPL.

Theorem 3.1 *All the constructs defined above are well-defined and continuous.*

Proof: See [lai90].

4 Applications and related work

The mathematical model and refinement order developed in the previous Section can be used as the mathematical foundation for developing proofsystems and refinement calculi for CSP-like languages. In the following, we give an example of how to extend the above model to mixed terms (programming language with specifications) which can be used to develop proof rules and refinement laws in a unified framework.

4.1 Mixed terms

First, we introduce our specification formulas.

The specification of a program is a description of the way it is intended to behave. The specification is normally written in some formal mathematical language. Unlike program code, specifications generally are not executable. A specification can be regarded as an abstract program. On a more practical level, however, a programmer starts with a specification which gives the idea of a program's role in the system. Insight into the structure of the program points to a series of refinement of the original specification until an executable specification or a program is obtained.

A specification of a communicating sequntial process describes not only the communication behaviour, but also the relationship between communications, initial values of the program variables, and their final values. To denote the initial value of a program variable x, we decorate the variable name with a subscription 0, as in x_0. To denote the final value of x, we simply use the variable name x by itself. The value x is not observable untill the process has terminated. Because the internal states of a communicating sequential process is related to its traces, we have the following predicates to descibe the initial states, final states, and the communication behaviour of a process.

$$\begin{aligned} pre &: Comm^* \times State \rightarrow Bool \\ post &: Comm^* \times State \times State \rightarrow Bool \\ I_\alpha &: State \times Failures \rightarrow Bool \end{aligned} \tag{3}$$

> Note that the pre- and pos-tcoditions in our specifications are no longer assertions on machine states, but rather assertions on the combination of communication traces and machine states, $(tr,\ s)$, which are called **generalized states** in [Zw88].

Thus the pre-condition specifies over what subset of the possible generalized states the program may work, the post-condition specifies the terminating traces and the required relationships between initial and final states, and the communication invariant specifies the communication behaviour of the program.

Sometimes we distinguish between the safety and liveness properties of the communication behaviour of a process by I_α^S and I_α^L, respectively. Then, we have

$$I_\alpha = I_\alpha^S \wedge I_\alpha^L.$$

We postulate that traces in the expressions can only be related by the channel projection operator $\upharpoonright$ as $tr \upharpoonright A$.

The specification formulae, or specification statements, as called in [Mor88], is defined as follows.

Definition 4.1 *(Specification statement)*
A specification statement Sp is a quadruple

$$I_\alpha,\ \vec{w}\ :\ [\ pre,\ post\ ],$$

where $\vec{w}$ *is a subset of variables* $\vec{v}$ *Sp has, which can be changed in Sp,* α *is the set of channels of Sp, and* I_α*, pre and post are predicates of the types as defined in (3).*

The domain of all specification statements is denoted by *Spec*.

A specification statement *Sp* can be understood as a generalized state transformer. Its intuitive meaning is as follows:

> A computation *Cp* of *Sp* starts in one of the initial generalized states $(tr_0,\ s_0)$ described by *pre*, with tr_0 and s_0 satisfying I_α; then, at any moment during its execution, I_α always holds for the communication behaviour of *Cp*; if *Cp* terminates, *Cp* will terminate in one of the final generalized states $(tr,\ s)$ described by *post*, with the terminating trace *tr* satisfying I_α. If there are more than one possible final states described by the postcondition, any one of them may be chosen nondeterministically. If *Cp* starts in an initial state which is not described by the precondition, then everything could happen, including divergence.

We give some examples of the specification statements of some concurrent systems.

Example 4.1
A one-place buffer B_1, with "*in*" being its input channel and "*out*" its output channel, can be specified by means of our specification statement as follows:

$$I_\alpha,\ x\ :\ [\mathit{in}\ =\ \mathit{out}\ = <>,\ \mathbf{false}],$$

where $\alpha\ =\ \{\mathit{in},\ \mathit{out}\}$, and

$$\begin{array}{rcl} I_\alpha^S & \hat{=} & 0 \le |\mathit{in}|\ -\ |\mathit{out}| \le 1\ \wedge\ \mathit{out} \le \mathit{in} \\ I_\alpha^L & \hat{=} & |\mathit{out}|\ <\ |\mathit{in}|\ \Longrightarrow\ \mathit{out} \notin \mathit{ref} \\ & & \wedge\ |\mathit{out}|\ =\ |\mathit{in}|\ \Longrightarrow\ \mathit{in} \notin \mathit{ref}. \end{array}$$

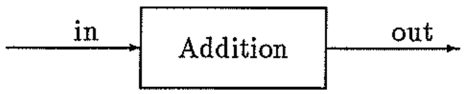
where $\mathit{ref} \subseteq \alpha_{\surd}$. This usually omit in the following when it is obvious.

The postcondition **false** merely indicates that B_1 will never terminate, i.e. the final stable internal state **false** can never be reached. I_α^S states that the one-place buffer can keep at most one message and the sequence of the input messages is not changed in order and content while being output. I_α^L states that, when the buffer is empty, B_1 must input a message before it outputs any. When the buffer has one message in it, it must output it before it take another.
□

The following example is a bit more complicated than the above one. It involves communications as well as internal states.

Example 4.2

A process B_1' is a one-place buffer which inputs ten integers and adds up the integers in even-numbered positions in the input trace *in* and the integers in odd-numbered positions.



$$B_1' \ \widehat{=}\ I_\alpha^S \wedge I_\alpha^L,\ x, y, z\ :\ [\, preB_1',\ postB_1' \,]$$

where

$$\begin{array}{lcl}
I_\alpha^S & \widehat{=} & 0 \leq |in| - |out| \leq 1 \wedge out \leq in \\
I_\alpha^L & \widehat{=} & |in| < 10 \vee |out| < 10 \Longrightarrow (\,|in| = |out| \Longrightarrow in \notin ref \\
 & & \qquad\qquad\qquad\qquad\qquad \wedge\, |in| > |out| \Longrightarrow out \notin ref\,) \\
 & & \wedge\, |in| = |out| = 10 \Longrightarrow ref \subseteq \{\, in,\ out \,\} \\
preB_1' & \widehat{=} & in = out = <> \\
postB_1' & \widehat{=} & |in| = |out| = 10 \wedge x = \sum_{i=1}^{5} in[2i] \wedge y = \sum_{i=1}^{5} in[2i-1].
\end{array}$$

I_α^S and I_α^L state that B_1' behaves like a one-place buffer and must terminate after having received and output ten messages.

preB$_1'$ describes the initial states in which the process can start and *postB*$_1'$ states that, when B_1' terminates, the sums of integers in the even- and odd-numbered positions are stored in x and y, respectively.
□

4.2 Semantics of Program Constructs

In order to give a formal definition of the specification statement, we expand the process (or program) domain *Prog* to mixed terms *Prog*$'$, which includes specification statements.

$$\begin{array}{lcl}
P & ::= & I_\alpha,\ \vec{w}\ :\ [pre,\ post] \mid x := e \mid \mathbf{div} \mid \mathbf{skip} \mid \mathbf{stop} \mid c?x \mid c!e \mid \\
 & & P_1 \parallel_A P_2 \mid \mathbf{if}\ \Box_i\ G_i\ \mathbf{fi} \mid \mathbf{while}\ b\ \mathbf{do}\ P\ \mathbf{end} \mid P_1\ ;\ P_2 \mid \\
 & & \mathbf{chan}\ c\ \mathbf{in}\ P\ \mathbf{end} \mid \mathbf{var}\ x\ \mathbf{in}\ P\ \mathbf{end}; \\
G & \widehat{=} & b\,\&\sigma \rightarrow P; \\
\sigma & \widehat{=} & c?x \mid c!e \mid \mathbf{skip}.
\end{array}$$

As we regard a communicating sequential process as a generalized state transformer, the state domain *State* in the previous Section has to be extended to the domian of generalized states, denoted *Gstate* ($\widehat{=}$ $Comm^* \times State$). Then, a mixed term or program can also be modeled by a pair $(Fr',\ Tr')$, where Fr' and Tr' are the same as those defined before, except that the *State* is replaced by the *Gstate*. As a specification statement may not be implementable, the element $(Fr',\ Tr')$ of the domain *Proc*$'$ does not have to satisfy the conditions in the definition of a process in the previous section.

We define f' and t' as

$$f'(P) \;=\; Fr'_P \qquad \text{and} \qquad t'(P) \;=\; Tr'_P$$

We define a total function ft : $Failures \rightarrow Failures$

$$\begin{aligned} ft(A) \;=\; & \{ (tr,\, X) \mid (tr,\, X \cup \{\surd\}) \in A \} \\ & \bigcup \{ (tr,\, \phi) \mid (tr,\, \phi) \in A \,\&\, (tr,\, \{\surd\}) \notin A \} \end{aligned}$$

ft is used to make the refusal of a terminating trace empty. This is important in the definition of the refinement order on $Prog'$.

The refinement order on $Proc'$ is modified accordingly as follows.

Definition 4.2 *(Refinement Order)*
A mixed term P_1 is said to be refined by another P_2, denoted

$$P_1 \sqsubseteq_M P_2,$$

if and only if, for all $(tr_0,\, s_0) \in Gstate$ and $tr \in Comm^*$

1) $ft(f'(P_2)(tr_0,\, s_0))) \subseteq ft(f'(P_1)(tr_0,\, s_0)));$

2) $t'(P_2)(tr_0,\, s_0)(tr) \subseteq t'(P_1)(tr_0,\, s_0)(tr).$

The least element $\bot_{Proc'}$ on $Proc'$ under $\sqsubseteq_M$ is defined

$$\bot_{Proc'} \;\widehat{=}\; (Fr'_\bot,\, Tr'_\bot)$$

where $Fr'_\bot(tr_0,\, s_0) \;=\; Failures$ and $Tr'_\bot(tr_0,\, s_0)(tr) \;=\; \bot_{State}$.

Theorem 4.1 $(Proc',\, \sqsubseteq_M,\, \bot_{Proc'})$ *is a complete lattice.*

Proof:
The proof is similar to that of Theorem ??.

4.3 Semantics

In the following, we give a denotational semantics to the mixed terms.

The main semantic function is

$$\mathcal{M} \;:\; Prog' \rightarrow Proc'$$

The intuitive meaning of M is: given a piece of mixed term, it produces an element on $Proc'$.

1. **Semantics of The Specification Statement**
   The semantics of a specification statement is defined as follow:

$$\mathcal{M}[\![I_\alpha,\, \vec{w} \;:\; [\, pre,\, post \,]]\!] \;\widehat{=}\; SP, \qquad \text{where}$$

$$\begin{array}{rcll} f'(SP)(tr_0,\, s_0) & = & \{\,(tr,\, X) \mid I_\alpha(s_0,\, tr_0^\wedge tr,\, X)\,\} & \text{if } pre(tr_0,\, s_0) \\ & = & \{\,(tr,\, X) \mid X \subseteq Chan_{\surd} \,\wedge\, tr \in Comm^*\,\} & \text{otherwise} \\ t'(SP)(tr_0,\, s_0)(tr) & = & \{\, s \mid post(tr_0^\wedge tr,\, s_0,\, s) \,\wedge\, I_\alpha(s_0,\, tr_0^\wedge tr,\, \phi)\,\} & \text{if } pre(tr_0,\, s_0) \\ & = & \perp_{State} & \text{otherwise} \end{array}$$

We give some examples to explain the meaning of a specification statement.
Example 4.3

$$\mathbf{true}_\alpha,\ \vec{w}\ :\ [\,\mathbf{false},\ \mathbf{true}\,].$$

The intuitive meaning is : the above process cannot starting in any initial generalized state; it is considered to be diverging at the very beginning. It is the weakest specification statement which can be refined by any thing, as indicated by its semantics,

$$\mathcal{M}[\![\mathbf{true}_\alpha,\ \vec{w}\ :\ [\,\mathbf{false},\ \mathbf{true}\,]]\!]\ =\ P_{div}$$

where, for any $(tr_0,\, s_0) \in Gstate$,

$$\begin{array}{rcll} f'(P_{div})(tr_0,\, s_0) & = & \{\,(\,tr,\, X\,) \mid X \subseteq \alpha_{\surd} \,\wedge\, tr \in Comm^*\} & \\ t'(P_{div})(tr_0,\, s_0)(tr) & = & \perp_{State}, & \text{for all } tr \in Comm^* \end{array}$$

We call it **Miracle** as in [Mor88]. **div** is its counterpart on process domain *Proc*.
□
Example 4.4

$$\mathbf{false}_\alpha,\ \vec{w}\ :\ [\,\mathbf{true},\ \mathbf{false}\,].$$

Its semantics is defined as follows:

$$\mathcal{M}[\![\mathbf{false}_\alpha,\ \vec{w}\ :\ [\,\mathbf{true},\ \mathbf{false}\,]]\!]\ =\ P_{mac}$$

where

$$\begin{array}{rcll} f'(P_{mac})(tr_0,\, s_0) & = & \phi & \\ t'(P_{mac})(tr_0,\, s_0)(tr) & = & \phi, & \text{for all } tr \in Comm^* \end{array}$$

This is the strongest specification statement which refines any other specification statements. But, unfortunately, it itself is not implementable. It is similar to the "**Magic**" in [Mor88].
□

- **Semantics of Program Constructs**
- Divergence

$$\mathcal{M}[\![\mathbf{div}]\!]\ =\ \perp_{Proc'},$$

where $\perp_{Proc'}$ is the bottom element on *Proc'*.

- Termination

$$f'(\mathcal{M}[\![\mathbf{skip}]\!])(tr_0,\, s_0) \;=\; f(\Im[\![\mathbf{skip}]\!])(s_0).$$
$$t'(\mathcal{M}[\![\mathbf{skip}]\!])(tr_0,\, s_0)(tr) \;=\; t(\Im[\![\mathbf{skip}]\!])(s_0, tr).$$

- Deadlock

$$f'(\mathcal{M}[\![\mathbf{stop}]\!])(tr_0,\, s_0) \;=\; f(\Im[\![\mathbf{stop}]\!])(s_0).$$
$$t'(\mathcal{M}[\![\mathbf{stop}]\!])(tr_0,\, s_0)(tr) \;=\; t(\Im[\![\mathbf{stop}]\!])(s_0, tr).$$

- Assignment

$$f'(\mathcal{M}[\![x := e]\!])(tr_0,\, s_0) \;=\; f(\Im[\![x := e]\!])(s_0).$$
$$t'(\mathcal{M}[\![x := e]\!])(tr_0,\, s_0)(tr) \;=\; t(\Im[\![x := e]\!])(s_0, tr).$$

- Output

$$f'(\mathcal{M}[\![c!e]\!])(tr_0,\, s_0) \;=\; f(\Im[\![c!e]\!])(s_0).$$
$$t'(\mathcal{M}[\![c!e]\!])(tr_0,\, s_0)(tr) \;=\; t(\Im[\![c!e]\!])(s_0, tr).$$

- Input

$$f'(\mathcal{M}[\![c?x]\!])(tr_0,\, s_0) \;=\; f(\Im[\![c?x]\!])(s_0).$$
$$t'(\mathcal{M}[\![c?x]\!])(tr_0,\, s_0)(tr) \;=\; t(\Im[\![c?x]\!])(s_0, tr).$$

- Sequential Composition

$$\begin{aligned}
& f'(\mathcal{M}[\![P_1 \,;\; P_2]\!])(tr_0,\, s_0) \\
= \quad & \{\, (\, tr,\; X\,) \mid (\, tr,\; X \cup \{\surd\}\,) \in f'(\mathcal{M}[\![P_1]\!])(tr_0,\, s_0)\,\} \\
& \bigcup \{\, (\, tr_1^\frown tr_2,\; X\,) \mid t'(\mathcal{M}[\![P_1]\!])(tr_0,\, s_0)(tr_1) \;=\; \bot \;\&\; tr_2 \in \mathit{Comm}^* \;\wedge\; X \in \mathit{Chan}\,\} \\
& \bigcup \{\, (\, tr_1^\frown tr_2,\; X\,) \mid \exists\; s'.\; s' \in t'(\mathcal{M}[\![P_1]\!])(tr_0,\, s_0)(tr_1) \\
& \qquad\qquad\qquad\qquad \&\; (tr_2,\; X) \in f'(\mathcal{M}[\![P_2]\!])(tr_1,\, s')\,\}.
\end{aligned}$$

$$\begin{aligned}
& f'(\mathcal{M}[\![P_1 \,;\; P_2]\!])(tr_0,\, s_0)(tr) \\
= \quad & \bot, \quad \text{if } f'(\mathcal{M}[\![P_1]\!])(tr_0,\, s_0)(tr) \;=\; \bot, \text{ or if } tr \;=\; tr_1^\frown tr_2, \\
& \qquad \text{where } s' \in t'(\mathcal{M}[\![P_1]\!])(tr_0,\, s_0)(tr_1) \text{ and } t'(\mathcal{M}[\![P_2]\!])(tr_1,\, s')(tr_2) \;=\; \bot \\
= \quad & \bigcup \{\; t'(\mathcal{M}[\![P_2]\!])(tr_1,\, s')(tr) \mid \exists\, tr_1, tr_2, s'.\; tr \;=\; tr_1^\frown tr_2 \\
& \qquad\qquad \&\; s' \in t'(\mathcal{M}[\![P_1]\!])(tr_1,\, s_0)(tr_1)\,\} \\
& \qquad \text{otherwise}
\end{aligned}$$

- Iteration

$$\mathcal{M}[\![\mathbf{while}\; b\; \mathbf{do}\; P\; \mathbf{end}]\!] \;=\; (\; \bigsqcup_{n=0}^{\infty} H_M^n(\,\bot_{Proc}\,)\,)$$

where $H_M : Proc' \to Proc'$, is defined, for any $s_0 \in State$, tr_0 and $tr \in Comm^*$,

$$\begin{array}{rcll}
f(H_M(A))(tr_0, s_0) & = & f'(\Im[\![P]\!]\,;\,A)(tr_0, s_0), & \text{if } \mathcal{E}[\![b]\!]s_0 = \mathbf{true} \\
 & = & \{(<>, X) \mid X \in Chan\}, & \text{if } \mathcal{E}[\![b]\!]s_0 = \mathbf{false}, \\
t(H_M(A))(tr_0, s_0)(tr) & = & t'(\Im[\![P]\!]\,;\,A)(tr_0, s_0)(tr), & \text{if } \mathcal{E}[\![b]\!]s_0 = \mathbf{true} \\
 & = & t'(\Im[\![\mathbf{skip}]\!])(tr_0, s_0)(tr) & \text{if } \mathcal{E}[\![b]\!]s_0 = \mathbf{false}, \\
 & & \text{provided that } \mathcal{E}[\![b]\!]s_0 \neq error &
\end{array}$$

- Parallelism

$$\begin{array}{rl}
 & \mathcal{M}[\![P_1 \parallel P_2]\!](tr_0, s_0) \\
= & (\mathcal{M}[\![P_1]\!](tr_0 \upharpoonright \alpha P_1), s_0) \parallel_{Prog} (\mathcal{M}[\![P_2]\!](tr_0 \upharpoonright \alpha P_2), s_0)
\end{array}$$

where $\parallel_{Prog}$ is the same as that defined before.

- Choice

$$\begin{array}{rl}
 & f'(\mathcal{M}[\![\mathbf{if}\Box_i G_i \mathbf{fi}]\!])(tr_0, s_0) \\
= & \{(<>, X) \mid \exists\, i.\ B[\![G_i]\!]s_0\ \&\ (<>, X) \in f'(\mathcal{M}[\![G_i]\!])(tr_0, s_0)\} \\
 & \bigcup\{(<>, X) \mid \forall\, i.\ \neg B[\![G_i]\!]s_0\ \&\ (<>, X) \in f'(\mathcal{M}[\![G_i]\!])(tr_0, s_0)\} \\
 & \bigcup\{(<>, X) \mid \exists\, i.\ t'(\mathcal{M}[\![G_i]\!])(tr_0, s_0)(<>) = \bot_{State}\} \\
 & \bigcup\{(tr, X) \mid tr \neq <>\ \&\ (tr, X) \in f'(\mathcal{M}[\![G_i]\!])(tr_0, s_0)\}.
\end{array}$$

$$t(\mathcal{M}[\![\mathbf{if}\Box_i G_i \mathbf{fi}]\!])(tr_0, s_0)(tr) = \bigcup_i (t'\mathcal{M}[\![G_i]\!])(tr_0, s_0)(tr))$$

- Channel Hiding

$$\begin{array}{rl}
 & f'(\mathcal{M}[\![\mathbf{chan}\, c\, \mathbf{in}\, P\, \mathbf{end}]\!])(tr_0, s_0) \\
= & \{(tr \backslash c, X) \mid (tr, X \cup \{c\}) \in f'(\mathcal{M}[\![P]\!])(tr_0, s_0)\} \\
 & \bigcup\{(tr, X) \mid \{tr' \mid tr' \in traces(P_{(tr0,s0)})\ \&\ tr' \backslash c \leq tr\} \text{ is infinite}\}
\end{array}$$

$$\begin{array}{rl}
 & f'(\mathcal{M}[\![\mathbf{chan}\, c\, \mathbf{in}\, P\, \mathbf{end}]\!])(tr_0, s_0)(tr) \\
= & \bot_{State}, \quad \text{if } \{tr' \mid tr' \in traces(P_{(tr0,s0)})\ \&\ tr' \backslash c \leq tr\} \text{ is infinite} \\
= & \bigcup\{t'(\mathcal{M}'[\![P]\!])(tr_0, s_0)(tr') \mid tr' \backslash c = tr\}
\end{array}$$

- Local Variable

$$f'(\mathcal{M}[\![\mathbf{var}\ x\ \mathbf{in}\ P\ \mathbf{end}]\!])(tr_0, s_0) = f'(\mathcal{M}[\![P]\!])(tr_0, s_0),$$

and

$$\begin{array}{rl}
 & t'(\mathcal{M}[\![\mathbf{var}\ x\ \mathbf{in}\ P\ \mathbf{end}]\!])(tr_0, s_0)(tr) \\
= & \{s \backslash x \mid s \in t'(\mathcal{M}[\![P]\!])(tr_0, s_0)(tr)\}, \qquad \text{for all } tr \in Comm^*
\end{array}$$

This completes the definitions of the semantics for the mixed terms.

4.4 Related Work

- **The Features of Our Mathematical Model**
  The mathematical model developed in this paper is based on the divergence model for the purely parallel version of CSP [Ho85].

  However, since the divergence model has omitted several central sequential constructs, like assignment, it can not model the transitions of internal machine states appropriately. The framework chosen in this paper for dealing with this problem is the well-known idea of regarding a program as a state transformer between initial and final states [HH85]. We add to the divergence model the initial states and termination components, i.e. the final states after its termination. Doing so, our model is capable of modelling both the communication behaviour of a program, as well as its internal state transitions. The purpose of this is to obtain a unified formalism to combine the techniques developed for purely parallel processes and sequential programs.

  A closely related model is that of A. W. Roscoe's for Occam [Ros84]. The main difference between his and ours is that: although the internal states is modelled in his model by functions from traces to states, the initial states of a process are assumed to be arbitrary. In our model, we assume that a program can start in only certain set of initial states like a sequential program does. The result of this is thatsequential programs can be properly modelled. It is also easy to extend our model to include specificaiton statements and give a denotational semantics to the specification statement and mixed terms. In this end, we can handle specifications and programs in a unified framework as Carroll Morgan did in [Mor88].

- **The Features of Our Specification Formulas**
  The correctness formula in our formalism is inspired by C. Morgan's "specification statement calculus"[Mor88].

  In [Zw88], a similar specification formula was developed. As the communication invariant I does not contain any program varibles, his specification languages cannot spefication internal-state related communication conditionals precisely, such as **if** $x = 1 \rightarrow \ a?x \ \Box x = 2 \rightarrow \ b?x$ **fi**. We will discuss this problem in a forthcoming paper.

  Our specification statements can also specify certain liveness properties of a communicating process, such as nondeterminism and deadlocks, etc.. However, as CSP is not suitable for specifying fairness properties, nor is ours.

5 Conclusion

In this paper, we present a formalism for modeling and specifying communicating sequential processes with internal states.

Our model and specification statements can handle both communicating processes as well as purely sequential programs. The model is a hybrid model. It has the advantages in that, on purely parallel programs, the results obtained would correspond closely to the old failure-sets model, and that, on purely sequential programs, the results would be relations on states.

We will develop a refinement calculus based on this model for CSP-like parallel languages in a forthcoming paper. Since there is not only one way to Rome, we would like to investigate some other kind of specification methods, like temporal logic. As the way of specifying a system is very vital to the application of the development method, this research direction is of great interest.

We also need to do more examples to explore the refinement laws which would help to reduce the development efforts.

Another area of application of our method might be the design of VLSI.

6 Acknowlagement

The first author would like to thank Wei Chen and Jifeng He for many fruitful discussions.

References

[AFR80] K.R. Apt, N. Francez and W.P. de Roever. *A Proofsystem for Communicating Sequential Processes.* Toplas 2 (1980).

[BHR84] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe. *A Theory of Communicating Sequential Processes.* J. Assoc. Comput. March. 31, 560-599. 1984.

[BR84] S.D. Brookes and A. W. Roscoe,*An Improved Failures Model for Communicating Processes.* LNCS 197, 1984.

[HH85] C.A.R. Hoare and Jifeng He. *The Weakest Prespecification.* Technical Monograph PRG-44. Computing Lab. PRG, Oxford University. 1985.

[Ho84] C.A.R. Hoare. *Programs Are Predicates.*In *Mathematical Logic and Programming Languages*, 141-155. Prentice Hall. 1984.

[Ho85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London. 1985.

[Lai87] L.M. Lai. *The Decomposition of Concurrent Systems.* D.Phil. transfer dissertation.Computing Lab. PRG, Oxford University. 1987.

[Mor88] Carroll Morgan. *Programming from Specifications.* Book draft. Computing Lab. PRG, Oxford University. 1988.

[Ros84] A. W. Roscoe,*Denotational Semantics for OCCAM.* LNCS 197, 1984.

[So87] N.Soundararajan. *Liveness of CSP programs.* Technical Research Report, OSU-CISRC-9/87-TR38. Computer and Information Science Research Centre, The Ohio State University. 1987.

[ZRE85] J. Zwiers, W.P. de Roever and P. Van Emde Boas. *Compositionality and Concurrent Networks: Soundness and Completeness of a Proofsystem.* Proc. of ICALP'85, LNCS 194, 1985.

[Zw88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness: Proof theories for networks of processes, and their connection.* Ph.D. thesis, University of Nijmegen, 1988.

A Mathematical Properties of Process Space Proc

To give a denotational semantics to our CSPL, we need the following mathematical concepts and properties on the process domain.

Let $\mathcal{D}$ be a set with a partial order $\sqsubseteq$.

Definition A.1 *Consider a partially ordered set $(\mathcal{D}, \sqsubseteq)$. a is a lower bound of a set $X \subseteq \mathcal{D}$, and b is an upper bound, provided that $a \subseteq x$ for all $x \in X$, and $x \subseteq b$ for all $x \in X$, respectively. If the set of upper bound has a unique smallest element, we call it the least upper bound and write it as $\sqcup X$ or sup X. Similarly the greatest lower bound is written as $\sqcap X$ or inf X.*

Definition A.2 *A semilattice is a partially ordered set $(\mathcal{D}, \sqsubseteq)$ in which every nonempty finite subset has an inf. A sup-lattice is a partially ordered set in which every finite subset has a sup. $(\mathcal{D}, \sqsubseteq)$ is a called a lattice if it is both a semilattice and a sup-lattice.*
□

Definition A.3 *A lattice is called a complete lattice if every subset of it has an inf.*

Definition A.4 *An operator $f : \mathcal{D} \to \xi$ from one cpo $\mathcal{D}$ to another ξ is called strict if it preserves the least element, monotonic if it preserves the partial order, and continuous if it preserves the least upper bound, i.e. if $x_1 \sqsubseteq x_2 \sqsubseteq \cdots$ is an ascending chain in $\mathcal{D}$, then*

$$f(\bigsqcup_i x_i) = \bigsqcup_i f(x_i).$$

□

By Knaster-Tarski's fixed point theorem [Tar49], every monotonic operator $f : \mathcal{D} \to \mathcal{D}$ has a least fixed point $\mathit{fix}(f)$ in $\mathcal{D}$. If f is continuous, $\mathit{fix}(f)$ can be represented as

$$\mathit{fix}(f) = \bigsqcup_{n \geq 0} f^n(\bot),$$

where $f^0(d) = d$ and $f^{n+1}(d) = f^n(f(d))$ for all $d \in \mathcal{D}$.

We define a partial order $\sqsubseteq_r$ on *Proc*.

Definition A.5 *A process $P_1 = (Fr_1, Tr_1)$ is said to be refined by another process $P_2 = (Fr_2, Tr_2)$, denoted $P_1 \sqsubseteq_r P_2$, if and only if, for any $s_0 \in State$,*

1). $Fr_2(s_0) \subseteq Fr_1(s_0)$;

2). $Tr_2(s_0, tr) \subseteq Tr_1(s_0, tr)$,   for all $tr \in Comm^*$

where $\alpha P_1 = \alpha P_2$.

The least element $\perp_{Proc}$ (or simply $\perp$) on *Proc* under $\sqsubseteq_r$ is defined

$$\perp_{Proc} \;\widehat{=}\; (Fr_\perp,\; Tr_\perp),$$

where $Fr_\perp(s_0) = (Comm^* \times \mathcal{P}(Chan_{\surd}))$ and $Tr_\perp(s_0, tr) = \perp_{State}$, for any $s_0 \in State$ and $tr \in Comm^*$.

To prove that $(Proc, \sqsubseteq_r, \perp_{Proc})$ is a complete semilattice, we define two binary operators, the union operator $\bigsqcup$ and the intersection operator $\prod$ on *Proc*.

Definition A.6 *Let $P_1 = (Fr_1, Tr_1)$ and $P_2 = (Fr_2, Tr_2)$ be two processes on Proc with $\alpha P_1 = \alpha P_2$. Then, we define*

$$P_1 \bigsqcup P_2 \;\widehat{=}\; (Fr_1 \cap Fr_2,\; Tr_1 \cap Tr_2)$$
$$P_1 \prod P_2 \;\widehat{=}\; (Fr_1 \cup Fr_2,\; Tr_1 \cup Tr_2)$$

where $\cap$ and $\cup$ are set intersection and union, respectively, $(f_1 \cup f_2)(x) = f_1(x) \cup f_2(x)$ and $(f_1 \cap f_2)(x) = f_1(x) \cap f_2(x)$. □

Then we have the following lemmas.

Lemma A.1 *The intersection, $P_1 \prod P_2$, of two processes P_1 and P_2 is a process.*
□

Proof:

Let $P_1 = (Fr_1, Tr_1)$ and $P_2 = (Fr_2, Tr_2)$ be two processes and $P = (Fr, Tr) = P_1 \prod P_2$, so that $Fr = Fr_1 \cup Fr_2$ and $Tr = Tr_1 \cup Tr_2$.

We want to prove that, starting in any initial state s_0, P satisfies all the properties in the definition of a process, i.e. $P_1) \sim P_3)$ and $S_1) \sim S_4)$.

The proofs of $P_1) \sim P_3)$ are similar to those in [BHR84]. By way of illustration, we give the proof of $P_3)$ here.

- The proof of $P_3)$

  Suppose

  $$(tr, ref) \in Fr(s_0) \quad \text{and} \quad \forall v.\, (tr^\wedge < c.v >, \phi) \notin Fr(s_0).$$

  By the definition of P and the finiteness of Sv, there is a process, say P_1, with

  $$(tr, ref) \in Fr_1(s_0) \quad \text{and} \quad \forall v.\, (tr^\wedge < c.v >, \phi) \notin Fr_1(s_0).$$

  But P_1, being a process, has property $P_3)$, which gives

  $$(tr, ref \cup \{c\}) \in Fr_1(s_0).$$

Hence, it follows that

$$(tr,\ ref \cup \{c\}) \in Fr(s_0),$$

as required.

The proofs of $S_1) \sim S_4)$ are similar. We only give the details of the proof of $S_3)$.

- The proof of $S_3)$

  Suppose

  $$Tr(s_0,\ tr) = \perp \quad \text{and} \quad tr' \in Comm^*.$$

  By definition of P, there is a process, say P_1, with

  $$Tr_1(s_0,\ tr) = \perp.$$

  But P_1, being a process, has property $S_3)$, which gives

  $$Tr_1(s_0,\ tr^\wedge tr') = \perp \quad \text{for} \quad tr' \in Comm^*.$$

  And hence, by definition of P, it follows that

  $$Tr(s_0,\ tr^\wedge tr') = \perp \quad \text{for} \quad tr' \in Comm^*.$$

  as required.

This completes the proof.

Lemma A.2 *The intersection of two processes P_1 and P_2, $P_1 \prod P_2$, is the greatest lower bound of them under $\sqsubseteq_r$.*

$\square$

Proof:

Let $P_1 = (Fr_1,\ Tr_1)$ and $P_2 = (Fr_2,\ Tr_2)$ be two processes and $P = (Fr,\ Tr) = P_1 \prod P_2$, so that $Fr = Fr_1 \cup Fr_2$ and $Tr = Tr_1 \cup Tr_2$.

We first prove that $P_1 \prod P_2$ is a lower bound of P_1 and P_2 under $\sqsubseteq_r$. Then, we prove that it is the greatest lower bound.

$P_1 \prod P_2$ is a lower bound of P_1 and P_2, i.e. $P_1 \prod P_2 \sqsubseteq_r P_1$ and $P_1 \prod P_2 \sqsubseteq P_2$

The proofs of P_1 and P_2 are the same. We only give details for P_1. We check the three conditions of $\sqsubseteq_r$, for any state $s_0 \in State$:

1). By definition of $\prod$, we have

$$Fr_1(s_0) \subseteq Fr(s_0)\ (= Fr_1(s_0) \cup Fr_2(s_0));$$

2). Also by definition of $\prod$, we have

$$Tr_1(s_0,\ tr) \subseteq Tr(s_0,\ tr)\ (= Tr_1(s_0,\ tr) \cup Tr_2(s_0,\ tr)),$$

as required.

$\underline{P_1 \prod P_2 \text{ is the greatest lower bound of } P_1 \text{ and } P_2}$

Suppose $P' = (Fr', T')$ is a lower bound of P_1 and P_2.

1). We have, for any $s_0 \in State$,

$$Fr_1(s_0) \subseteq Fr'(s_0) \quad \text{and} \quad Fr_2(s_0) \subseteq Fr'(s_0),$$

and hence,

$$Fr_1(s_0) \cup Fr_2(s_0) \subseteq Fr'(s_0),$$

as required.

2). We have

$$Tr_1(s_0, tr) \subseteq Tr'(s_0, tr) \quad \text{and} \quad Tr_2(s_0, tr) \subseteq Tr'(s_0, tr),$$

and hence

$$Tr(s_0, tr) = Tr_1(s_0, tr) \cup Tr_2(s_0, tr) \subseteq Tr'(s_0, tr),$$

as required.

This completes the proof.

The union of two processes P_1 and P_2, $P_1 \bigsqcup P_2$ may not be in *Proc*, i.e. it may not be a process. For example, the intersection $Fr_1 \cap Fr_2$ of the failure sets of P_1 and P_2 may be empty which does not satisfy the definition of a process. Therefore we can, at most, prove that $(Proc, \sqsubseteq_r, \bot_{Proc})$ is a complete *semi*-lattice.

Lemma A.3 *The union of an ascending chain of processes,* $P_1 \sqsubseteq_r P_2 \sqsubseteq_r P_3 \sqsubseteq_r \cdots$, *exists, that is*

$$P = \bigsqcup_i P_i = (\bigcap_i Fr_i, \bigcup_i Tr_i).$$

□

Proof:

We only need to prove that $\bigcap_i Fr_i$ and $\bigcup_i Tr_i$ exist, for any initial state $s_0 \in State$ and $tr \in Comm^*$.

– Firstly, By definition of $\sqsubseteq_r$, we have

$$Fr_1(s_0) \supseteq Fr_2(s_0) \supseteq Fr_3(s_0) \supseteq \cdots,$$

and $Fr_i(s_0)$ is finite. Hence, $\bigcap_i Fr_i(s_0)$ exists as required.

– Secondly, from

$$Tr_1(s_0, tr) \supseteq Tr_2(s_0, tr) \supseteq Tr_3(s_0, tr) \supseteq \cdots,$$

and $Tr_i(s_0, tr)$ is finite, we have $\bigcap_i Tr_i(s_0, tr)$ exists, for every $s_0 \in State$ and $tr \in Comm^*$.

This completes the proof.

Lemma A.4 *The union of an ascending chain of processes,* $P_1 \sqsubseteq_r P_2 \sqsubseteq_r P_3 \sqsubseteq_r \cdots$,

$$P = \bigsqcup_i P_i = (\bigcap_i Fr_i, \bigcup_i Tr_i),$$

is a process.
□

Proof:
Let $P = (Fr, Tr) = \bigsqcup_i P_i$ with

$$Fr = \bigcap_i Fr_i \quad \text{and} \quad Tr = \bigcap_i Tr_i.$$

We want to prove that P starting in any state $s_0 \in State$, satisfies all the properties in the definition of a process, i.e. $P_1) \sim P_3)$ and $S_1) \sim S_4)$.

The proofs of $P_1) \sim P_3)$ are trivial. By way of illustration, we only give the proof of $P_3)$ here.

- The proof of $P_3)$
  Suppose

  $$(tr, ref) \in Fr(s_0) \quad \text{and} \quad \forall v. (tr^\wedge < c.v >, \phi) \notin Fr(s_0).$$

  This means that, for every P_i,

  $$(tr, ref) \in Fr_i(s_0),$$

  and there is at least a process, say P_k, such that

  $$\forall v. (tr^\wedge < c.v >, \phi) \notin Fr_k(s_0)$$

  because Sv is finite. We want to prove that $(tr, ref \cup \{c\}) \in Fr(s_0)$, and this will be true unless there is a process, say P_j, with

  $$(tr, ref \cup \{c\}) \notin Fr_j(s_0).$$

  If such a process existed, we would be able to find a process P_n in the chain with $n > k$ and $n > j$ such that

  $$P_k \sqsubseteq_r P_n \quad \text{and} \quad P_j \sqsubseteq_r P_n,$$

  that is

  $$Fr_n(s_0) \subseteq Fr_k(s_0) \quad \text{and} \quad Fr_n(s_0) \subseteq Fr_j(s_0).$$

  Then we have

  $$Fr_n = Fr_k(s_0) \cap Fr_j(s_0).$$

But, we should have, by the assumptions, that

$$(tr,\ ref) \in Fr_n(s_0),$$

$$\forall v.\ (tr^\wedge < c.v >,\ \phi) \notin Fr_n(s_0),$$

$$(tr,\ ref \cup \{c\}) \notin Fr_n(s_0).$$

This contradicts P_3) for P_n. Therefore, it must be the case that

$$(tr,\ ref \cup \{c\}) \in Fr(s_0).$$

The proofs of $S_1) \sim S_4)$ are as follows:

- The proof of S_1)

Suppose

$$Tr(s_0,\ tr) \neq \phi \quad \text{for } s_0 \in State \text{ and } tr \in Comm^*.$$

Because $Tr(s_0,\ tr) = \bigcap_i Tr_i(s_0,\ tr)$, we have

$$Tr_i(s_0,\ tr) \neq \phi, \quad \text{for all } i.$$

Hence, P_i being a process, by S_1, we have

$$tr \in traces(P_i(s_0)), \qquad \text{for all } i.$$

Therefore we have

$$(tr,\ \phi) \in Fr(s_0)\ (= \bigcap_i Fr_i(s_0)),$$

as required.

- The proof of S_2)

Suppose

$$(tr,\ ref) \in Fr(s_0) \quad \text{and} \quad Tr(s_0,\ tr) = \phi.$$

We want to prove that $(tr,\ ref \cup \{\surd\}) \in Fr(s_0)$. By definition of $\sqsubseteq_r$, we have

$$Tr(s_0,\ tr) = \bigcap_i Tr_i(s_0,\ tr) = \phi$$

By the finiteness of $State_\perp$, there must exist a k such that

$$Tr_k(s_0,\ tr) = \phi, \qquad \text{for all } i \geq k.$$

Hence, P_i being a process, by S_2), we have

$$(tr,\ ref \cup \{\surd\}) \in Fr_i(s_0), \quad \text{for all } i \geq k.$$

$\{ P_i \mid i \geq 1 \}$ being a chain , we have

$$(tr,\ ref \cup \{\surd\}) \in Fr(s_0)\ \ (= \bigcap_i Fr_i(s_0)),$$

as required.

- The proof of S_3)

  Suppose
  $$Tr(s_0,\ tr)\ =\ \phi \quad \text{and} \quad tr' \in Comm^*.$$
  By definition of P, we have
  $$Tr_i(s_0,\ tr)\ =\ \perp \qquad \text{for all } i.$$
  P_i, beign a process, has the property S_3, which gives
  $$Tr_i(s_0,\ tr^\wedge tr')\ =\ \perp \qquad \text{for } tr' \in Comm^*.$$
  Hence, it follows, by definition of P, that
  $$Tr(s_0,\ tr^\wedge tr')\ =\ \bigcap_i Tr_i(s_0,\ tr^\wedge tr')\ =\ \perp,$$
  as required.
- The proof of S_4)

  The proof of S_4) is similar to that of S_3), and is omitted.

This completes the proof.

Lemma A.5 *The union of an ascending chain of processes,* $P_1 \sqsubseteq_r P_2 \sqsubseteq_r P_3 \sqsubseteq_r \cdots$,
$$\bigsqcup_i P_i\ =\ (\bigcap_i Fr_i,\ \bigcap_i Tr_i\ ),$$
is the least upper bound of them.
□

Proof:

Let $P\ =\ (Fr,\ Tr)\ =\ \bigsqcup_i P_i$, with
$$Fr\ =\ \bigcap_i Fr_i \quad \text{and} \quad Tr\ =\ \bigcap_i Tr_i,$$
We want to prove that P is the least upper bound of $P_1 \sqsubseteq_r P_2 \sqsubseteq_r P_3 \sqsubseteq_r \cdots$, i.e. for any P' which is an upper bound of all P_i's, $P \sqsubseteq_r P'$. We first prove that P is an upper bound of all P_i's.

- P is an upper bound of all P_i's

  For any P_i in $\{P_i | i \geq 0\}$, we check the two conditions of $\sqsubseteq_r$: for any $s_0 \in State$,

  1). we have, by definition of $\bigsqcup$
  $$Fr_i(s_0)\ \supseteq\ Fr(s_0)\,(=\ \bigcap_i Fr_i(s_0)\,);$$

2). and, also by definition of $\bigsqcup$ and the fact that $P_i's$ is a chain, we have

$$Tr_i(s_0,\ tr) \supseteq Tr(s_0,\ tr)\ (=\bigcap_i Tr_i(s_0,\ tr)\ ), \qquad \text{for every } tr \in Comm^*$$

as required.

Then we prove that it is the least upper bound.

- $\underline{\underline{P \text{ is the least upper bound}}}$

Suppose P' is an upper bound of all $P_i's$.

1). We have, for any $s_0 \in State$ and $tr \in Comm^*$

$$Fr'(s_0) \subseteq Fr_i(s_0), \qquad \text{for all } i,$$

and hence,

$$Fr'(s_0) \subseteq Fr(s_0)\ (=\bigcap_i Fr_i(s_0)\ ),$$

2). and

$$Tr'(s_0,\ tr) \subseteq Tr_i(s_0,\ tr),$$

and hence

$$Tr'(s_0,\ tr) \subseteq Tr(s_0,\ tr)\ (=\bigcap_i Tr_i(s_0,\ tr)\ ),$$

as required.

This completes the proof.

Now we can give the following main theorem.

Theorem A.1 *(* $Proc$, $\sqsubseteq_r$, $\perp_{Proc}$ *) is a complete semi-lattice.*

□

Proof:

This follows by the definition of a semilattice, Lemma A.1, Lemma A.2, Lemma A.3, Lemma A.4 and Lemma A.5.