Report Number: WUCS-90-24

1990-06-20

Introduction to Communicating Sequential Processes

Luming Lai

In the last two decades, mathematical theories have been helping computer scientists see, in a fresh light, problems in the area of programming methodology and solve these problems more efficiently and reliably than before. In this series of seminars we demonstrate the application of mathematics in parallel languages and programming.

INTRODUCTION TO COMMUNICATING SEQUENTIAL PROCESSES

Luming Lai

WUCS-90-24

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

Introduction to Communicating Sequential Processes (CSP)[*]

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

June 20, 1990

1 Introduction

In the last two decades, mathematical theories have been helping computer scientists see, in a fresh light, problems in the area of programming methodology and solve these problems more efficiently and reliably than before. In this series of seminars we demonstrate the application of mathematics in parallel languages and programming.

- **Purposes**
  The main purpose of this seminar is to demonstrate the importance of mathematics in computer science: 1) how to apply some mathematical theories, like algebra, predicate calculus, set theory, etc., into the concurrency world, that is, how to use these theories to study the specification, design and implementation of concurrent systems, particularly parallel programs with communications; 2) how to find and solve both current and future problems in the area of programming methodology by using mathematics.

  To achieve these goals, we use C.A.R. Hoare's CSP as a paradigm.

- **What is CSP**
  CSP, a theory of Communicating Sequential Processes, is a mathematical notation for expressing and reasoning about systems of concurrent processes. By *expressing* we mean the specification and implementation of the concurrent system. Without a formal notation it is difficult to describe a process precisely enough to modify it or to contrast it with contending designs. By *reasoning about* we mean the modification of the design, the development of a concurrent system from its specification, and the verification of such an implementation as correct with respect to the original design. CSP supports a body of fairly simple algebraic laws and a semantic theory which enable us to do so in a mathematical framework.

- **Applications of CSP**
  Applications of CSP: CSP uses the following operators to construct processes: parallel

[*]A research seminar notes, Spring, 1990

composition, external choice, internal choice, communication, abstraction and sequential composition. The resulting notation is expressive enough to describe the problems in the following areas: multiprocessing systems, operating systems, distributed systems, systems using remote procedure calls, and systolic arrays of processors.

- **Background**
  1. high-school algebra;
  2. concepts of set theory;
  3. notations of the predicate calculus.

- **Summary**
  Seminar 1 demonstrates that CSP provides a notation which is rich enough for expressing and reasoning about systems of concurrent processes. Somes examples are given to introduce CSP notations and to show how CSP can be used to model (or describe) some potentially complex systems by building the specifications in small steps. Laws governing the behaviour of CSP operators are given. They give an algebraic semantics to CSP notation.

  Seminar 2 gives a mathematical treatment to CSP, mainly its mathematical semantics. A trace model is developed for CSP notations and some interesting properties of CSP are proven in the model. A denotational semantics is given to CSP notations and laws about CSP operators are proved. A specification method is developed which handles only safety properties of communicating processes. Proof rules for each CSP operator are given which enable us to prove a CSP process correct with respect to its specification.

  Seminars 3 develops a more sophisticated model, the failures model for CSP, in which determinism and nondeterminism can be distinguished. Doing so the resulting specification method can handle both safety and liveness properties. Proof rules in this model are developed.

2 Seminar One : A Description Of A PingPong Game

During a CSP course it is easy to lose sight of the wood for the trees; after spending several hours immersed in the formal properties of an operator, one is apt to forget why the operator was deemed to have been important in the first place. It has thus been found convenient to begin a CSP course with a seminar whose purpose is to provide an overview and informal introduction to the features of CSP.

In this seminar we attempt to give such an introduction. Particular features are:

(a) the use of a small example to introduce the notation, and demonstration of how CSP can be used to model a potentially complex system by building the specification in small steps;

(b) early emphasis on the combination for the internal and external choice;

(c) contrast between CSP and the notation of finite automata and regular expressions; and

(d) introduction of algebraic laws governing the behaviour of CSP constructs.

Some basic concepts:

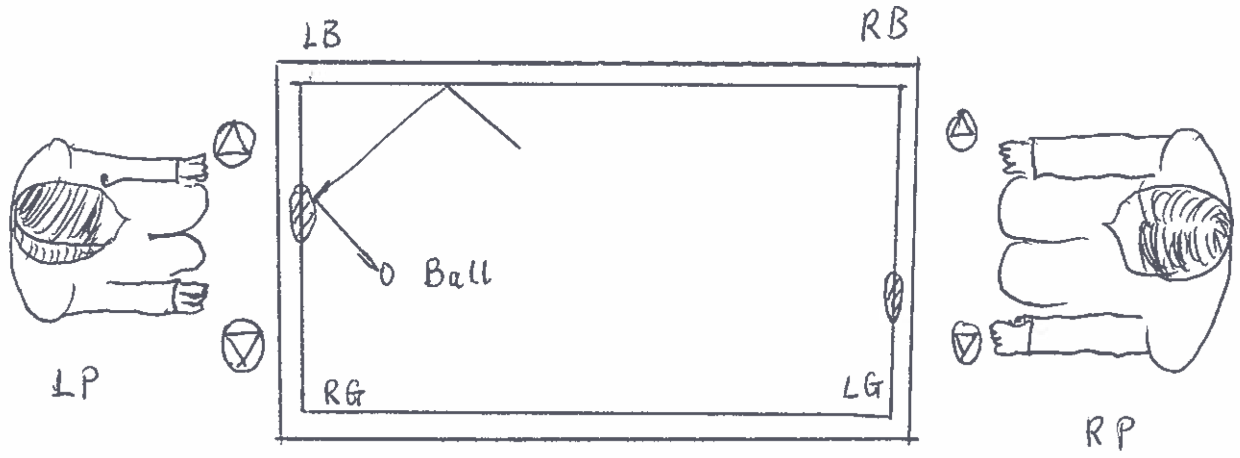
- The word **process** is used to stand for the behaviour of an object. There are two ways to describe a process:
  1. by CSP syntax.
     Result: progams.
  2. by its semantics.
     Result: specifications

  This seminar describes a PingPong game by using CSP syntax.

- The **alphabet** of a process is the set of names of events which are considered relevant for a particular description of an object. The alphabet is a permanently predefined property of an object. That is, it is logically impossible for an object to engage in an object outside its alphabet.

2.1 A PingPong Game

Task: describe a PingPong game formally in CSP.



PingPong Game

2.2 Informal Description

PingPong is a game for two players who sit at the ends of a small table. Set into the table is a horizontal video monitor which displays a ball bouncing around inside a four-wall court. The wall nearest the right-hand player, RP, is the goal, LG, of the left-hand player, LP, and vice visa. To defend his opponent's goal, the RP controls a bat, RB, which can be moved along LG by pressing "moving right" button or "moving left" button (judged from the viewpoint of RG), or neither. The ball starts at the middle of the court at a random angle (not parallel to the goals) and collides elastically with the bats and walls. But, if it reaches a player's goal, that player wins and the session ends.

The first difficulty is to decide at what level to model the game. We can describe the game in one of the following ways:

1. the starting time, the ending time and the winner; or

2. second by second, the intensity of the screen and the movement of the players.

We choose the second way to describe the PingPong game which contains more information than the first which is called more high-level or more abstract.

The next difficulty is to decide the relevant events, i.e. its alphabet. We are not interested in

- the time nor how fast the ball moves, nor

- the player's physical attributes,

but

- how players control their bats (by pressing the buttons)

- how the bats deflect the ball.

Thus we shall focus on an intermediate description which ignores time and most physical attributes of the players of the games.

2.3 A Top-down Description

We conceive of the game as proceeding as the two players control their bats by button and the bats deflect the ball which rebounds around the court. Our first step is to isolate the five interacting processes:

LP the left-hand player

RP the right-hand player

LB the left-hand bat

RB the right-hand bat

Ball the ball

It is our intention to describe the game as proceeding with evolution in parallel of these processes

$$\text{PingPong} = \mathbf{LP} \parallel \mathbf{LB} \parallel \mathbf{Ball} \parallel \mathbf{RB} \parallel \mathbf{RP} \qquad (1)$$

A process is defined by describing the whole range of its potential behaviour, for example, the right-hand bat. Frequently, there will be a choice between several actions, like the movement of the bat to the right or left. On each occasion, the choice of which event will actually occur can be controlled by the environment (in this case the right-hand player and the ball) within which the process evolves. For example, it is the player who controls which direction the bat moves. Fortunately, the environment of a process itself may be described as a process (the right-hand player). This permits investigation of the behaviour of a complete system composed from the process together with its environment, acting and interacting with each other as they evolve concurrently. The complete system should also be regarded as a process, whose range of behaviour is definable in terms of the behaviour of its component processes, like $RP \parallel RB$; and the system may in turn be placed within a yet wider environment, like $RP \parallel RB$ in PingPong.

The informal meaning of $\parallel$ is: when two processes are running in parallel like $RP \parallel RB$, the components must synchronized on their common actions or events (the left bat responds to the left-hand player's wishes) although an event which can be performed by only one of them, occurs whenever that process permits it to (the bat is insensitive to its player's victory).

Laws governing $\parallel$ are given bellow, whenever a new CSP construct is introduced.

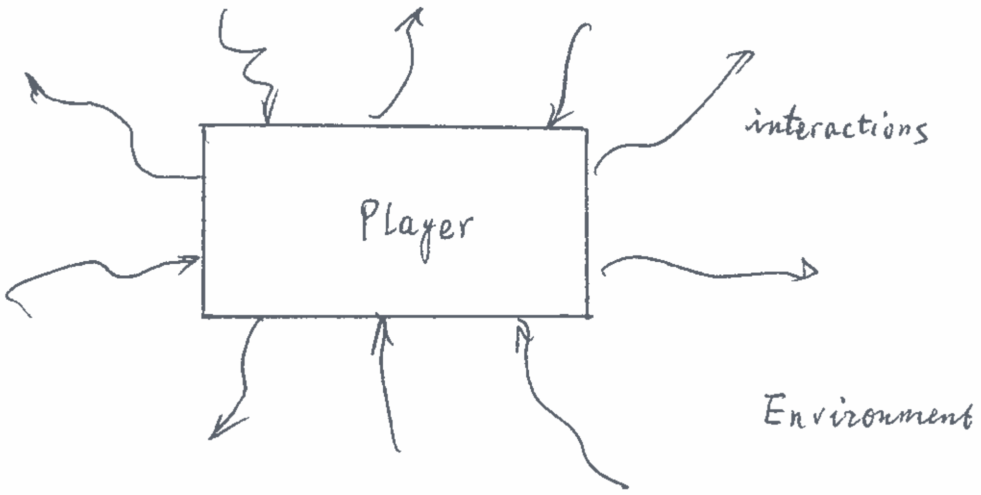
L1 $P \parallel Q \;=\; Q \parallel P$

L2 $P \parallel (Q \parallel R) \;=\; (P \parallel Q) \parallel R.$

But let us first discuss the players, LP and RP.

2.4 Specifications of Players

Think of a process as a black box. Its internal constitution is concealed from us but we can nevertheless describe it by its interactions with its environment.



Player

The relevant events in which the left-hand player can engage are

right	push the right-hand button
left	push the left-hand button
stay	push neither

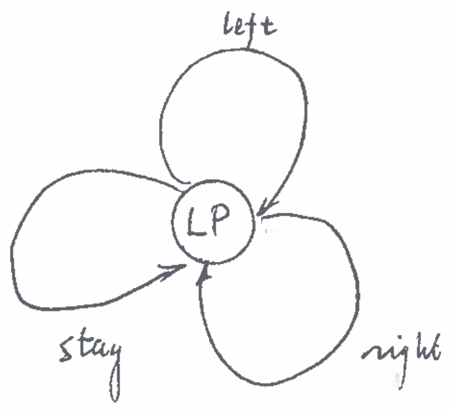
We say these events constitute the alphabet of LP,

$$\alpha(LP) \;\widehat{=}\; \{left,\ right,\ stay\}.$$

The left-hand player can push buttons as frequently as he likes and in any order. The choice of what he does is determined by factors not revealed at this level of abstraction. So in our model LP can, at any stage, engage in any of the events in its alphabet.

To contrast between CSP and the notation of finite automata and regular expressions, we give

(a) A diagram of a finite automaton



(b) A regular grammar

$$LP \;=\; left\ LP \;\mid\; right\ LP \;\mid\; stay\ LP \;\mid \epsilon$$

where ϵ denotes the empty string and its language consists of all the finite sequences of *left*, *right* and *stay*, denoted

$$\{\ left,\ right,\ stay\ \}^*$$

The way in which we describe LP using CSP makes apparent that the choice between the different events in its alphabet is made internally by LP and cannot be influenced by its environment.

The CSP description of LP:

$$LP \;=\; (\,left \rightarrow LP\,) \sqcap (\,right \rightarrow LP\,) \sqcap (\,stay \rightarrow LP\,) \tag{2}$$

This recursive definition can also be written as

$$LP \;=\; \mu X.(\,left \rightarrow X\,) \sqcap (\,right \rightarrow X\,) \sqcap (\,stay \rightarrow X\,) \tag{3}$$

Some care is necessary to ensure that such a definition makes sense. For instance LP is not defined by either

$$LP \;=\; LP \qquad \text{or} \qquad LP \;=\; LP \sqcap LP$$

because everything is a solution to these two equations.

Definition 2.1 *(Guarded Process)*
A process is guarded if it begins with a prefix.

We introduce some CSP operators.

- $\mathbf{stop}_A$ is process with alphabet A which never engages in anty events in A. This describes the behaviour of a broken process: although it is equipped with the physical capabilities to engage in the events of A, it never exercises those capabilities. Note $\mathbf{stop}_A \neq \mathbf{stop}_B$ if $A \neq B$.

  L3 $P \parallel \mathbf{stop}_{\alpha P} = \mathbf{stop}_{\alpha P}$

  L4 $\mathbf{stop} \neq (d \rightarrow P)$

- **Prefix**

  $$a \rightarrow P$$

  It describes a process which first engages in the event a and then behaves exactly the same as P.

  $$\alpha(a \rightarrow P) = \alpha P \qquad \text{provided that } a \in \alpha P$$

  Laws governing the prefix operator are: let $a \in (\alpha P - \alpha Q)$, $b \in (\alpha Q - \alpha P)$ and $\{c, d\} \in (\alpha P \cap \alpha Q)$

  L4 $(c \rightarrow P) \neq (d \rightarrow Q) \qquad \text{if } c \neq d$

  L5 $(c \rightarrow P) = (c \rightarrow Q) \equiv P = Q$

  L6 $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$

  L7 $(c \rightarrow P) \parallel (d \rightarrow Q) = \mathbf{stop} \qquad \text{if } c \neq d$

  L8 $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$

  L9 $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$

- **Internal choice** or nondeterministic choice

  $$P \sqcap Q$$

  is a process which behaves either like P or like Q, where the selection between them is made arbitrarily, without the knowledge or control of the external environment.

  $$\alpha(P \sqcap Q) = \alpha P = \alpha Q$$

  An example of the internal choice is a change machine: a one-dollar bill is inserted and different combinations of coins may be returned, but the choice cannot be controlled by the customer.

  The algebra laws governing nondeterministic choice are as follows:

  L10 $P \sqcap P = P$

  L11 $P \sqcap Q = Q \sqcap P$

  L12 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

L13 $a \rightarrow (P \sqcap Q) \;=\; (a \rightarrow P) \sqcap (a \rightarrow Q)$

L14 $P \parallel (Q \sqcap R) \;=\; (P \parallel Q) \sqcap (P \parallel R)$

L15 $(P \sqcap Q) \parallel R \;=\; (P \parallel R) \sqcap (Q \parallel R)$

- **recursion**

  If $F(X)$ is a guarded expression containing the process name X, and A is the alphabet of X, then we claim that the equation

  $$X \;=\; F(X)$$

  has a unique solution with alphabet A. It is sometime convenient to denote this solution by the expression

  $$\mu X : A.\, F(X).$$

  Therefore, LP can also be written as

  $$LP \;=\; \mu X : \alpha LP.\, (\mathit{left} \rightarrow X) \sqcap (\mathit{right} \rightarrow X) \sqcap (\mathit{stay} \rightarrow X).$$

  Laws about recursion are:

  L16 If $F(X)$ is a guarded expression,

  $$(Y = F(Y)) \;\equiv\; (Y = \mu X.F(X))$$

  L17 $\mu X.F(X) = F(\mu X.F(X))$

  The recursion operator is not distributive through nondeterminism.

  $$\begin{aligned} P &= \mu X.((a \rightarrow X) \sqcap (b \rightarrow X)) \\ Q &= (\mu X.(a \rightarrow X)) \sqcap (\mu X.(b \rightarrow X))) \end{aligned}$$

  are different processes.

We introduce traces of a process.

Definition 2.2 *(traces)*
a trace of a process is a finite sequence of events in which the process has engaged up to some moment in time.

The set of all finite traces of a process P is denoted

$$\mathit{traces}(P).$$

For example, the traces of LP are

$$\mathit{traces}(LP) \;=\; \{\mathit{left},\, \mathit{right},\, \mathit{stay}\}^*.$$

- Operations on traces:
  - $<>$ denotes the empty trace. $< x, y >$ consists of two events, x followed by y. $< x >$ is a trace containing only one event x.

- $tr^\wedge tr_1$ is the catenation of two traces tr and tr_1.
- $tr \upharpoonright A$ denotes the trace when restricted to symbols in the set A; it is formed from tr simply by omitting all the symbols outside A. For example, $< a, b, c, d > \upharpoonright \{a, d\} = < a, d >$.
- tr_0 is the first event of trace tr and tr' is the trace obtained by removing the first symbol of tr.
- $\#tr$ is the length of tr.
- $tr_1 \leq tr_2 \;=\; \exists\, tr_3.\ tr_1^\wedge tr_3 \;=\; tr_2$.

Laws about traces:

L18 $traces(\mathbf{stop}) \;=\; \{<>\}$

L19 $traces(c \rightarrow P) \;=\; \{<>\} \cup \{< c >^\wedge\, tr \mid tr \in traces(P)\,\}$

L20 $traces(\mu\, X : A.F(X)) \;=\; \bigcup_{n \geq 0} traces(F^n(\mathbf{stop}_A)).$

L21 $traces(P \sqcap Q) \;=\; traces(P) \cup traces(Q)$

L22 $traces(P \parallel Q) \;=\; \{tr \mid tr \upharpoonright \alpha P \in traces(P) \;\wedge\; tr \upharpoonright \alpha Q \in traces(Q)$
$\wedge\; tr \in (\alpha P \cup \alpha Q)^*\}$

Finally, as a trace is a sequence of symbols recording the events in which a process P has engaged up to the moment in time. From this it follows that $<>$ is a trace of every process up to the moment in which it engages in its very first event. Furthermore, if $(tr^\wedge tr1)$ is a trace of process up to some moment, then tr must have been a trace of that process up to some earlier moment. Finally, every event that occurs must be in the alphabet of the process. Thus we have the following three laws:

L23 $<> \in traces(P)$

L24 $tr^\wedge tr1 \in traces(P) \Rightarrow tr \in traces(P)$

L25 $traces(P) \subseteq (\alpha P)^*$

We define an **after** operator as follows:

$$P/tr$$

is a process which behaves the same as P does from the time after P has engaged in all the events in the trace tr. If tr is not a trace of P, P/tr is not defined. For example,

(1) $P/ <> = \; P$

(2) $(c \rightarrow P)/ < c > = \; P.$

Laws about **after** are:

L26 $traces(P/tr) \;=\; \{tr1 \mid tr^\wedge tr1 \in traces(P)\}$, provided that $tr \in traces(P)$.

L27 $(P \parallel Q)/tr \;=\; (P/(tr \upharpoonright \alpha P)) \parallel (Q/(tr \upharpoonright \alpha Q))$

L28 $(P \sqcap Q)/tr \;=\; Q/tr, \qquad \text{if } tr \in (traces(Q) - traces(P))$
$\qquad = \; P/tr, \qquad \text{if } tr \in (traces(P) - traces(Q))$
$\qquad = \; (P/tr) \sqcap (Q/tr), \qquad \text{if } tr \in (traces(P) \cap traces(Q))$

Let us return to our PingPong game. So far, we have only described the behaviour of LP when he is playing. At the end of the game the left-hand player behaves like either

$$(lwin \rightarrow \mathbf{stop}) \;\; or \;\; (rwin \rightarrow \mathbf{stop}).$$

The choice between them is not made by the player himself (would he ever choose th latter choice?), so $\sqcap$ is an appropriate combinator. The choice is made by LP's environment because it depends on the position of the ball. We must introduce a new, environment-controlled, choice combinator: it is written $\Box$ and called **external** or **determinisitc**, choice. So at the end of the game LP behaves like

$$(lwin \rightarrow \mathbf{stop}) \,\Box\, (rwin \rightarrow \mathbf{stop})$$

and the choice between such behaviour and nontermination (as described in our previous version of LP) is again made by LP's environment — it depends on whether the ball is at a goal. Thus finally

$$\begin{aligned} \alpha(LP) \;&\hat{=}\; \{left,\, right,\, stay,\, lwin,\, rwin\} \\ LP \;&\hat{=}\; ((left \rightarrow LP) \sqcap (right \rightarrow LP) \sqcap (stay \rightarrow LP)) \\ &\quad \Box \\ &\quad ((lwin \rightarrow \mathbf{stop}) \,\Box\, (rwin \rightarrow \mathbf{stop})) \end{aligned}$$

The Deterministic choice $(P \Box Q)$ offers the choice between P and Q to its environment, provided that this choice is made at the very first action. If this action is not a possible first action of P, then Q will be selected; but if Q cannot engage initially in the action, P will be selected. If however, the first action is possible for both P and Q then the choice between them is nondeterministic. We stipulate

$$\alpha(P \Box Q) \;=\; \alpha P \;=\; \alpha Q$$

In the case that no initial event of P is also possible for Q, the external choice can be written as, for example

$$(c \rightarrow P \Box d \rightarrow Q) \;=\; (c \rightarrow P \mid d \rightarrow Q) \qquad \text{provided that } c \neq d$$

or in a more general notation

$$(c \rightarrow P \mid d \rightarrow Q) \;=\; (x : B \rightarrow R(x))$$

where $B = \{c,\, d\}$ and $R(x) \;=\; \mathbf{if}\; x = a \;\mathbf{then}\; P \;\mathbf{else}\; Q$.
Laws of $\Box$ are given as follows:

L29 $P \Box P \;=\; P$

L30 $P \Box Q \;=\; Q \Box P$

L31 $P\Box(Q\Box R) \;=\; (P\Box Q)\Box R$

L32 $P\Box\mathbf{stop} \;=\; P$

L33 $P\Box(Q\sqcap R) \;=\; (P\Box Q)\sqcap(P\Box R)$

L34 $P\sqcap(Q\Box R) \;=\; (P\sqcap Q)\Box(P\sqcap R)$

L35

$$\begin{array}{rl} & (x:A\to P(x))\Box(y:B\to Q(y)) \\ = & (z:(A\cup B)\to(\mathit{if}\ z\in(A-B)\ \mathit{then}\ P(z) \\ & \qquad \mathit{else\ if}\ z\in(B-A)\ \mathit{then}\ Q(z) \\ & \qquad \mathit{else\ if}\ z\in(B\cap A)\ \mathit{then}\ (P(z)\sqcap Q(z)))) \end{array}$$

L36 $\mathit{traces}(P\Box Q) \;=\; \mathit{traces}(P)\cup\mathit{traces}(Q)$

L37 $(P\Box Q)/tr \;=\; Q/tr$, if $tr\in(\mathit{traces}(Q)-\mathit{traces}(P))$
$= P/tr$, if $tr\in(\mathit{traces}(P)-\mathit{traces}(Q))$
$= (P/tr)\sqcap(Q/tr)$, if $tr\in(\mathit{traces}(P)\cap\mathit{traces}(Q))$

L38 $(a\to P)\parallel(b\to Q) \;=\; (a\to(P\parallel(b\to A))\mid b\to((a\to P)\parallel Q))$,
provided that $a\in(\alpha P-\alpha Q)$ and $b\in(\alpha Q-\alpha P)$.

L39 Let $P=(x:A\to P(x))$ and $Q=(y:B\to Q(y))$, then

$$(P\parallel Q) \;=\; (z:C\to P'\parallel Q'),$$

where $C=(A\cap B)\cup(A-\alpha Q)\cup(B-\alpha P)$ and $P'=$ if $z\in A$ then $P(z)$ else P, and $Q'=$, if $z\in B$ then $Q(z)$ else Q.

These laws permit a process defined by concurrency to be redefined without concurrency, as shown in the following example
Example 2.1
Let $\alpha P=\{a,c\}$, $\alpha Q=\{b,c\}$, $P=(a\to c\to P)$ and $Q=(c\to b\to Q)$. Then,

$$\begin{array}{rcll} P\parallel Q & = & (a\to b\to P)\parallel(c\to b\to Q) & \text{by defs} \\ & = & a\to((c\to P)\parallel(c\to b\to Q)) & \mathbf{L8} \\ & = & a\to c\to(P\parallel(b\to Q)) & \mathbf{L6} \\ P\parallel(b\to Q) & = & & \\ & = & (a\to(c\to P)\parallel(b\to Q) & \\ & & \mid b\to(P\parallel Q)) & \mathbf{L38} \\ & = & (a\to b\to((c\to P)\parallel(Q) & \\ & & \mid b\to(P\parallel Q)) & \mathbf{L8} \\ & = & (a\to b\to c\to(P\parallel(b\to Q) & \end{array}$$

$$\begin{aligned} & \mid b \to a \to c \to (P \parallel (b \to Q))) && \text{by } \mathbf{L6,\ 7,\ 8} \\ = \ & \mu X.(a \to b \to c \to X \\ & \mid b \to a \to c \to X && \text{this is guarded} \end{aligned}$$

We have

$$(P \parallel Q) \ = \ (a \to c \to \mu X.((a \to b \to c \to X) \mid (b \to a \to c \to X)))$$

After having completed the specification of LP, we next observe the similarity between LP and RP.
Exercise Define RP in CSP.

There is a systematic way to exploit such a isomorphism, called **relabelling**. Let us introduce three **new** events *left′*, *right′* and *stay′* (why new?), and define a function

$$f : \alpha(P) \to \{\mathit{left}', \mathit{right}', \mathit{stay}', \mathit{lwin}, \mathit{rwin}\}$$

by setting

$$\begin{aligned} f(\mathit{left}) &= \mathit{right}' \\ f(\mathit{right}) &= \mathit{left}' \\ f(\mathit{stay}) &= \mathit{stay}' \\ f(\mathit{lwin}) &= \mathit{lwin} \\ f(\mathit{rwin}) &= \mathit{rwini} \end{aligned}$$

We define a process named $f(LP)$ as follows: its alphabet is $f(\alpha LP)$ and it engages in event$f(e)$ iff LP engages in event e. This "lifts" the function f from events to processes, and enable us to define

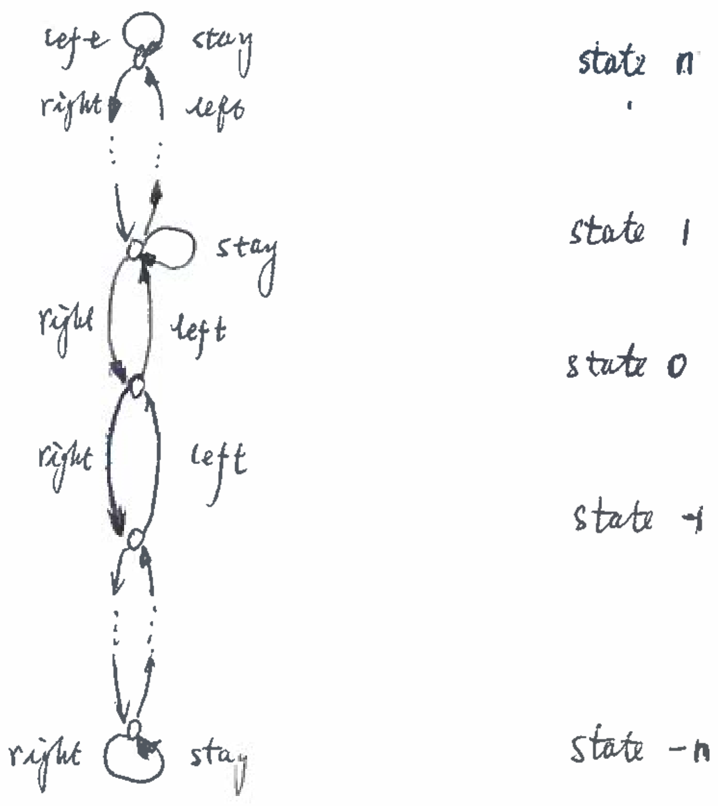
$$RP \ \widehat{=} \ f(LP)$$

This finished our specification of the players in CSP.

2.5 The Specifications of Bats

The left-hand bat, LB, starts in the centre of RP's goal and when it moves, it does so in small uniform jumps left or right (directed from the point of view of LP) as far as the extremes of the court. We deems LB to be capable of at most n moves in either direction from its original position, for some $n \in \mathcal{N}$ at least 1. In the implementation n will be quite large, its actual value being determined by matters of graphical and computational efficiency. But let us abstract from these and treat n as a constant of the specification.

- A finite automaton description.



- A grammar description.

$$\begin{aligned} LB &= LB_0 \\ LB_t &= left\ LB_{t+1} \mid right\ LB_{t-1} \mid stay\ LB_t \mid \epsilon \\ LB_{-n} &= left\ LB_{-n+1} \mid right\ LB_{-n} \mid stay\ LB_{-n} \\ LB_n &= left\ LB_n \mid right\ LB_{n-1} \mid stay\ LB_n \end{aligned}$$

  The language of this grammar is, of course, identical to that for LP. At this level of abstraction we have made no use of the "position marker" n and were we not to make use of it later in the specification, our description would be unnecessarily complicated (not **fully abstract**).

For convenience, we choose, as we did with LP, to specify LB in two steps. Again the second version supersedes the first and so we use the same name, LB, in both. We express LB in CSP

$$\begin{aligned} \alpha(LB) &= \{left,\ right,\ stay\} \\ LB &= LB_0 \\ LB_t &= (left \rightarrow LB_{t+1}) \mid (right \rightarrow LB_{t-1}) \mid (stay \rightarrow LB_t) \\ &\qquad \text{provided } |t| \leq n-1 \\ LB_{-n} &= (left \rightarrow LB_{-n+1}) \mid (right \rightarrow LB_{-n}) \mid (stay \rightarrow LB_{-n}) \\ LB_n &= (left \rightarrow LB_n) \mid (right \rightarrow LB_{n-1}) \mid (stay \rightarrow LB_n). \end{aligned}$$

Note that the choice between events *left*, *right* and *stay* is not made by the bat, but its environment, that is, LP. This distinction does not appear in the finite automaton for LB.

- A simplification of LB_n

  LB_n is permitted to engage in the event *left* without changing its state (i.e. its position). Can we bar it from performing a *left* by defining instead

$$\begin{aligned} LB_n \;=\; & (right \rightarrow LB_{n-1} \\ & \mid stay \rightarrow LB_n). \end{aligned}$$

  This yields a mismatch between LP and LB which can lead to a deadlock. Let us consider

$$LP \parallel LB.$$

  It is the process which results from LP and LB interacting in parallel. Its alphabet is

$$\alpha(LP \parallel LB) \;=\; \{left,\; right,\; stay,\; lwin,\; rwin\}$$

  and an event common to both LP and LB is performed by $LP \parallel LB$ when and only when it is performed by both LP and LB. So, from the start, LP chooses internally between *left*, *right* and *stay*, and LB allows it to make this choice and so engages in the same event. It is, after all, the player who controls the next position of the bat, and **not** the bat which determines the next action of the player. The processes proceed in step, synchronising on their common events. Have we made the modification to LB as above, $LP \parallel LB$ would proceed in the same way until LB reached an extreme state – say LB_n. Now if LP insisted on engaging in the event *left*, then since LB also contains *left* in its alphabet but is not prepared to engage in it at this stage, deadlock would occur.

We leave the definition of the right-hand bat to the reader.
Exercise: Define RB.
Anwser:
Define $f' \;=\; f \restriction \{left,\; right,\; stay\}$, where f was defined in section 2.2.3 (here $f \restriction E$ denotes the restriction of function f to set E), then

$$RB \;=\; f'(LB).$$

This time it is important that LB and RB be mirror images of each other in order for RB to present the correct choice of events to its environment in its extreme states.

Since we are defining the five processes LP, RP, LB, RB and Ball to be self-contained, some synchronisation is necessary between them. When the ball reaches the left-hand goal, for example, it must find out the position of the defending bat so that it can determine whether the game ends or whether the ball should rebound. This implies, from the meaning of $\parallel$, that LB must always be prepared to offer its position (containing the ball) to its environment: an offer which will be accepted by the ball whenever it reaches the left-hand goal. For $|t| \leq n$ we must thus incorporate in the alphabet of LB the events (to be common with those of Ball)

$$lposition_t.$$

For reasons which will become apparent later on, we rewrite these events

$$lposition!t.$$

The modification to LB is thus to augment its alphabet by the 2n+1 new events

$$\{lposition!t \ : \ |t| \leq n\},$$

and to permit its environment to read them.

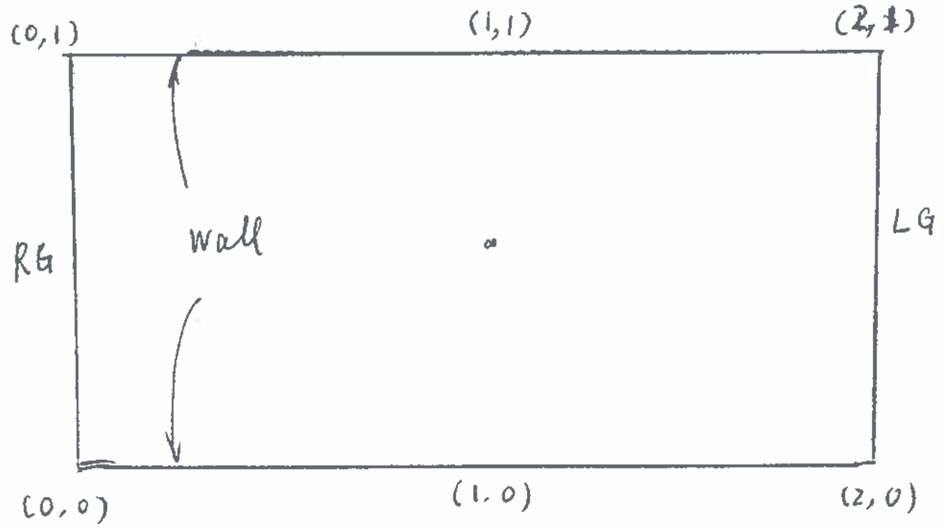
$$\begin{aligned}
\alpha(LB) &= \{left, \ right, \ stay\} \cup \{lposition!t \ : \ |t| \leq n\} \\
LB &= LB_0 \\
LB_t &= (left \rightarrow LB_{t+1} \\
&\quad \mid right \rightarrow LB_{t-1} \\
&\quad \mid stay \rightarrow LB_t \\
&\quad \mid lposition!t \rightarrow LB_t) \\
&\qquad \text{provided } |t| \leq n-1 \\
LB_{-n} &= (left \rightarrow LB_{-n+1} \\
&\quad \mid right \rightarrow LB_{-n} \\
&\quad \mid stay \rightarrow LB_{-n} \\
&\quad \mid lposition!t \rightarrow LB_{-n}) \\
LB_n &= (left \rightarrow LB_n \\
&\quad \mid right \rightarrow LB_{n-1} \\
&\quad \mid stay \rightarrow LB_n \\
&\quad \mid lposition!t \rightarrow LB_n).
\end{aligned}$$

What happens when the ball reaches the goal and the bat, only one step away, is frantically directed by its player to cover the ball's progress? Does the bat respond to its player's wish or does it reply to the ball's request for its position? The answer is simple: it performs whichever event reaches it first by virtue of preceding its opponent in the trace of events from LB's environment.
Exercise: Define RB.

2.6 The Specification of Ball

It is now convenient to settle upon some notations for the court. The set of possible positions for the ball is

$$Court \ \widehat{=} \ \{(x, \, y) \in R^2 \mid 0 \leq x \leq 2 \ \wedge \ 0 \leq y \leq 2\} \ = \ [0, \, 2] \times [0, \, 2]$$



The segments of the boundary of the court which interest us are:

$$\begin{aligned}
Wall &\mathrel{\widehat{=}} \{(x,\ y) \in R^2 \mid 0 < x < 2 \ \wedge\ 0 < y < 2\} \\
RG &\mathrel{\widehat{=}} \{(x,\ y) \in R^2 \mid x = 0 \ \wedge\ 0 \leq y \leq 1\} \\
LG &\mathrel{\widehat{=}} \{(x,\ y) \in R^2 \mid x = 2 \ \wedge\ 0 \leq y \leq 1\} \\
Court^0 &\mathrel{\widehat{=}} Court\ -\ (Wall \cup LG \cup RG)
\end{aligned}$$

In this notation, LB starts at (0, 1/2) and is at (0, 1) when it is at its left-most state.

The ball is to start at (1, 1/2) in a random direction and bounces elastically (so that the angle of incidence equals the angle of reflection) from the walls and bats. Just as the bats move in small steps, so does the ball. Its subsequent position depends on its present one, its direction, and (sometimes) on the position of the bats. Thus the state of the ball is taken to be a pair

(p, d) where p is the present position of the ball, p$\in$ *Court*, and d is the direction of the ball, d$\in$ R^2, $|d| = 1$.

For convenience we write $p = (p_x,\ p_y)$ and $d = (d_x,\ d_y)$. We are assuming that the speed of the ball is constant, that there is no friction, that there is no spin imparted to the ball by the bats, and so on.

Since the ball starts in a random nonvertical direction,

$$Ball_1 \mathrel{\widehat{=}} \sqcap_{dx\neq 0} Ball((1,\ 1/2)$$

where $\sqcap$ is a prefix form of internal choice.

- **The Ball's Movement**
  Away from the edges of the court the ball moves in the same direction, from the state (p, d) to state (p+d, d). It penetrates a goal unless deflected by a bat in which case the ball changes from state (p, d) to $((p_x - d_x,\ p_y + d_y),\ (-d_x,\ d_y))$. Similarly, at the

wall the ball changes from state (p, d) to state $((p_x + d_x,\ p_y - d_y),\ (d_x,\ d_y))$: this occurs whenever the ball lies in $Court^0$ but its subsequent position does not.

Thus, to define the ball's movement, we suppose that it is in state $Ball(p,\ d)$, with $p \in Court^0$, and consider the cases:

- if $p + d \in Court^0$, then the ball moves to state $(p + d,\ d)$;
- if $p + d$ lies on or over the wall, then the ball moves to state $((p_x + d_x,\ p_y - d_y),\ (d_x,\ -d_y))$;
- if $p + d$ lies on or over a bat, then the ball moves to state $((p_x - d_x,\ p_y + d_y),\ (-d_x,\ d_y))$;
- if $p + d$ lies on or past the left-hand goal not covered by the bat, then *lwin* occurs and the ball returns to the middle of the court;
- if $p + d$ lies on or past the right-hand goal not covered by the bat, then *rwin* occurs and the ball returns to the middle of the court.

We introduce a conditional construct

$$P \triangleleft B \triangleright Q$$

which is interpreted

if B **then** P **else** Q.

Thus

if $B1$ **then** $P1$ **if** $B2$ **then** $P2$ **else** $P3$

becomes

$$P1 \triangleleft B1 \triangleright (P2 \triangleleft B2 \triangleright P3).$$

or, in two-dimensional form

$$\begin{array}{c} P1 \\ \triangleleft B1 \triangleright \\ P2 \\ \triangleleft B2 \triangleright \\ P3 \quad . \end{array}$$

Note that the ternary combinator is not associative, i.e. the following law is not true

$$P1 \triangleleft B1 \triangleright (P2 \triangleleft B2 \triangleright P3) \neq (P1 \triangleleft B1 \triangleright P2) \triangleleft B2 \triangleright P3.$$

Now we can specify the ball's changes of state as follows:

$$\begin{array}{lcc} Ball(p,d) & \widehat{=} & (\qquad Ball(p+d,\ d) \\ & & \triangleleft \qquad p + d \in Court^0 \qquad \triangleright \\ & & Ball((p_x + d_x, P_y - d_y), (d_x, -d_y)) \\ & & \triangleleft \qquad p + d\ on/over\ Wall \qquad \triangleright \end{array}$$

$$\begin{array}{c} Ball((p_x - d_x, P_y + d_y),(-d_x, d_y)) \\ \lhd \quad p + d \ on/past \ a \ bat \quad \rhd \\ lwin \rightarrow Ball_1 \\ \lhd p + d \ on/past \ LG \ without \ bat \quad \rhd \\ rwin \rightarrow Ball_1 \quad). \end{array}$$

As the description of Ball is self-contained (there is no refree and Ball should be able to determine who wins), it should communicate with its environment (the bats) which determine whether p+d is on or past the position of a bat. Since, for example, LB determines which of the events *lpositin!j* occurs, Ball must be prepared for any of them. Ball thus offers an external choice between all events in the set

$$\{lposition!j \ : \ |j| \leq n\}$$

and its subsequent behaviour depends on which j is communicated. To express this, we introduce a complementary half of a communication event (of which *lposition!j* is the **send**, or output, half) to be the **receive** (or input) event

$$lposition?x$$

where x is instantiated to whatever value matches the complementary send event. Here ! stands for **output** and ? for **input**. They enable the resulting process to progress with a net communication from the outputting sub-process to the inputting process.

We modify Ball to read the position of the left-hand bat LB when the ball is in the right-hand goal RG, and symmetrically. First, we convert the state being output by LB

$$x \in \{-n, \ -n+1, \ \ldots, \ -1, \ 0, \ 1, \ \ldots, \ n-1, \ n\}$$

to the corresponding value

$$x' \in (x+n)/(2n) \ \in [0,1]$$

required by Ball. Then we have

$$\begin{array}{rcc} Ball(p, d) & \widehat{=} & (\quad Ball(p + d, \ d) \\ & & \lhd \quad p + d \in Court^0 \quad \rhd \\ & & Ball((p_x + d_x, P_y - d_y),(d_x, -d_y)) \\ & & \lhd \quad p + d \ on/over \ Wall \quad \rhd \\ & & Ball((p_x - d_x, P_y + d_y),(-d_x, d_y)) \\ & & \lhd \quad p + d \ on/past \ a \ bat \quad \rhd \\ & & LX \\ & & \lhd p + d \ on/past \ LG \ without \ bat \quad \rhd \\ & & RX \quad). \end{array}$$

where, with x' defined as above,

$$\begin{aligned}
LX &\;\widehat{=}\; lposition?x \rightarrow \; Ball((p_x - d_x, P_y + d_y), (-d_x, d_y)) \\
&\qquad\qquad \lhd \quad x' \; on/past \; p_y \quad \rhd \\
&\qquad\qquad lwin \rightarrow Ball_1 \qquad), \\
RX &\;\widehat{=}\; rposition?x \rightarrow \; Ball((p_x - d_x, P_y + d_y), (-d_x, d_y)) \\
&\qquad\qquad \lhd \quad x' \; on/past \; p_y \quad \rhd \\
&\qquad\qquad rwin \rightarrow Ball_1 \qquad),
\end{aligned}$$

Finally, we must record the alphabet of Ball

$$\alpha(Ball) \;\widehat{=}\; \alpha(Ball_1) \cup \{lposition?x : |x| \leq n\} \cup \{rposition?x : |x| \leq n\}.$$

This completes our description of Ball and of PingPong.

In this seminar an algebraic semantics has been given to a subset of CSP. It is convenient to use these algebraic laws in CSP program transformation and proofs of process equivalence. But there are still some questions remained.

- **Questions:**
  How do we know our description captures all the features embraced in the requirement? or in other words, How can we ensure users that all the CSP laws given above are consistent and complete?

- **solution:**
  Formalization of the semantics of CSP notations: a mathematical model and semantics.

3 Seminar Two : A Trace Model

In the previous seminar, we have stated a large number of laws which are used in the proofs and process transformation. These laws have not been justified. So there arises the questions: are these laws in fact true? are they even consistent? Should there be more of them? or are they complete in the sense that they permit all true facts about processes to be proved from them? Could one manage with fewer and simpler laws? These are questions for which an answer must be sought in a deeper mathematical investigation.

3.1 The Description of Processes

In constructing a mathematical model of a physical system, it is a good strategy to define the basic concepts in terms of attributes that can be directly or indirectly observed or measured. For a deterministic process, we are familiar with two such attributes:

- αP is the set of events in which the process is in priciple capable of engaging;

- $traces(P)$ is the set of all sequences of events in which the process can actually engage if required.

Definition 3.1 *(Deterministic Processes)*
A deterministic process is a pair

$$(A,\ S)$$

where A is any set of symbols and S is any subset of A^* which satisfies the two conditions

C0 $<>\in S$

C1 $\forall tr,\ tr1.\ tr^\wedge tr1 \in S \Rightarrow tr \in S$

Examples

E1 $\alpha(\mathbf{stop}_A)\ \widehat{=}\ A$
$traces(\mathbf{stop}_A)\ \widehat{=}\ \{<>\}$

E2 $\alpha(LP)\ \widehat{=}\ \{left,\ right,\ stay,\ lwin,\ rwin\}$
$traces(LP)\ \widehat{=}\ \{left,\ right,\ stay,\ lwin,\ rwin\}^*$

E3 $P\ =\ (a \to b \to \mathbf{stop}) \sqcap (b \to c \to \mathbf{stop})$
$\alpha P\ =\ \{a,\ b,\ c\}$
$traces(P)\ =\ \{<>,\ <a>,\ <ab>,\ <b>,\ <bc>\}$

3.2 A Denotational Semantics

The various operators on processes can now be formally defined by showing how the alphabet and traces of the result are derived from the alphabet and traces of the operands.

D1 $\mathbf{stop}_A\ =\ (A,\ \{<>\})$

D2 $\mathbf{run}_A\ =\ (A,\ A^*)$

D3 $((A,S) \square (A,T))\ =\ (A,\ S \cup T)$

D4 $(A,\ S)/tr\ =\ (A,\ \{tr1 \mid tr^\wedge tr1 \in S\})$ provided $tr \in S$

D5 $\mu X : A.F(X)\ =\ (A,\ \bigcup_{n \geq 0} traces(F^n(\mathbf{stop}_A)))$
provided F is a guarded expression

D6 $(A,S) \parallel (B,T)\ =\ (A \cup B,\ \{tr \mid tr \in (A \cup B)^* \wedge (tr \restriction A) \in S \wedge (tr \restriction B) \in T\})$

D7 $f(A,S)\ =\ (f(A),\ \{f^*(tr) \mid tr \in S\})$ provided f is one-one

As the above definitions are processes, we have to prove that they do satisfy the two conditions in the definition of a process. We prove, for instance, $(A,S) \parallel (B,T)$ is a process.

C0 because $<>\in S$ and $<>\in T$, we have $<>\in traces((A,S) \parallel (B,T))$ by its definition.

C1 suppose $tr^\wedge tr1 \in traces((A,S) \parallel (B,T))$. Then, by the definition, we have $(tr^\wedge tr1) \restriction A \in traces(S)$ and $(tr^\wedge tr1) \restriction B \in traces(T)$. Since the pairs of sets (A,S) and (B,T) satisfy **C1**, we have, by properties of $\restriction$, $tr \restriction A \in traces(S)$ and $tr \restriction B \in traces(T)$. Then by definition of $\parallel$, we have $tr \in traces((A,S) \parallel (B,T))$ as required.

Now we can prove all the laws given in the previous seminar are actually consistent with the new model.
Exercise: prove all the laws in the previous seminar.

3.3 Fixed-point Theory

The purpose of this subsection is to give an outline of a proof of the fundamental theorem of the recursion, that a recursively defined process is indeed a solution of the corresponding recursive equation, i.e.,

$$\mu X.F(X) \;=\; F(\mu X.F(X))$$

The treatment follows the fixed-point theory of Scott.

First, we define an ordering relationship $\sqsubseteq$ among processes

D1 $(A,S) \sqsubseteq (B,T) \;=\; (A = B \wedge S \subseteq T)$

Two processes are comparable in this ordering if they have the same alphabet and one of them can do everything done by the other – and maybe more. This ordering is a partial order in the sense that

L1 $P \sqsubseteq P$

L2 $P \sqsubseteq Q \wedge Q \sqsubseteq P \Rightarrow P = Q$

L3 $P \sqsubseteq Q \wedge Q \sqsubseteq R \Rightarrow P \sqsubseteq R$

Definition 3.2 *(Basic concepts)*

- *A* chain *in a partial order is an infinite sequence of elements*

  $$\{P_0, P_1, P_2, \ldots\}$$

  such that

  $$P_i \sqsubseteq P_{i+1} \qquad \textit{for all } i$$

- *The* limit *(least upper bound) of such a chain is defined*

  $$\bigsqcup_{i \geq 0} P_i \;=\; (\alpha P_0,\; \bigcup_{i \geq 0} traces(P_i))$$

- *A partial order is said to be* complete *(c.p.o.) if it has a least element, and all the chains have a least upper bound.*

- *A function F from one c.p.o. to another is said to be* continuous *if it distributes over the limits of all chains, i.e.,*

  $$F(\bigsqcup_{i \geq 0} P_i) \;=\; \bigsqcup_{i \geq 0} F(P_i) \qquad \textit{if } \{P_i \mid i \geq 0\} \textit{ is a chain}$$

  A function G of several arguments is defined as continuous if it is continuous in each of its arguments separately.

Theorem 3.1 *The set of all processes with a given alphabet A form a complete partial order.*

Exercise: prove the following three laws.

L4 $\mathbf{stop}_A \sqsubseteq P$ provided $\alpha P = A$

L5 $P_i \sqsubseteq \bigsqcup_{i \geq 0} P_i$

L6 $(\forall i \geq 0.\, P_i \sqsubseteq Q) \Rightarrow (\bigsqcup_{i \geq 0} P_i) \sqsubseteq Q$

We can reformulate the definition of μ in terms of a limit

L7 $\mu X : A.F(X) \;=\; \bigsqcup_{i \geq 0} F^i(\mathbf{stop}_A)$

Theorem 3.2 *All the operators (except /) defined in D3 to D7 are continuous.*

Exercise: prove the following laws

L8 $(x : B \rightarrow (\bigsqcup_{i \geq 0} P_i(x))) \;=\; \bigsqcup_{i \geq 0}(x : B \rightarrow P_i(x))$

L9 $\mu X : A.F(X, (\bigsqcup_{i \geq 0} P_i)) \;=\; \bigsqcup_{i \geq 0} \mu X : A.F^i(X, P_i)$ provided F is continuous

L10 $(\bigsqcup_{i \geq 0} P_i) \parallel Q \;=\; Q \parallel (\bigsqcup_{i \geq 0} P_i) \;=\; \bigsqcup_{i \geq 0}(P_i) \parallel Q)$

L11 $f(\bigsqcup_{i \geq 0} P_i) \;=\; \bigsqcup_{i \geq 0} f(P_i)$

Consequently, if $F(X)$ is an expression solely constructed in terms of these operators, it will be continuous in X.

Now it is possible to prove

Theorem 3.3 $\mu X : A.F(X) \;=\; \bigsqcup_{i \geq 0} F^i(\mathbf{stop}_A)$ *is a solution to* $F(X) = X$ *if* F *is continuous, i.e.,*

$$F(\mu X : A.F(X)) \;=\; \mu X : A.F(X)$$

Proof:

$$\begin{aligned}
F(\mu X : A.F(X)) &= F(\bigsqcup_{i \geq 0} F^i(\mathbf{stop}_A)) && \text{def of } \mu \\
&= \bigsqcup_{i \geq 0} F(F^i(\mathbf{stop}_A)) && \text{continuity of } F \\
&= \bigsqcup_{i \geq 1} F^i(\mathbf{stop}_A) && \text{def of } F^{i+1} \\
&= \bigsqcup_{i \geq 0} F^i(\mathbf{stop}_A) && \mathbf{stop}_A \sqsubseteq F(\mathbf{stop}_A) \\
&= \mu X : A.F(X) && \text{def of } F^{i+1}
\end{aligned}$$

This completes the proof.

3.4 Unique Solution

In this subsection we will prove that an equation defining a process by guarded recursion has only one solution.

Definition 3.3 *If P is a process and n is a natural number, we define $(P \upharpoonright n)$ as a process which behaves like P for its first n events, and then stops*

$$(A, S) \upharpoonright n \;=\; (A, \{tr \mid tr \in S \wedge \#tr \leq n\}).$$

It follows that

L12 $P \upharpoonright 0 \;=\; \mathbf{stop}$

L13 $P \upharpoonright n \sqsubseteq P \upharpoonright (n+1) \sqsubseteq P$

L14 $P \;=\; \bigsqcup_{n \geq 0} P \upharpoonright n$

L15 $\bigsqcup_{n \geq 0} P_n \;=\; \bigsqcup_{n \geq 0}(P_n \upharpoonright n)$

Definition 3.4 *Let F be a monotonic function from processes to processes. F is said to be constructive if*

$$F(X) \upharpoonright (n+1) \;=\; F(X \upharpoonright n) \upharpoonright (n+1) \qquad \textit{for all } X$$

Examples

E4 Prefixing is a constructive function

$$(c \rightarrow P) \upharpoonright (n+1) \;=\; (c \rightarrow (P \upharpoonright n)) \upharpoonright (n+1)$$

E5 General choice is also constructive

$$(x : B \rightarrow P(x)) \upharpoonright (n+1) \;=\; (x : B \rightarrow (P(x) \upharpoonright n)) \upharpoonright (n+1)$$

E6 The identity function I is not constructive

$$\begin{aligned} I(c \rightarrow P) \upharpoonright 1 &= c \rightarrow \mathbf{stop} \\ &\neq \mathbf{stop} \\ &= I((c \rightarrow P) \upharpoonright 0) \upharpoonright 1 \end{aligned}$$

Theorem 3.4 *Let F be a constructive function. The equation*

$$X = F(X)$$

has only one solution for X.

Proof:

Let X be an arbitrary solution.

First by induction we prove the lemma that

$$X \upharpoonright n \;=\; F^n(\mathbf{stop}) \upharpoonright n$$

Base case. $X \upharpoonright 0 = \mathbf{stop} = \mathbf{stop} \upharpoonright 0 = F^0(\mathbf{stop}) \upharpoonright o.$

Induction step.

$$\begin{array}{rcll}
X \upharpoonright (n+1) & = & F(X) \upharpoonright (n+1) & \text{since } X = F(X) \\
& = & F(X \upharpoonright n) \upharpoonright (n+1) & F \text{ is constructive} \\
& = & F(F^n(\mathbf{stop}) \upharpoonright n) \upharpoonright (n+1) & \text{hypothesis} \\
& = & F(F^n(\mathbf{stop})) \upharpoonright (n+1) & F \text{ is constructive} \\
& = & F^{n+1}(\mathbf{stop})) \upharpoonright (n+1) & \text{def of } F
\end{array}$$

Now we go back to the main theorem.

$$\begin{array}{rcll}
X & = & \bigsqcup_{n \geq 0} (X \upharpoonright n) & \text{L14} \\
& = & \bigsqcup_{n \geq 0} F^n(\mathbf{stop}) \upharpoonright n & \text{just proved} \\
& = & \bigsqcup_{n \geq 0} F^n(\mathbf{stop}) \qquad ; & \text{L15} \\
& = & \mu X.F(X) & \text{L7}
\end{array}$$

Thus all solutions of $X = F(X)$ are equal to $\mu X.F(X)$; or in other words, $\mu X.F(X)$ is the only solution of the equation.

The usefulness of this theorem is much increased if we can clearly recognize which functions are constructive and which are not.

The constructiveness can be defined syntactically by the following conditions for guardedness.

D0 An expression constructed solely by means of the operators concurrency, symbol change, and general choice are said to be guard-preserving.

D1 An expression which does not contain X is said to be guarded in X.

D2 A general choice

$$(x : B \rightarrow P(X, x))$$

is guarded in X if $P(X, x)$ is guard-preserving for all x.

D3 A symbol change $f(P(X))$ is guarded in X if $P(X)$ is guarded in X.

D4 A concurrent system $P(X) \parallel Q(X)$ is guarded in X if both $P(X)$ and $Q(X)$ are guarded in X.

Finally, we have

Theorem 3.5 *If E is guarded in X, then the equation*

$$X = F(X)$$

has an unique solution.

3.5 Specifications

A specification of a product is a description of the way it is intended to behave. This description is a predicate containing free variables, each of which stands for some observable aspect of the behaviour of the product.

In our case, a process P with alphabet αP has only one observable aspect: its traces. Therefore, the specification for a process is a predicate containing the trace variable tr

$$P \ \widehat{=}\ (A, S) \Longleftrightarrow S_A(tr).$$

Definition 3.5 *(Specification)*
A specification of a process P with alphabet A, is of the form

$$Sp_A(tr) \ = \ \alpha(tr) \subseteq A \ \wedge \ H_T(tr)$$

where tr occurs in H_T only in the form

$$tr \upharpoonright B \qquad \textit{with } B \subseteq A$$

and $\alpha(tr)$ is the set of events occurred in trace tr.

Examples

E1 The specification of process

$$P \ = \ (a \rightarrow b \rightarrow \mathbf{stop}) \sqcap (b \rightarrow a \rightarrow \mathbf{stop})$$

is defined

$$\begin{aligned} Sp_{\{a,b\}}(tr) \ &= \ \alpha(tr) \subseteq \{a, b\} \wedge (tr =<> \vee tr =< a > \vee tr =< ab > \\ &\qquad \vee tr =< b > \vee tr =< ba >), \end{aligned}$$

where $tr \upharpoonright \alpha$ is written as tr and sometimes $tr \upharpoonright \{c\}$ is written c.

E2 The above specification can also be written as

$$Sp_{\{a,b\}} \ = \ \alpha tr \subseteq \{a, b\} \wedge (tr \leq < ab > \vee tr \leq < ba >)$$

or

$$Sp_{\{a,b\}} \ = \ \alpha tr \subseteq \{a, b\} \wedge (\#a \leq 1 \vee \#b \leq 1).$$

E3 A one-place buffer can be specified

$$Buff_{\{in,out\}} \ = \ \alpha(tr) \subseteq \{in, out\} \wedge (0 \leq \#(tr \upharpoonright \{in\}) - \#(tr \upharpoonright \{out\}) \leq 1).$$

Definition 3.6 *(Satisfaction Relation)*
A process $P \ = \ (A, S)$ is said to satisfy a specification Sp_B, denoted

$$P \ \mathbf{sat} \ Sp$$

if and only if

$$A = B \qquad \textit{and} \qquad \forall tr.\ tr \in traces(P) \Rightarrow Sp(tr)$$

Then we can prove the following proof rules for CSP operators in our mathematical model (as exercises).

L16 P **sat true**

L17 If P **sat** Sp and P **sat** Tp, then

$$P \textbf{ sat } (Sp \wedge Tp).$$

L18 If $\forall n. (P$ **sat** $Sp(n)$ then

$$P \textbf{ sat } \forall n. Sp(n).$$

provided P does not depend on n.

L19 If P **sat** Sp and $Sp \Rightarrow Tp$, then

$$P \textbf{ sat } Tp.$$

L20 **stop sat** $(tr =<>)$

L21 If P **sat** Sp then

$$(c \rightarrow P) \textbf{ sat } (tr =<> \vee (tr_0 = c \wedge Sp(tr')).$$

L22 If P **sat** Sp and Q **sat** Tp, then

$$(c \rightarrow P) \mid (d \rightarrow Q) \textbf{ sat } tr =<> \vee (tr_0 = c \wedge Sp(tr')) \vee (tr_0 = d \wedge Tp(tr')).$$

L23 If P **sat** Sp and Q **sat** Tp, then

$$(P \parallel Q) \textbf{ sat } Sp(tr \restriction \alpha P) \wedge Tp(tr \restriction \alpha Q).$$

L24 If P **sat** $Sp(tr)$ and $tr \in traces(P)$, then

$$P/tr \textbf{ sat } Sp(tr^\wedge tr1)$$

L25 If X is guarded and **stop sat** $Sp(tr)$ and $(X \textbf{ sat } S) \Rightarrow (F(X) \textbf{ sat } S)$, then

$$\mu X.F(X) \textbf{ sat } Sp$$

Examples

E1 For any process specification Sp_A, we have

$$\textbf{stop}_A \textbf{ sat } Sp_A$$

E2 $(a \rightarrow \textbf{stop})$ $\textbf{sat}(a \rightarrow b \rightarrow \textbf{stop})$

From E2 we can see that the satisfaction relation is not good enough because we are not able to specify something which will eventually happen.

We will solve this problem in next seminar.

4 Seminar Three : The Failures Model

In the previous section we have already seen the problems with the trace model: it can not express liveness properties, and neither can it distinguish between interal and external choice.

Nondeterminism is usefull for maintaining a high level of abstraction in the description of the behaviour of physical systems and machines. For example, the combination of changes given by a change machine may depend on the way in which the machine has been loaded; but we have excluded these events from the alphabet.

4.1 Deterministic Processes

We have already seen the choice operator

$$x : B \rightarrow P(x)$$

which exhibits a range of possible behaviours.

The concurrency operator $\|$ permits some other process to make a selection between the alternatives offered in the set B. For example, a customer is usually able to choose Coke or Diet Coke on a Coke machine after having inserted enough coins.

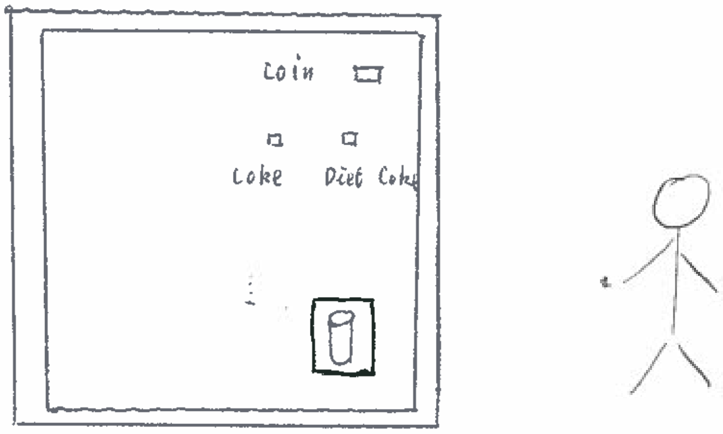
Example 4.1

Suppose we have a Coke machine,

$$CokeM \;\;\widehat{=}\;\; coin \rightarrow (coke \rightarrow CokeM \Box diet \rightarrow CokeM),$$

and a customer who takes cokes every time

$$Customer \;\;\widehat{=}\;\; coin \rightarrow coke \rightarrow \textbf{stop}.$$



Coke Machine

In this case, the customer is the environment of the coke machine. Operating in such an environment, the resulting system behaves as follows

$$(Customer \parallel CokeM) \;=\; coin \rightarrow coke \rightarrow \textbf{stop}.$$

The choice between *coke* and *diet* coke can be controlled by the customer. We call such processes (*CokeM*) deterministic processes.

Definition 4.1 *(Deterministic Processes)*
A process is a deterministic process if whenever there is more than one event possible, the choice between them is determined by the environment of the process.

4.2 Nondeterministic Processes

The informal description of a nondeterministic process is: it has a range of possible behaviours, but its environment does not have the ability to influence or even observe the selection between these alternatives.

Example 4.2
The following change machine *CoinM* is a nondeterministic process. It gives nondeterministically one of the two coin-combinations of a one-dollar bill.

$$\begin{aligned} CoinM \;=\; \$1 \;\rightarrow\; & (q \rightarrow q \rightarrow q \rightarrow d \rightarrow d \rightarrow n \rightarrow CoinM \\ & \sqcap q \rightarrow q \rightarrow d \rightarrow d \rightarrow d \rightarrow n \rightarrow n \rightarrow n \rightarrow n \rightarrow CoinM) \end{aligned}$$

If a customer *Customer* insists on having the first combination of coins, he may or may never get it from *CoinM*. It depends on his "luck".

$$Customer \;=\; \$1 \rightarrow q \rightarrow q \rightarrow q \rightarrow d \rightarrow d \rightarrow n \rightarrow \mathbf{stop}$$

$$\begin{aligned} CoinM \parallel Customer \;=\; \$1 \rightarrow q \rightarrow q \rightarrow \quad & (\; q \rightarrow d \rightarrow d \rightarrow n \rightarrow \mathbf{stop} \\ & \;\sqcap \mathbf{stop}) \end{aligned}$$

□

4.3 The Difference Between ⊓ and □

The difference between nondeterministic choice and deterministic choice is very subtle. The only way to distinguish between them is to put them in an environment where the nondeterministic process may lead to a deadlock, but the deterministic does not.

Example 4.3
Let

$$P \;=\; (x \rightarrow P), \qquad Q \;=\; (y \rightarrow Q)\text{and} \qquad \alpha P = \alpha Q = \{x, y\}$$

Then

$$\begin{aligned} (P \Box Q) \parallel P &= (x \rightarrow P) \\ &= P \\ (P \sqcap Q) \parallel P &= (P \parallel P) \sqcap (Q \parallel P) \\ &= P \sqcap \mathbf{stop} \end{aligned}$$

In environment P, $(P \sqcap Q)$ may reach deadlock, but $(P \Box Q)$ cannot. Of course, even with $(P \Box Q)$ we cannot be sure that deadlock will occur; and if it does not occur, we will never know that it might have. But the mere possibility of an occurrence of deadlock $(P \sqcap Q)$ at its first step in such an environment is enough to distinguish it from $(P \Box Q)$.
□

4.4 Refusals

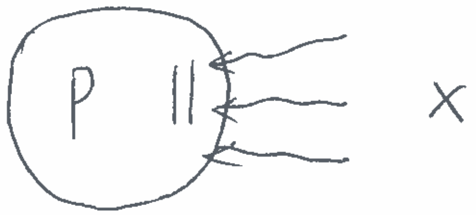
In general, let X be a set of events which are offered initially by the environment of a process P, which in this context we take to have the same alphabet as P. If it is possible for P to deadlock on its first step when placed in this environment, we say that X is a *refusal* of P. The set of all such refusals of P is denoted

$$\mathit{refusals}(P)$$

Note that the refusals of a process constitute a family of sets of symbols.

Example 4.4

Let $P = (x \rightarrow \textbf{stop} \sqcap y \rightarrow \textbf{stop})$. We put P in an environment which offers the events in X.



Then the refusals of P are:

$$\begin{aligned} X &= \{x, y\} \\ X &= \{x\} \\ X &= \{y\} \\ X &= \phi \end{aligned}$$

□

The introduction of the concept of a refusal permits a clear formal distinction to be made between deterministic and nondeterministic processes.

$$P \text{ is deterministic } \equiv \forall\, tr : \mathit{traces}(P).\ (X \in \mathit{refusals}(P/tr) \equiv X \cap P^0 = \{\})$$

where $P^0 = \{x \mid < x > \in \mathit{traces}(P)\}$.

In other words a set is a refusal of a deterministic process after tr only if that set contains no event in which that process can engage after tr.

A nondeterministic process is one that does not enjoy this property, i.e., there is at some time some event in which it can engage; but also (as a result of some internal nondeterministic choice) it may refuse to engage in that event, even though the environment is ready for it.

The laws about refusals are

L1 $\mathit{refusals}(\textbf{stop}_A)$ = all subsets of A (including A itself)

L2 $\mathit{refusals}(c \rightarrow P) = \{X \mid X \subseteq (\alpha P - \{c\})\}$

L3 $\mathit{refusals}(x : B \rightarrow P(x)) = \{X \mid X \subseteq (\alpha P - B)\}$

L4 $\mathit{refusals}(P \sqcap Q) \;=\; \mathit{refusals}(P) \cup \mathit{refusals}(Q)$

L5 $\mathit{refusals}(P \Box Q) \;=\; \mathit{refusals}(P) \cap \mathit{refusals}(Q)$

L6 $\mathit{refusals}(P \parallel Q) \;=\; \{X \cup Y \mid X \in \mathit{refusals}(P) \wedge Y \in \mathit{refusals}(Q)\}$

L7 $\mathit{refusals}(f(P)) \;=\; \{f(X) \mid X \in \mathit{refusals}(P)\}$

A process can refuse only events in its own alphabet. A process deadlocks when the environment offers no events; and if a process refuses a nonempty set, it can refuse any subset of it. Finally, any event x which cannot occur initially may be added to any set X already refused. Therefore, we have

L8 $X \in \mathit{refusals}(P) \Rightarrow X \subseteq \alpha P$

L9 $\{\} \in \mathit{refusals}(P)$

L10 $(X \cup Y) \in \mathit{refusals}(P) \Rightarrow X \in \mathit{refusals}(P)$

L11 $X \in \mathit{refusals}(P) \Rightarrow (X \cup \{x\}) \in \mathit{refusals}(P) \vee <x> \in \mathit{traces}(P)$

4.5 Failures

We are going to develop a new mathematical model for CSP, which is based on the new observable aspects of a process, i.e., its alphabet, its traces and refusals.

In addition to refusals at the first step of a process P, it is also necessary to take into account what P may refuse after having been engaged in an arbitrary trace tr of its behaviour. Therefore we introduce another concept, *failures* of a process.

Definition 4.2 *(Failures)*
The failures of a process is a set of pairs

$$\mathit{failures}(P) \;=\; \{(tr, X) \mid tr \in \mathit{traces}(P) \wedge X \in \mathit{refusals}(P/tr)\}.$$

If (tr, X) is a failure of P, it means that P can engage in the sequence of events recorded in tr, and then refuse to engage in any of the events in X, in spite of the fact that its environment is prepared to engage in them.

We can define traces and refusals of a process in terms of failures

$$\begin{aligned} \mathit{traces}(P) &= \{tr \mid \exists X.\,(tr, X) \in \mathit{failures}(P)\} \\ \mathit{refusals}(P) &= \{X \mid (<>, X) \in \mathit{failures}(P)\} \end{aligned}$$

Now we are able to define a process in terms of failures.

Definition 4.3 *(Processes)*
A process is a pair

$$(A, F)$$

where A is any set of symbols (finite) and F is a relation between A^* and $\mathcal{P}(A)$, provided they satisfy

F0 $(<>, \{\}) \in F$

F1 $(tr^\wedge tr_1, X) \in F \Rightarrow (tr, X) \in F$

F2 $(tr, Y) \in F \wedge X \subseteq Y \Rightarrow (tr, X) \in F$

F3 $(tr, X) \in F \wedge x \in A \Rightarrow (tr, X \cup \{x\}) \in F \vee (tr^\wedge < x >, \{\}) \in F$

The refinement ordering $\sqsubseteq$ is defined

$$(A, F_1) \sqsubseteq (B, F_2) \;\equiv\; F_2 \subseteq F_1 \;\wedge\; A = B.$$

$P \sqsubseteq Q$ means that Q is equal to P or better in the sense that it is less likely to fail. Q is more predictabl and more controllable than P because if Q can do something, P can do it too; if Q can refuse to do something, P can also refuse. The least element of all the processes with alphabet A under $\sqsubseteq$ is

$$\mathbf{chao}_A \;=\; (A,\; (A^* \times \mathcal{P}(A))).$$

In fact, this ordering is a complete partial order, with a limit operation defined in terms of the intersections of descending chains of failures

$$\bigsqcup_{n \geq 0} (A, F_n) \;=\; (A,\; \bigcap_{n \geq 0} F_n),$$

provided that $\forall\, n \geq 0.\; F_{n+1} \sqsubseteq F_n$.

4.6 A Failures Semantics

Now we give a denotational semantics to CSP in terms of failures.

D1 $\mathbf{stop}_A \;=\; (A, \{<>\} \times \mathcal{P}(A))$

D2 $(A, F_1) \sqcap (A, F_2) \;=\; (A, F_1 \cup F_2)$

The definitions of all the other operators can be given similarly; but it seems slightly elegant to write separate definitions for the alphabets and failures.

D3 If $\alpha P(x) = A$ for all x and $B \subseteq A$, then

$$\alpha(x : B \rightarrow P(x)) = A$$

D4 $\alpha(P \parallel Q) = (\alpha P \cup \alpha Q)$

D5 $\alpha(f(P)) = f(\alpha P))$

D6 $\alpha(P \Box Q) = \alpha P = \alpha Q$

D7

$$\begin{aligned} \mathit{failures}(x : B \rightarrow P(x)) \;&=\; \{(<>, X) \mid X \subseteq (\alpha P - B)\} \\ &\quad \cup \{(< x >^\wedge tr, X) \mid x \in B \wedge (tr, X) \in \mathit{failures}(P(x))\} \end{aligned}$$

D8

$$failures(P \parallel Q) = \{(tr, X \cup Y) \mid tr \in (\alpha P \cup \alpha Q)^* \wedge (tr \restriction \alpha P, X) \in failures(P) \wedge (tr \restriction \alpha Q, X) \in failures(Q)\}$$

D9 $failures(f(P)) = \{(f^*(tr), f(X)) \mid (tr, X) \in failures(P)\}$

D10

$$failures(P \Box Q) = \{(tr, X) \mid (tr, X) \in (failures(P) \cap failures(Q)) \vee (tr \neq <> \wedge (tr, X) \in (failures(P) \cup failures(Q)))\}$$

D11 $\mu X : A.F(X) = \bigsqcup_{n \geq 0} F_n(\mathbf{chao}_A)$

We claim that all the above operators are well-defined and continuous.

4.7 Specifications and Proof Rules

1. Specifications

In order to be able to capture the liveness properties of a process, we introduced the concept, *failures*. Then, in the specification of a process, there are two free variables

(a) the trace variable tr and

(b) the refusal variable ref.

Therefore, the specification of a process can be written

$$Sp(tr, ref)$$

This also changes the satisfaction relationship between processes and specifications.

Definition 4.4 *(Satisfaction Relation)*
A process P is said to satisfy a specification Sp, denoted

$$P \ \mathbf{sat} \ Sp(tr, ref)$$

if

$$\forall tr, ref.\ tr \in traces(P) \wedge ref \subseteq refusals(P/tr) \Rightarrow Sp(tr, ref)$$

or

$$\forall tr, ref.\ (tr, ref) \in failures(P) \Rightarrow Sp(tr, ref).$$

Examples

E1 $\mathbf{stop}_A$ **sat** $(tr = <> \wedge ref \subseteq A)$

E2 $P_2 = a \rightarrow \mathbf{stop}$ can be specified by

$$\begin{aligned} Sp(re, ref) &= \alpha tr \subseteq \{a, b\} \\ &\quad (\ tr = <> \wedge ref \subseteq \{b\} \\ &\quad \vee\ tr = < a > \wedge ref \subseteq \{a, b\}) \end{aligned}$$

E3 $P_1 = a \rightarrow b \rightarrow \mathbf{stop}$ can be specified by

$$\begin{aligned} Sp(re, ref) &= \alpha tr \subseteq \{a, b\} \\ &\quad (\ tr = <> \wedge ref \subseteq \{b\} \\ &\quad \vee\ tr = < a > \wedge ref \subseteq \{a\} \\ &\quad \vee\ tr = < b > \wedge ref \subseteq \{a, b\}) \end{aligned}$$

- The difference between **sat** in the trace model and **sat** in the failures model

  (a) in trace model, the behaviour of a process is described in terms of traces. Only safety properties can be captured. Therefore we have

  $$\mathbf{stop}_A \ \ \mathbf{sat} \ \ P_A$$

  and

  $$P_2 \ \ \mathbf{sat} \ \ P_3$$

  (b) in the failures model, the behaviour of a process is described in terms of traces as well as refusals. Certain liveness properties are captured in this model. The above two relation no longer holds in failures model.

2. Proof Rules

In the following proof rules, a specification will be written ia any of the forms S, $S(tr)$, $S(tr, ref)$, according to convenience. In all cases, it should be understood that the specification may contain tr and ref among its free variables.

L1 If P **sat** Sp and Q **sat** Tp, then

$$(P \sqcap Q) \ \ \mathbf{sat} \ \ (Sp \vee Tp)$$

Proof: Hypothesis:

$$\forall tr, ref.\ (tr, ref) \in failures(P) \Rightarrow Sp(tr, ref)$$

$$\forall tr, ref.\ (tr, ref) \in failures(Q) \Rightarrow Tp(tr, ref)$$

We have

$$\forall tr, ref.\ (tr, ref) \in (failures(P) \cup failures(Q)) \Rightarrow (Sp(tr, ref) \vee Tp(tr, ref))$$

hold.

L2 $\mathbf{stop}_A$ **sat** $(tr = <> \wedge ref \subseteq A)$

L3 If P **sat** $Sp(tr)$, then

$$(c \to P) \ \mathbf{sat} \ (tr = <> \wedge c \notin ref) \vee (tr_0 = c \wedge Sp(tr'))$$

L4 If $\forall x \in B.\, P(x)$ **sat** $Sp(tr, x)$, then

$$\begin{aligned}(x : B \to P(x)) \ \ \mathbf{sat} \ \ & (tr = <> \wedge (B \cap ref = \{\}) \\ & \vee (tr_0 \in B \wedge Sp(tr', tr_0))\end{aligned}$$

L5 If P **sat** Sp and Q **sat** Tp, then

$$(P \parallel Q) \ \mathbf{sat} \ \exists X, Y.\, ref = X \cup Y \wedge Sp(tr \upharpoonright \alpha P, X) \wedge Tp(tr \upharpoonright \alpha Q, Y)$$

L6 If P **sat** Sp and Q **sat** Tp, then

$$(P \Box Q) \ \mathbf{sat} \ (Sp \wedge Tp) \lhd tr = <> \rhd (Sp \vee Tp)$$

L7 If P **sat** $Sp(tr, ref)$, then

$$f(P) \ \mathbf{sat} \ Sp(f^{-1}(tr), f^{-1}(ref)) \qquad f \text{ is one-one}$$

L8 If $(Sp(0)$ and $(X$ **sat** $Sp(n)) \Rightarrow (F(X)$ **sat** $Sp(n+1)$, then

$$\mu X.F(X) \ \mathbf{sat} \ (\forall n.\, Sp(n)$$

This completes the proof rules.

5 Extensions

1. Models

In the previous seminars two mathematical models are developed for a subset of CSP. Based on these models, two specification methods are developed also. As indicated in Section 4, the failures model and its specification method are more sophisticated than the trace model and its specification method. They contain more information about the behaviour of a process and the specification method can handle both safety and liveness properties. But even in the failures model, we have no way to distinguish between **stop** and a process having only the empty trace but which perpetually undergos unobservable internal events. For this we must use some more sophisticated models, like CSP's divergence model. Details of these models can be found in Hoare's CSP book [1].

2. Development method

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

The purpose of the study of semantic models for parallel languages is to achieve hierarchical and modular development and verification methods. By hierarchical and modular development and verification, is meant that the specifications and implementation of the subsystems of a concurrent system can be deduced from its original specification of the system (at a more abstract level), and that the correctness of the system with respect to its original specification can be obtained from that of its subsystems.

Let SP_0 be a specification of the requirements which a software system is expected to fulfill, expressed in some formal specification language SL. This specification SP_0 is obtained from the informal and often vague requirement of the customer. The ultimate goal is a program P written in some given programming language PL which satisfies the requirement in SP_0.

The usual way to proceed is to construct P by whatever means are available, making informal references to SP_0 in the process, and then verify in some way that P does satisfy SP_0. The only practical verification method available at the present is to test P, checking that in certain selected cases the behaviour of the system satisfies the constraints imposed by SP_0 . But this method has the obvious disadvantage that the correctness of P is never guaranteed, even if the system has passed all the test cases.

An alternative to testing is a formal proof that the program P is correct with respect to the specification SP_0. However after two decades of work on program verification, it seems now, more or less widely accepted, that this will probably never be feasible for programs of real sizes. At the least, the initial hopes for a system capable of automatically producing proofs of program correctness are now regarded as unrealistic.

Most recent work in this area has focused on methods for developing programs from specifications in such a way that the resulting program is guaranteed to be correct to the specification by construction.

Let SP_0 be the specification. The final program P is expected to be developed from SP_0 through a series of small refinement steps, as depicted in the following picture,

$$SP_0 \sqsubseteq SP_1 \sqsubseteq SP_2 \sqsubseteq \cdots \sqsubseteq SP_n \sqsubseteq P.$$

Each refinement step captures a single design decision. If each refinement step $SP_i \sqsubseteq SP_{i+1}$ can be proved correct, then P itself is guaranteed to be correct with respect to the initial specification SP_0 by transitivity of the refinement ordering $\sqsubseteq$. Each of these refinement steps should be much easier than an overall proof that P is correct with respect to SP_0 because these refinement steps can be made very small. In principle, it would be possible to combine all the proofs of these refinement steps to obtain an overall proof. But, in practice, this will never be necessary.

When this approach is used to develop large and complex programs, the individual specification SP_i becomes large and unwieldy. As a consequence, the proof of the correctness of each refinement step becomes difficult. The solution to this problem is to adapt the "divide and conquer" strategy to allow specifications to be decomposed into smaller units during the development process. These smaller specifications then may be refined independently of one another.

A simple development involving two decompositions and six refinement steps would then give the following diagram

$$SP_0 \sqsubseteq \begin{cases} SP_1 \sqsubseteq SP_2 \sqsubseteq P_1 \\ \oplus \\ SP_1' \sqsubseteq \begin{cases} SP_2' \sqsubseteq P_2 \\ \otimes \\ SP_2'' \sqsubseteq P_3. \end{cases} \end{cases}$$

Here $\oplus$ and $\otimes$ are intended to denote arbitary specification-building constructs, and P_1, P_2 and P_3 are program modules. The program $P_1 \oplus (P_2 \otimes P_3)$ is guaranteed to be correct with respect to SP_0, provided that each of the individual refinement steps is correct. This also assumes that $\oplus$ and $\otimes$ can be used for combining program modules as well as specifications.

Evidently formal program development methods do not claim to remove the possibility of unwise design decisions.

There exist several refinement calculi for sequential languages, inspired by Dijkstra's "Weakest-Precondition Calculus" . Among them, is Carroll Morgan's "Specification Statement Calculus", which is mathematically elegant and, when combined with Z, could be used to handle program systems of real sizes.

For the development of parallel programs, the idea of stepwise refinement is the same, that is , we hope to start with some specification SP_0 and, through small refinement steps, to obtain the final executable program code P,

$$SP_0 \sqsubseteq SP_1 \sqsubseteq SP_2 \sqsubseteq \cdots \sqsubseteq SP_n \sqsubseteq P.$$

But the main difference is the specification language in use and the refinement ordering on the specification space. This means that the underlying mathematical model is completely different.

First of all, parallel programs can not be modelled only by binary relations. For instance, nonterminating programs can not be modelled by the empty relation. So the techniques used in sequential program development and verification do not seem to work for parallel programs. Therefore we have to look for new mathematical models and new techniques.

The mathematical model for the "purely parallel" CSP (i.e. communicating processes without internal machine states) is the failure model. This model can describe the communication behaviour of a process. It captures the safety properties of a process by means of its communication traces *tr* and the liveness properties by means of its failures (*tr*, *ref*).

To model a CSP-like language with machine states, we need trace-indexed binary relations to describe the internal state changes. For instance, a *tr*-indexed relation R_{tr} means that, before the communications *tr* starts, the machine state is in the domain of R and, after the process has done *tr*, the final machine state is in the range of R.

Based on this kind of mathematical models, new refinement methods can be developed which may be able to handle both sequential and parallel program development.

This task remains as future work.

Acknowlegement

I would like to thank Dr. Roman and Dr. Cox for helping organize the seminar.

• References

1. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, London. 1985.
2. E.-R. Olderog and C.A.R. Hoare. *Specification-Oriented Semantics for Communicating Processes.* Acta Informatica 23, 9-66. 1986.
3. J. Sanders *An Introduction to CSP.* Technical Monograph PRG-65, Computing Lab, Oxford University, March 1988.