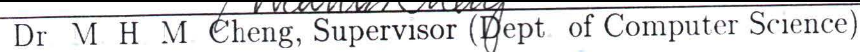
A TOOL FOR PROTOTYPING CONCURRENT DESIGN SPECIFICATIONS

by

Philip John Wiebe
Bachelor of Science, University of Victoria, 1992

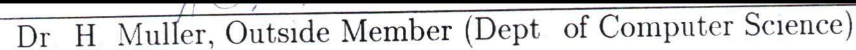
A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in the
Department of Computer Science

We accept this thesis as conforming
to the required standard

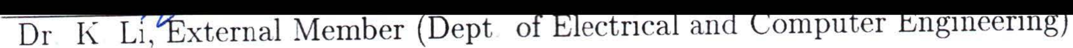


Dr M H M Cheng, Supervisor (Dept of Computer Science)

Dr G Shoja, Departmental Member (Dept of Computer Science)



Dr H Muller, Outside Member (Dept of Computer Science)



Dr K Li, External Member (Dept of Electrical and Computer Engineering)

© PHILIP JOHN WIEBE, 1996

University of Victoria

All rights reserved This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author

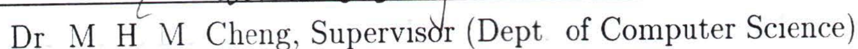
Supervisor: Dr. M. H. M. Cheng

Abstract

Process algebra is one formalism which aids in the design and verification of complex concurrent systems by using algebraic expressions to describe the architectural and behavioral aspects.

We introduce ACS (Algebra of Communicating Systems), a process algebraic specification language which combines the simple syntax of CCS (Calculus of Communicating Systems) and the sequential composition of ACP (Algebra of Communicating Processes), adding the notion of data and value-passing, and user-definable operators. To aid in the simulation and debugging of ACS designs, a concurrent path expression language (CPE) is introduced. A Prolog implementation of ACS and CPE is presented.

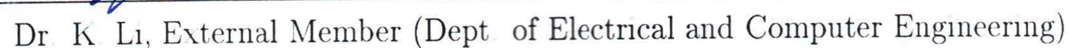
Examiners:



Dr. M. H. M. Cheng, Supervisor (Dept. of Computer Science)

Dr. G. Shoja, Departmental Member (Dept. of Computer Science)

Dr. H. Muller, Outside Member (Dept. of Computer Science)



Dr. K. Li, External Member (Dept. of Electrical and Computer Engineering)

Table of Contents

Abstract ii

Table of Contents iii

List of Figures v

Acknowledgements vi

Dedication vii

1 Introduction 1

2 The Language 7
- 2.1 The Basics 7
- 2.2 Syntax 13
- 2.3 Semantics 16
  - 2.3.1 Static Semantics 16
  - 2.3.2 Operational Semantics 19

3 Simulating ACS Specifications 24
- 3.1 Path Expression Language 25
  - 3.1.1 Path Expression Semantics 28
- 3.2 Simulation 31

4 Examples 34
- 4.1 Dining Philosopher 34
- 4.2 Mobile Robots 42

5 Implementation 52
- 5.1 Optimizations 52
- 5.2 System Description 56

5.2.1 Expression Parsing 56
5.2.2 Implementation of ACS Operational Semantics 57
5.2.3 CPE Semantics 64
5.2.4 Simulation 64

6 Concluding Remarks 67
6.1 Contributions 67
6.2 Future Work 68

Bibliography 70

Appendices 72

A Value Expressions 73
A.1 Syntax 73
A.2 Data Evaluation 74

B Bisimulation Equivalence 77

C Source Code 79

List of Figures

2.1 Buffer of size 2 . . . 10
2.2 Buffer of size N . . . 12
2.3 Abstract Syntax of ACS . . . 15
2.4 Termination Predicate . . . 21
2.5 Operational Semantics for ACS . . . 23

3.1 Syntax of Path Expressions . . . 27
3.2 Semantics for Path Expressions . . . 29
3.3 Path Termination Predicate . . . 30
3.4 Path Deadlock Predicate . . . 30

4.1 Eight Dining Philosophers . . . 35
4.2 Schematic Diagram of Robot . . . 43
4.3 Robot Subsumption Architecture . . . 44

5.1 Efficient Operational Semantics for ACS . . . 55

A.1 Evaluation Semantics . . . 76

Acknowledgements

I would like to thank my supervisor, Dr. M. H. M. Cheng for his continuous encouragement and support during my thesis research. Without his patience and continual support I would have never finished this manuscript. Besides being a source of knowledge and expertise, we shared many experiences preparing and eating a variety of Chinese dishes.

I would like to thank Kim Hogervorst and Marlene Cheng for reading and critiquing the grammatical content of this manuscript. Their comments have only increased the quality of this thesis.

I would like to acknowledge the financial support from the National Science and Engineering Research Council, without it I would not have been able to pursue my studies. In addition, I would like to thank Bell-Northern Research for allowing me to visit their Computing Research Laboratory, and for their monetary support.

Finally, I would like to express my gratitude to my family and friends for their moral support throughout my studies.

Dedicated to my beloved parents

Chapter 1

Introduction

A *concurrent system* is regarded as a set of processes which interact with each other and their environment to achieve a well-defined goal. Some examples include multi-computer real-time systems, communication protocols, process control systems, distributed information systems, and the man-machine interface of many kinds of software. Due to issues such as resource contention, non-determinism, and communication and synchronization between processes, concurrent systems are often much more complex than sequential systems. Since we are interested in the process of computation, not just the results, it can be difficult to achieve sufficient confidence in the correctness of a concurrent system through rigorous testing.

Many formalisms have been introduced to aid in the design and verification of concurrent systems, one of which is *process algebra*. Process algebras use algebraic expressions to express both the architectural and behavioral specifications of concurrent systems. They provide specifications for various levels of abstraction and have a sound algebraic semantic. There are a variety of process algebras. The principal ones are *Communicating Sequential Processes* (CSP) by Hoare [Hoa78, Hoa85], *Algebra*

of Communicating Processes (ACP) by Bergstra and Klop [BK85], and *Calculus of Communicating Systems* (CCS) by Milner [Mil80, Mil89].

Pure process algebras do not include any notion of data or the values being passed, they simply specify the signals or synchronizations that occur. At certain levels of abstractions this is acceptable, but at some point the data does become crucial to the validity of the specification. In [Mil89], Milner defines a simple extension to CCS with value-passing, but defines the semantics in terms of a translation to the pure CCS. For hand proofs this translation is acceptable, however, it is inadequate for automated verification and analysis. Other examples of *value-passing* process algebras include μCRL[GP90, GP91] and Bruns' extension to CCS [Bru93]. In [Bru93], Brun extends CCS with explicit data types and expressions. μCRL is based on ACP, and combines the notions of abstract data types using equational specifications and process algebra.

We introduce a value-passing process algebraic language, called *Algebra of Communicating Systems* (ACS), and provide an operational semantics directly on the value-passing syntax. ACS incorporates features from CCS and ACP, in addition to introducing user-definable operators.

ACS overcomes several limitations of CCS and ACP:

1. variable scoping and binding,
2. pattern-matching of action and agent parameters,
3. sequential composition of processes, and
4. user-definable operators.

Variable scoping and binding. In current value-passing CCS([Mil89], [Bru93], [HL93]), variables are given a *scope* by forcing all parameters of input actions to be

variables. The scope of the variables is then the agent expression following the *prefix* combinator. For example, the CCS expression

$$a(x).b(x).\mathbf{0}^1$$

is interpreted, using lambda notation, as

$$\lambda x \cdot a(x).(\lambda x \cdot b(x).\mathbf{0})$$

The problem lies in trying to express a process description of the form,

$$\sum_{i \in V} a_i.b_i.\mathbf{0}$$

where V is some value domain. We would like to express this in value-passing syntax as

$$a(x).b(x),$$

where the variable x in $b(x)$ is *bound* by the previous x in $a(x)$. [Bru93] overcomes this problem by introducing a notion of indexed actions, where the domain of the index must be finite. [Mil89] and [HL93] cannot express such a specification. In our specification language ACS, the scope of a variable is the entire expression. The first occurrence of a variable provides a binding for all other occurrences.

Pattern-matching in the parameters of input/output actions and agent identifiers. In current value-passing process algebras, input variables may only

[1] In CCS notation, $a(x)$ and $b(x)$ are input actions, x is a variable, and '.' is the prefix combinator.

have *distinct* variables as actual parameters. For example, the expression

$$a(1).P$$

is not allowed. Therefore, an input action can synchronize and communicate with *any* similar output action. Sometimes it is desirable to restrict what output actions with which an input action may synchronize, as in the above example. This also holds for the parameters with agent identifiers used in defining equations. In ACS, input actions and agent identifiers are allowed to have arbitrary expressions as formal parameters.

Sequential composition of processes. CCS only allows action prefixing, not general process sequencing. Therefore, the simple concept of a *fork* and *join* cannot be described succinctly. Consider the simple ordering of tasks — task a is performed before task b and c, both c and b occur before task d, tasks b and c are performed independently. Even this simple description cannot be written directly without adding extra synchronizations. In ACP, this can be described succinctly as

$$a \cdot (b \| c) \cdot d,$$

where '·' is the *sequential composition* combinator (meaning that the processes or tasks are performed in sequence), and '‖' is the *merge* combinator (meaning that the processes are performed independently and concurrently.) We adopt this sequential composition in ACS.

User-definable operators. It is often desirable to capture a certain process *template* or functionality in one reusable syntax. In ACS, users are allowed to define

operators built on top of ACS base combinators.

The system description language ACS is used to describe *how* a specification is achieved. The general technique in designing systems is to start with an abstract model and then iteratively refine it. At each iteration, the design must be *equivalent* to the previous, more abstract, design. Equivalence is an often-used criterion for a correct refinement of a specification. There are a variety of notions of equivalence, some of which are summarized and compared in [vG90, vG93]. However, this type of verification simply verifies that both designs perform the same function. It does not extend to whether or not the design satisfies such criteria as being free of deadlock, *eventually* reaching a certain state or performing a certain action, or the existence of race conditions. A syntax is required to specify *what* criteria a system must satisfy. We incorporate a path expression-like syntax, called *Concurrent Path Expressions* (CPE). CPE is similar to the *Data Path Expressions* introduced in [HK90] for modeling concurrency in parallel debugging. CPE specifications can be used to test properties that ACS descriptions must satisfy.

The last contribution of this thesis is a tool which implements the language of ACS and CPE and provides an extensive set of commands to run, trace, and simulate designs.

In summary, this thesis presents three main contributions:

1. an algebraic description language, ACS, for designing systems,
2. a concurrent path expression specification language, CPE, for specifying properties that ACS designs must satisfy, and
3. a tool to verify and analyze ACS designs.

Chapter 2 introduces the algebraic description language ACS and provides a concrete operational semantics. Chapter 3 introduces the path expression specification

language CPE. Chapter 4 gives detailed examples illustrating the usage of ACS and CPE. Chapter 5 presents an implementation of ACS and CPE in Prolog. Chapter 6 concludes the thesis with a summary of its main contributions, and provides some insights for future work.

Chapter 2

The Language

2.1 The Basics

ACS is an amalgamation of Milner's full value-passing CCS ([Mil89]), sequential composition of ACP, and user-definable operators. The *prefix* combinator of CCS is replaced with the more general *sequential composition* combinator of ACP. ACS is an example of a *very high-level language* (VHLL) as described by Krueger in [Kru92]. The emphasis in [Kru92] is on software re-use, whereas our emphasis is on design abstractions, and the execution, verification and analysis of these designs. Although the description of ACS in this chapter is complete, we do not provide any detailed explanation of CCS, ACP or process algebras in general. For an introduction to process algebraic specification, see [Che92, Che94, Wal87]. For more information about a particular process algebra, see [Mil89] for CCS and [BK85] for ACP, [BW90] provides more insight on process algebras in general. We now introduce the notations of ACS and their intuitive meanings by examples, and examine the syntax and semantics more formally.

A *process* or *agent* is any (concurrent) system that performs some set of discrete actions. An *action* is either an interaction with another agent or with its environment. The simplest processes are the degenerate agents `SKIP` and `STOP`. `SKIP` denotes a process that is capable of only terminating successfully, without performing any action. Conversely, `STOP` denotes the process that is not capable of performing any action, it represents a deadlock. `SKIP` and `STOP` differ in that `SKIP` allows execution to continue in a sequential composition; `STOP` does not.

In ACS, there are six combinators for constructing agent expressions — *sequential*, *summation*, *parallel*, *restriction*, *relabeling*, and *conditional* combinators.

A gas station. Consider an automated gas station that accepts only credit cards and sells three types of fuel — regular, premium and diesel. A possible specification could be as follows.

```
Gs  = card,('regular + 'premium + 'diesel),'receipt,SKIP
```

The constant `Gs` denotes a gas station defined by the following process expression: after accepting a credit `card`, it delivers either `regular`, `premium`, or `diesel` fuel followed by a `receipt`. There are two types of actions — an *input* action (e.g. `card`), and an *output* action (e.g. `'receipt`). The actions `receipt` and `'receipt` are considered to be *complementary* actions. Every action is also a process, so a single action a is equivalent to a,`SKIP`. The expression P,Q (called a *sequential composition*), denotes a process that behaves like P, then if P terminates successfully, it behaves like Q. The expression P+Q, called a *summation*, denotes a process that behaves like P or like Q. If the first actions of P and Q are the same, the choice is non-deterministic, as soon as P (or Q) performs its first action, Q (or P) is discarded. A definition of the form A =P states that the constant A behaves like the expression P. We use the convention that all lower case names represent actions, and capitalized

names represent agent identifiers

Actions may take parameters. Consider a process that requests two identical values and outputs a single value.

```
Process  = in(x),in(x),'out(x).
```

The scope of `x` is the whole expression. The first occurrence provides a binding for the rest. For example, after receiving a value 2 via the first `in` action, the process becomes `in(2),'out(2)`. We use the convention that lower case names as arguments represent value variables.

Agent identifiers may also take arguments. Consider a simple nonnegative counter with operations `zero`, `inc`, and `dec` which test for zero, increment the counter, and decrement the counter, respectively. A plausible definition could be as follows:

```
Counter(x)  = ((zero,Counter(x)) if x=0)
            + inc,Counter(x+1)
            + ((dec,Counter(x-1)) if x>0).
```

The agent identifier `Counter(x)` denotes a counter with current value `x`. If an `inc` occurs, the counter state is incremented by one; if a `dec` occurs, the counter state is decremented by one. The expression P `if` E denotes the process that if the boolean expression E *evaluates* to *true* then it behaves like P. The scope of x in $A(x) = P$ is simply P.

A definition for an agent identifier may consist of multiple definitions. For example, the above counter could also be defined as follows:

```
Counter(0)  = zero,Counter(0),
Counter(x)  = inc,Counter(x+1),
Counter(x)  = (dec,Counter(x-1)) if x>0.
```

Notice that the parameter of the first definition contains the constant `0`, other value-passing process algebras do not allow this. This enables pattern-matching between the actual and formal parameters of the agent identifier. Multiple definitions of the same agent identifier are considered to form a choice and conditional based on the matching of the actual to formal parameters.

A buffer of size 2. A buffer of size two can be defined in terms of two single buffers.

```
Cell    := in(x),'out(x),Cell,
Buffer  = (Cell#[comm/out] | Cell#[comm/in])\{comm}
```

The agent identifier `Cell` denotes a single element buffer. The agent identifier `Buffer` denotes a bounded FIFO buffer of size two. Figure 2.1 illustrates how the two cells are connected. The rounded boxes, called nodes, represent actual process expressions. The undirected edge connecting two or more nodes represents a plausible communication and synchronization between processes, and unconnected edges represent interactions with the external environment.

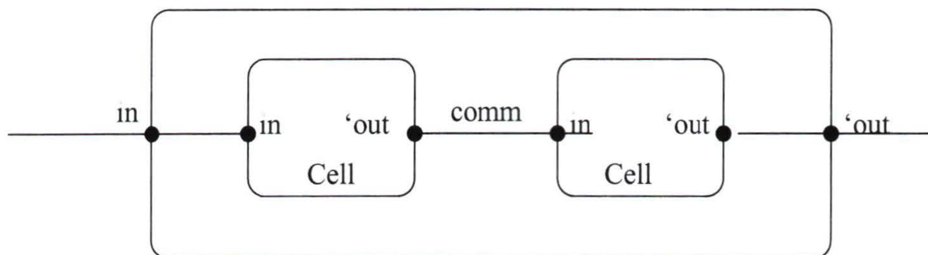


Figure 2.1: Buffer of size 2

The expression P_1 | P_2, called a *parallel composition*, denotes a process that behaves like P_1 and P_2 concurrently and independently. P_1 and P_2 may communicate (or

synchronize) when they are able to perform complementary actions. A communication is regarded as a silent or unobservable action (`tau`). The expression $P\#f$, called a *relabeling*, denotes a process that behaves like P, except that the actions are renamed by the relabeling function f of the form $[b_1/a_1, \ldots, b_n/a_n]$, where $a_1, \ldots, a_n, b_1, \ldots, b_n$ are actions, and where b_i/a_i means $f(a_i) = b_i$, and $f(`a_i) = `b_i$, $1 \leq i \leq n$. The relabeling combinator is a means of reusing processes where two instances have similar behaviors but different actions. In the agent `Buffer`, the expression `Cell#[comm/out]` has the two actions `in(x)` and `'comm(x)`, and the expression `Cell#[comm/in]` has the two actions `comm(x)` and `'out(x)`. This enables the two expressions to communicate which each other via the action `comm`.

The two expressions in `Cell#[comm/out] | Cell#[comm/in]` are not forced to communicate; they may select to perform the actions `'comm(x)` and `comm(x)` independently. The expression $P\backslash L$, called a *restriction*, denotes the process that behaves like P except that the actions and their complements in L are not allowed. The restriction combinator internalizes actions and forces communication. For example, in the agent `Buffer`, the actions `comm(x)` and `'comm(x)` cannot occur independently, forcing the communication (`comm(x)`,`'comm(x)`).

User-definable operators. Operators are similar to agent identifiers, except that the parameters are process variables rather than value variables. The two-element buffer could be defined as follows:

```
LINK($X,$Y)   ::= ($X#[comm/out] | $Y#[comm/in])\{comm},
Buffer2      ::= LINK(Cell, Cell).
```

`LINK($X,$Y)` is an operator that takes two arguments, process variables `$X` and `$Y`, and expands into a process expression. Notice that `Buffer2` is similar to the previous `Buffer` definition after expansion. A definition of the form O ::= P states that

the operator O behaves like the process expression P, where O may contain agent variables We use the convention that all capitalized names represent operators, and all upper case names preceded by the $ sign represent agent (process) variables

A three-element buffer could be described as follows

```
Buffer3  = LINK(Cell, LINK(Cell, Cell)),
```

or

```
Buffer3  = LINK(LINK(Cell, Cell), Cell)
```

Generalizing to a generic FIFO buffer of size N, as depicted in Figure 2 2, a plausible definition could be described recursively as follows

```
BufferN(1)  = Cell,
BufferN(x)  = LINK(Cell,BufferN(x-1)) if x>1
```

Note that the `comm` actions are not visible outside of the agent `BufferN`, but are observed as a communication, `tau`

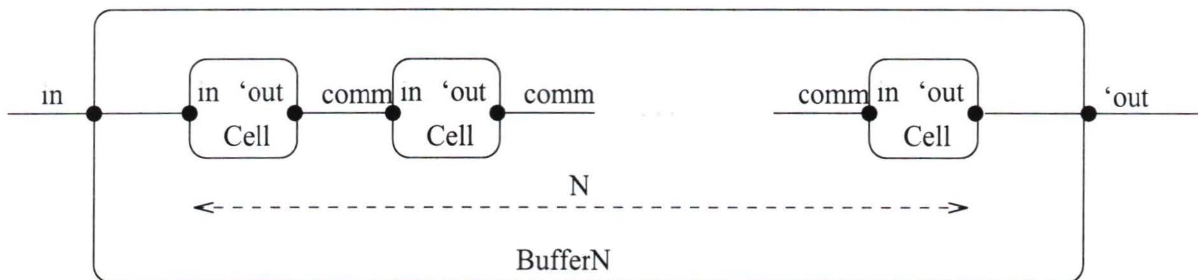


Figure 2 2 Buffer of size N

So far, the notion of operators has been absorved as a textual macro expansion, but they are much more powerful than that Consider the case where an action is to be repeated n times (a^n) A simple recursive definition

```
A(1)   = a,
A(n)   = a,A(n-1) if n > 1,
```

could be defined to capture that requirement Now consider the general case where we require a process to be repeated multiple times A recursive definition could be defined to repeat the required number of times This must be done for every recursive definition A general *repeat* operator could be defined as

```
REPEAT($X,0)   = SKIP,
REPEAT($X,n)   = ($X,REPEAT($X,n-1) ) if n>1,
```

that repeats a given process expression `$X`, `n` times Hence, the above definition could be redefined as

```
A(n)   = REPEAT(a,n)
```

From now on, we assume the precedence of the combinators (in decreasing order of binding power) is as follows *restriction* and *relabelling* (tightest binding), *sequential composition*, *parallel composition*, *summation*, and *conditional* (weakest binding) The expression

$$P_1 + P_2, P_3\#f \mid P_4, P_5\backslash L \texttt{ if } E$$

means

$$(P_1 + ((P_2, (P_3\#f)) \mid (P_4, (P_5\backslash L)))) \texttt{ if } E$$

2.2 Syntax

Let $\mathcal{E}$ denote the set of *value expressions* with typical element e (See Appendix A for a detailed description of all the value expressions and the base data types)

Let $\mathcal{L}$ be an infinite set of action *labels* consisting of two complementary but disjoint subsets, the set of *names* A and the set of *co-names* $\overline{A}$ Typical elements of A are

$l, l_1, \ldots$ and their complements $`l, `l_1, \ldots$ are in $\overline{A}$. a ranges over $\mathcal{L}$. We extend complementation to the whole of $\mathcal{L}$ so that $``a = a$. Now the set of actions consists of parameterized *labels*, and includes the distinguished *silent* action `tau`, which is considered to be unobservable outside a system. `tau` has no complement.

Definition 1 The set of *input actions InAct* is defined to be

$$InAct = \{l(t_1, \ldots, t_n) \mid n \geq 0, t_i \in \mathcal{E}, l \in A\}.$$

The set of *output actions OutAct* is defined to be

$$OutAct = \{`l(t_1, \ldots, t_n) \mid n \geq 0, t_i \in \mathcal{E}, `l \in \overline{A}\}.$$

The complete set of *actions Act* is

$$Act = InAct \cup OutAct \cup \{\texttt{tau}\}.$$

Typical elements in *Act* are $\alpha, \beta, \ldots$. We write $l(t_1, \ldots, t_n)$ as $l(\vec{t})$, and $a(t_1, \ldots, t_n)$ and $`a(t_1, \ldots, t_n)$ as $a(\vec{t})$ and $`a(\vec{t})$, respectively. *label(α)* denotes the label of the action α. □

Relabeling functions are total functions that map actions to actions. However, the syntax only allows the description of finite mappings from labels to labels.

Definition 2 A *finite relabeling* function $f : \mathcal{L} \mapsto \mathcal{L}$ has the property

$$f(a) = b \text{ if and only if } f(`a) = `b.$$

f is described as $[b_1/a_1, \ldots, b_n/a_n]$, $a_1, \ldots, a_n, b_1, \ldots, b_n \in \mathcal{L}$, where b_i/a_i means $f(a_i) = b_i$, and $f(`a_i) = `b_i$, $1 \leq i \leq n$. □

Definition 3 Let f be a finite relabeling function. $f_T : Act \mapsto Act$ is a (*total*)

relabeling function that extends f by the following definition

$$f_T(a(\vec{t})) \stackrel{\text{def}}{=} \begin{cases} f(a)(\vec{t}), & a \in dom(f) \\ a(\vec{t}), & a \notin dom(f) \end{cases},$$

$$f_T(\texttt{tau}) \stackrel{\text{def}}{=} \texttt{tau}$$

□

Let $\mathcal{X}$ be the set of *agent variables* with typical elements $X, Y, \ldots$ Let $\mathcal{A}$ denote the set of *agent identifiers*, and $\mathcal{A}_n$ denote the set of agent identifiers of *arity* n, a non-negative number representing the number of formal parameters it takes. Let Σ denote the collection of user-defined operators, and Σ_k denote the set of operators of arity k. A ranges over $\mathcal{A}$ (or $\mathcal{A}_n$), and O ranges over Σ (or Σ_k).

Definition 4 The set $\mathcal{P}$ of *process expressions* with typical element P is the set generated by the grammar shown in Figure 2.3, $\alpha \in Act$, $X \in \mathcal{X}$, f is a finite relabeling function, $L \subseteq \mathcal{L}$, $A \in \mathcal{A}_n$, $O \in \Sigma_k$, $e, e_1, \ldots, e_n$ are value expressions, be is a boolean expression, and $p_1, \ldots, p_k \in \mathcal{E} \cup \mathcal{P}$. □

P	$::= X$	*Agent Variable*
	$\mid \alpha$	*Action*
	$\mid A(e_1, \ldots, e_n)$	*Agent Identifier*
	$\mid O(p_1, \ldots, p_k)$	*User-defined Operator Identifier*
	$\mid P$`;`P	*Sequential Composition*
	$\mid P$`+`P	*Summation*
	$\mid P$`|`P	*Parallel Composition*
	$\mid P\backslash L$	*Restriction*
	$\mid P$`#`f	*Relabeling*
	$\mid P$ `if` be	*Conditional*
	$\mid$ `SKIP`	*Termination*
	$\mid$ `STOP`	*Deadlock*

Figure 2.3 Abstract Syntax of ACS

The meaning of an agent identifier is given by a collection of *agent definitions*.

Definition 5 Let $A \in \mathcal{A}_n$, $t_1, \ldots, t_n \in \mathcal{E}$, and $P \in \mathcal{P}$. Then an *agent definition* corresponding to A is an equation of the form

$$A(t_1, \ldots, t_n) = P$$

□

Similarly, the meaning of an operator identifier is given by a collection of *operator definitions*.

Definition 6 Let $O \in \Sigma_k$, $p_1, \ldots, p_k \in \mathcal{E} \cup \mathcal{P}$, and $P \in \mathcal{P}$. Then an *operator definition* corresponding to O is an equation of the form

$$O(p_1, \ldots, p_k) = P$$

□

An operator definition differs from an agent definition in that it allows arbitrary process expressions and value expressions as formal (actual) parameters.

2.3 Semantics

2.3.1 Static Semantics

Not all process expressions generated by the grammar in Figure 2.3 have valid meanings. For instance, any process expression containing a *free* agent variable has no meaning. In

$$\text{`}a(x)\texttt{,SKIP,}$$

the variable **x** is not *bound* to any value, and since it is an output action, it cannot receive any value.

Before we discuss what constitutes a valid process expression, we need to reiterate the notion of variable scoping, and *free* and *bound* variables. The scope of a variable is the entire process expression. The *first* occurrence of a variable is considered to be *free* and all other occurrences are *bound* to the value assigned to the free occurrence. The first occurrence is described syntactically, parsing from left to right. In the process expression

$$a(x), (b(x) + d(x)),$$

the variable x occurs free in $a(x)$ but bound in $b(x)$ and $d(x)$.

In agent and operator definitions, the variables occurring as formal parameters are considered to be the first occurrence, and bind all occurrences in the right hand process expression.

The set of (value and agent) *variables* of an expression P is denoted by *vars*(P). We will often need to refer specifically to the agent and value variables of a process expression as $avars(P)$ and $dvars(P)$, respectively. Note that $avars(P) \cap dvars(P) = \emptyset$.

A *substitution* is a function mapping variables to expressions. We will use the normal notation for substitutions, $\{e_1/x_1, \ldots, e_n/x_n\}$, which is the function that assigns the expression e to the variable x and leaves all other variables unmapped. We use $\theta, \theta_1 \ldots$ to range over substitutions, and write $P\theta$ for the process expression obtained from P by simultaneously replacing all occurrences of any variable x_i by $\theta(x_i)$. Similarly, $e\theta$ is the result of applying θ to the e. $\theta\theta_1$ is the composition of substitutions, therefore, $P\theta\theta_1 = (P\theta)\theta_1$.

Informally, for process expression P to be considered *well-formed*, the following conditions must be met:

- Only input actions can introduce new value variables. Therefore, output actions, agent identifiers, user-defined operator expressions, and value expression components of the *conditional* combinator cannot introduce new value variables.

- All process variables must be bound. Therefore, process expressions used in agent definitions cannot contain agent variables, and for an operator definition, $O(\vec{p}) ::= P$, $avars(P) \subseteq avars(O(\vec{p}))$.

- For any subexpression $P_1 \mid P_2$ of P, the *free* variables in P_1 and P_2 must be disjoint.

The last point ensures that all communication is via complementary actions only. For example, assume we allow expressions of the form $a_1(x) \mid a_2(x)$, where x is free. By interleaving semantics for concurrent processes and by using the definitions of variable scoping and binding, either a_1 or a_2 may communicate with the environment and upon communication, the other is *mysteriously* forced to receive the same value. That is, if a_1 receives the value 3, $a_2(x)$ implicitly becomes $a_2(3)$. Similarly, if, for example, a_2 receives `true`, $a_1(x)$ becomes $a_1(\texttt{true})$. This type of communication is undesirable, so by forcing the free variables of concurrent processes to be distinct, this *implicit* communication is eliminated.

Note that in an agent definition, $A ::= P$, the process expression P by itself may not be well-formed; by placing P in a definition, the free variables of P may be formal parameters of A, and hence the complete definition becomes well-formed. In defining a well-formed predicate for process expressions, a set of bound variables is required.

More formally, a well-formed predicate $wf(P)_\rho$ is defined and interpreted as: *given a set of bound variables* ρ, P *is well-formed*.

Definition 7 Let $P \in \mathcal{P}$, and $\rho \subseteq \mathcal{V} \cup \mathcal{X}$ be a set of bound variables. $wf(P)_\rho$, called the *well-formed predicate*, is defined inductively as follows.

1. $wf(\mathtt{tau})_\rho$, $wf(\mathtt{SKIP})_\rho$, $wf(\mathtt{STOP})_\rho$;

2. $wf(a(\vec{t}))_\rho$;

3. if $vars(a(\vec{e})) \subseteq \rho$ then $wf(\text{'}a(\vec{e}))_\rho$;

4. if $vars(A(\vec{e})) \subseteq \rho$ then $wf(A(\vec{e}))_\rho$;

5. if $vars(O(\vec{p})) \subseteq \rho$ then $wf(O(\vec{p}))_\rho$;

6. if $wf(P)_\rho$ then $wf(P\backslash L)_\rho$ and $wf(P\#f)_\rho$;

7. if $wf(P_1)_\rho$ and $wf(P_2)_{(vars(P_1)\cup\rho)}$ then $wf(P_1\,,P_2)_\rho$;

8. if $wf(P_1)_\rho$ and $wf(P_2)_\rho$ then $wf(P_1 + P_2)$;

9. if $wf(P_1)_\rho$, $wf(P_2)_\rho$ and $(vars(P_1) \setminus \rho) \cap (vars(P_2) \setminus \rho) = \emptyset$ then $wf(P_1 \mid P_2)_\rho$;

10. if $wf(P)_\rho$ and $vars(be) \subseteq \rho$ then $wf(P \mathtt{~if~} be)_\rho$.

□

An agent definition $A(\vec{e}) := P$ is considered *well-formed* if $avars(P) = \emptyset$ and the predicate $wf(P)_{vars(A(\vec{e}))}$ holds. Similarly, an operator definition $O(\vec{p}) ::= P$ is well-formed if $avars(P) \subseteq avars(O(\vec{(p}))$ and the predicate $wf(P)_{vars(O(\vec{p}))}$ holds.

For the remainder of this thesis, we will only consider well-formed process expressions, agent definitions, and operator definitions.

2.3.2 Operational Semantics

The standard approach in providing operational semantics to a value-passing process algebra is to translate it into its underlying *pure* (non value-passing) process algebra.

The main characteristic of the translation, mapping a process P to $\widehat{P}$, is that an input action $a(x)$ is mapped into the term,

$$\sum_{v \in V} a_v\{v/x\},$$

where V is the value domain of the free variable x in P, which may or may not be finite. This simple scheme may produce infinite process definitions and infinite branching structures which become impossible to implement. Therefore, the operational semantics for ACS is introduced directly for the value-passing calculus.

The semantics of the language is formally given as a *labeled transition system*.

Definition 8 A *labeled transition system* is a triple (S, L, R), where S is a set of states, L is a set of labels, and $R \subseteq S \times L \times S$ is a *labeled transition relation*. We will refer to $(s, l, s') \in R$ as $s \xrightarrow{l} s'$. □

In our transition system, we shall take S to be $\mathcal{P}$, the process expressions, and L to be the set of actions, *Act*. The transition relation is the largest set obtained by the semantic rules defined as follows:

$$R = \{(P, \alpha, P') \mid P \xrightarrow{\alpha} (P', \theta) \text{for some } \theta\}$$

The transition rule $P \xrightarrow{\alpha} (P', \theta)$ states that process P, under the substitution θ, may perform the action α to become the process P'. The transitions of composite processes are defined in terms of the transitions of its component processes. The general rule used to describe this inference is

$$\frac{T_1, \ldots, T_n}{T} C,$$

where $T_1, \ldots, T_n, T$ denote transitions. The rule states that if $T_1, \ldots, T_n$ are transi-

tions satisfying the condition C, then T is also a transition.

In forming the transition rules, an auxiliary predicate is needed to indicate whether or not a process has a (successful) termination option. Clearly, the degenerate process `SKIP` can always terminate, but `STOP` cannot, since it signifies unsuccessful termination. Similarly, any action cannot terminate since it must first perform the action. If a process P can terminate, then P combined in a restriction or relabeling can also terminate. In a summation process expression, if either process can terminate, the whole process expression can terminate. Both processes combined in sequence or in parallel must terminate before the whole expression can terminate. For the conditional combinator, an extra constraint is that the boolean expression must evaluate to `true`. This evaluation is possible since the predicate is defined for well-formed processes only. The complete inductive definition of the *termination predicate* ($\downarrow$) is given in Figure 2.4.

---

1. `SKIP`$\downarrow$;
2. if $P\downarrow$ then $(P\texttt{\#}f)\downarrow$, $(P\backslash L)\downarrow$, $(P\texttt{+}Q)\downarrow$, and $(Q\texttt{+}P)\downarrow$;
3. if $P_1\downarrow$ and $P_2\downarrow$ then $P_1\,\texttt{;}\,P_2$ and $P_1\,\texttt{|}\,P_2$;
4. if $P\downarrow$ and $t \Rightarrow_{\mathcal{D}}$ `true` then $(P\ \texttt{if}\ b)\downarrow$;
5. if $A(\vec{t_1}) \mathrel{\texttt{:=}} P$, $[\![\vec{t_1}\theta]\!] = \vec{v}$ for some θ, $(P\theta)\downarrow$, and $[\![\vec{t_2}]\!] = \vec{v}$ then $A(\vec{t_2})\downarrow$;
6. if $O(\vec{p_1}) \mathrel{\texttt{:=}} P$, $[\![\vec{p_1}\theta]\!] = \vec{v}$ for some θ, $(P\theta)\downarrow$, and $[\![\vec{p_2}]\!] = \vec{v}$ then $O(\vec{p_2})\downarrow$.

Figure 2.4: Termination Predicate

The complete set of transition rules is given in Figure 2.5, where $[\![\ ]\!]$ is the data evaluation semantics for value expressions (defined in detail in Appendix A.) The summation and parallel combinators are commutative, consequently, the **Sum**, **Com1**, **Com2**, and **Com3** rules each imply two rules. For example, the **Sum** rule implies

the two symmetric rules,

$$\frac{P_1 \xrightarrow{\alpha} (P', \theta)}{P_1 + P_2 \xrightarrow{\alpha} (P', \theta)} \text{ and } \frac{P_1 \xrightarrow{\alpha} (P', \theta)}{P_2 + P_1 \xrightarrow{\alpha} (P', \theta)}$$

The rule for input actions (**In**) can be read as follows: for any substitution θ such that $[\![\vec{e}\theta]\!] = \vec{v}$, the input action $a(\vec{e})$ can force a transition $\xrightarrow{a(\vec{v})}$ to become the residual $(\mathtt{SKIP}, \theta)$. The substitution in the residual is needed to propagate variable bindings in the sequential composition rule **Seq1**. Notice that only the input rule introduces bindings since, by definition of well-formed process expressions, only input actions may contain free variables as actual parameters.

In the **Defn1** and **Defn2** rules, the variables in the definitions $A(\vec{v_2}) \;\hat{=}\; P$ and $O(\vec{p_2}) \;\hat{=}\; P$ are assumed to be distinct from the variables in the actual usages $A(\vec{e_1})$ and $O(\vec{p_1})$, respectively. That is,

$$\begin{aligned} vars(A(\vec{e_1})) \cap (vars(A(\vec{e_2})) \cup vars(P)) &= \emptyset, \\ vars(O(\vec{p_1})) \cap (vars(O(\vec{p_2})) \cup vars(P)) &= \emptyset. \end{aligned}$$

$$\textbf{In}\quad \frac{}{a(\vec{e}) \xrightarrow{a(\vec{v})} (\texttt{SKIP}, \theta)}\ [\![\vec{e}\theta]\!] = \vec{v} \qquad \textbf{Out}\quad \frac{}{\text{`}a(\vec{e}) \xrightarrow{\text{`}a(\vec{v})} (\texttt{SKIP}, \emptyset)}\ [\![\vec{e}]\!] = \vec{v}$$

$$\textbf{Tau}\quad \frac{}{\texttt{tau} \xrightarrow{\texttt{tau}} (\texttt{SKIP}, \emptyset)}$$

$$\textbf{Seq1}\quad \frac{P \xrightarrow{\alpha} (P', \theta)}{P\texttt{,}Q \xrightarrow{\alpha} (P'\texttt{,}Q\theta, \theta)} \qquad \textbf{Seq2}\quad \frac{Q \xrightarrow{\alpha} (Q', \theta)}{P\texttt{,}Q \xrightarrow{\alpha} (Q', \theta)}\ P\!\downarrow$$

$$\textbf{Sum}\quad \frac{P \xrightarrow{\alpha} (P', \theta)}{P\texttt{+}Q \xrightarrow{\alpha} (P', \theta)}$$

$$\textbf{Com1}\quad \frac{P \xrightarrow{\alpha} (P', \theta)}{P\texttt{|}Q \xrightarrow{\alpha} (P'\texttt{|}Q, \theta)} \qquad \textbf{Com2}\quad \frac{Q \xrightarrow{\alpha} (Q', \theta)}{P\texttt{|}Q \xrightarrow{\alpha} (Q', \theta)}\ P\!\downarrow$$

$$\textbf{Com3}\quad \frac{P \xrightarrow{\text{`}a(\vec{v})} (P', \emptyset),\ Q \xrightarrow{a(\vec{v})} (Q', \theta)}{P\texttt{|}Q \xrightarrow{\texttt{tau}} (P'\texttt{|}Q', \theta)}$$

$$\textbf{Ren}\quad \frac{P \xrightarrow{\alpha} P', \theta}{P\texttt{\#}f \xrightarrow{f_T(\alpha)} (P'\texttt{\#}f, \theta)} \qquad \textbf{Res}\quad \frac{P \xrightarrow{\alpha} (P', \theta)}{P\backslash L \xrightarrow{\alpha} (P'\backslash L, \theta)}\ label(\alpha) \notin L$$

$$\textbf{If}\quad \frac{P \xrightarrow{\alpha} (P', \theta)}{P\ \texttt{if}\ be \xrightarrow{\alpha} (P', \theta)}\ [\![be]\!] = \texttt{true}$$

$$\textbf{Defn1}\quad \frac{P\theta_1 \xrightarrow{\alpha} (P', \theta)}{A(\vec{e_1}) \xrightarrow{\alpha} (P', \theta)}\ A(\vec{e_2}) = P,\ [\![\vec{e_1}]\!] = \vec{v},\ [\![\vec{e_2}\theta_1]\!] = \vec{v}$$

$$\textbf{Defn2}\quad \frac{P\theta_1 \xrightarrow{\alpha} (P', \theta)}{O(\vec{t_1}) \xrightarrow{\alpha} (P', \theta)}\ O(\vec{p_2}) = P,\ [\![\vec{p_1}]\!] = \vec{v},\ [\![\vec{p_2}\theta_1]\!] = \vec{v}$$

Figure 2 5 Operational Semantics for ACS

Chapter 3

Simulating ACS Specifications

There are many ways of designing the same system. When are two specifications the same? How do you know that the specification actually meets the requirements the system was designed to satisfy? In process algebra the common method for determining if two processes are considered *equivalent* is *bisimulation*. Informally stated, this says that each process can *simulate* the other. However, this type of equivalence is often too strong a requirement and also requires writing two specifications for the same system. Simple characteristics such as *eventuality* (the possibility that a certain criterion will ever be met), starvation, deadlock, and possible execution paths cannot be described by bisimulation. Some type of simulation is needed, or a way of testing if a specification actually satisfies the initial requirements. The most basic simulation is stepping through the process one action at a time. A more elegant way is to introduce simple path expressions, this allows us to define requirements of a given ACS specification.

In this chapter we introduce an ACS-like path expression that allows us to specify possible execution paths more concisely without using the complete syntax of ACS.

The idea is to remove the architectural structure from ACS and only specify certain requirements that a process expression must satisfy.

3.1 Path Expression Language

In defining requirements, we are more interested in the actions that are possible or not possible, rather than actual architectural design. For instance, requirements such as *will an action a ever occur?* could be described by the expression

$$(\nu{*}n)\texttt{,}a$$

The expression $S{*}n$ denotes zero or more (but not more than n) repetitions of the path expression S. The path expression sequential combinator (`,`) is similar to the process expression combinator (`,`). Using the `+` combinator, $S{*}n$ is equivalent to

$$\texttt{SKIP} + S + (S\texttt{,}S) + \cdots + \underbrace{(S\texttt{,} \cdots \texttt{,}S)}_{n}$$

The special action variable ν is interpreted as *any* variable, and is useful for *don't-care* action sequences. Using just regular variables would bind all other occurrences so that an expression of the form

$$X{*}n$$

would mean up to n occurrences of the *same* action. However,

$$\nu{*}n$$

means up to n different actions. In concrete examples the underscore symbol (_) is used to denote ν.

Another useful notation is exactly n occurrences denoted by S^n, which is equivalent to the following:

$$\underbrace{S, \ldots, S}_{n}$$

The reason for restricting the * and ^ combinators to a finite number of repetitions is that since this is to be automated, the determination of whether a process expression satisfies a path expression must terminate.

It is often a requirement that an action or a certain set of actions cannot occur in a certain state. The notation

$$!L,$$

where L is a set of actions, denotes the occurrence of an action that is not in the given set L. L may contain variables as well as actions. For example, in the `Counter` agent defined in Chapter 2, after incrementing the counter, the zero test should not be available. That is, `Counter` must satisfy

```
inc,!{zero}.
```

As in ACS, the | combinator denotes independent action. a | b is equivalent to the path expression

$$(a,b) + (b,a).$$

The path expression language is defined formally in the following definition.

Definition 9 The set of *path expressions* is the set generated by the grammar in Figure 3.1, where $\alpha \in Act$, $a \in \mathcal{X} \cup Act$, $X \in \mathcal{X}$, and ν is *any* variable. □

We conclude this section with some examples demonstrating the properties that are

S = X | ν | α | S+S | S|S | S,S | S^n | S*n | !{a, ,a} | STOP | SKIP

Figure 3.1 Syntax of Path Expressions

expressible with this path expression language

Deadlock The question of whether a process ever reaches deadlock after a finite number of actions can be stated as

$(\nu$*$n)$,STOP,

where n is some fixed number Notice that it doesn't say that anything more than n action sequences will not deadlock

Starvation Starvation means that a certain action is never performed The path expression

(!{a})^n

states that for all execution paths of no more than n actions, the action a is never performed

Eventuality The question of whether a certain action will ever occur can be stated by the path expression

$(\nu$*$n)$,a

Race Condition A possible race condition between two actions occurs when there exists a given state of the system such that both actions are possible The path

expression

$$(\nu{*}n),(a \,|\, b)$$

captures this property.

Fairness. The *fairness* requirement means that all processes or actions are treated equally. One interpretation of this is that for two actions a and b to be treated fairly, one cannot occur more often than the other in any execution path. The path expression

$$((('\{a,b\}){*}n),a) \,|\, ((('\{a,b\}){*}m),b)$$

denotes an execution path with exactly one occurrence of a and b. If the whole expression is placed within a * expression, the required fairness property would be satisfied.

3.1.1 Path Expression Semantics

The operational semantics of the path expressions are similar to the process expressions, and are given by a labeled transition system. The set of states is represented by the path expressions S, the labels are $Act \cup \mathcal{X} \cup \{\nu\} \cup \{'\{a, \ldots ,a\}\}$, and the transition relation is the largest set obtained by the semantic rules defined in Figure 3.2.

Two auxiliary predicates are needed. First is the *path termination* predicate, denoted by *term(S)*, that holds if a path expression S may terminate successfully. The other is the *path deadlock* predicate, denoted by *dead(S)*, that holds if a path expression S may terminate unsuccessfully. The meaning of these predicates is straightforward, and their inductive definitions are given in Figure 3.3 and Figure 3.4.

The meaning of the |, , , and + combinators for path expressions are the same as the

1	$\overline{\alpha \overset{\alpha}{\rightsquigarrow} \texttt{SKIP}}$	$\overline{X \overset{X}{\rightsquigarrow} \texttt{SKIP}}$
	$\overline{\nu \overset{\nu}{\rightsquigarrow} \texttt{SKIP}}$	$\overline{{}^{\shortmid}L \overset{{}^{\shortmid}L}{\rightsquigarrow} \texttt{SKIP}}$
2	$\dfrac{S_1 \overset{s}{\rightsquigarrow} S_1'}{S_1 \texttt{,} S_2 \overset{s}{\rightsquigarrow} S_1' \texttt{,} S_2}$	$\dfrac{term(S_1),\ S_2 \overset{s}{\rightsquigarrow} S_2'}{S_1 \texttt{,} S_2 \overset{s}{\rightsquigarrow} S_2'}$
3	$\dfrac{S_1 \overset{s}{\rightsquigarrow} S_1'}{S_1 \texttt{+} S_2 \overset{s}{\rightsquigarrow} S_1'}$	$\dfrac{S_2 \overset{s}{\rightsquigarrow} S_2'}{S_1 \texttt{+} S_2 \overset{s}{\rightsquigarrow} S_2'}$
4	$\dfrac{S_1 \overset{s}{\rightsquigarrow} S_1'}{S_1 \mid S_2 \overset{s}{\rightsquigarrow} S_1' \mid S_2}$	$\dfrac{S_2 \overset{s}{\rightsquigarrow} S_2'}{S_1 \mid S_2 \overset{s}{\rightsquigarrow} S_1 \mid S_2'}$
5	$\dfrac{S \overset{s}{\rightsquigarrow} S'}{S\texttt{^}1 \overset{s}{\rightsquigarrow} S'}$	$\dfrac{S \overset{s}{\rightsquigarrow} S'}{S\texttt{^}n \overset{s}{\rightsquigarrow} S' \texttt{,} (S\texttt{^}(n-1))}\ n > 1$
6	$\dfrac{S \overset{s}{\rightsquigarrow} S'}{S\texttt{*}1 \overset{s}{\rightsquigarrow} S'}$	$\dfrac{S \overset{s}{\rightsquigarrow} S'}{S\texttt{*}n \overset{s}{\rightsquigarrow} S' \texttt{,} (S\texttt{*}(n-1))}\ n > 1$

Figure 3 2 Semantics for Path Expressions

combinators for process expressions The first set of rules defines the base cases that have to be matched by an appropriate process expression transition The last two sets of rules define the semantics of the ^ and * combinators Notice that the *term()* predicate captures the fact that the * combinator means zero or more

1. $term(S*n)$, $term(\mathtt{SKIP})$;
2. if $term(S)$ then $term(S\hat{}n)$;
3. if $term(S_1)$ or $term(S_2)$ then $term(S_1+S_2)$ and $term(S_1+S_2)$;
4. if $term(S_1)$ and $term(S_2)$ then $term(S_1,S_2)$, $term(S_1|S_2)$, and $term(S_2|S_1)$.

Figure 3.3: Path Termination Predicate

1. $dead(\mathtt{STOP})$;
2. if $dead(S)$ then $dead(S\hat{}n)$, and $dead(S*n)$;
3. if $dead(S_1)$ then $dead(S_1,S_2)$, $dead(S_1+S_2)$, $dead(S_2+S_1)$, $dead(S_1|S_2)$, and $dead(S_2|S_1)$;
4. if $term(S_1)$ and $dead(S_2)$ then $dead(S_1,S_2)$.

Figure 3.4: Path Deadlock Predicate

3.2 Simulation

A process expression *simulates* a path expression if, whenever the path expression makes a transition, a *matching* transition exists for the process expression. For example, in the automated gas station example presented in Chapter 2,

```
GS  = card,('regular + 'premium + 'diesel),'receipt,SKIP
```

simulates the following path expressions:

```
card,'regular,
card,'premium,
card,'diesel
```

It does not, however, simulate

```
_*100,(card | ('regular + 'premium + 'diesel),
```

which says that there exists a state where either the environment presents a credit card, or fuel is delivered. In other words, a customer has an option of either receiving free fuel or paying for it. This is not a desirable property from the retailer's point of view.

A few preliminary definitions are needed to express the meaning simulation.

Definition 10 If $t = \alpha_1 \ldots \alpha_n \in Act*$, P a process expression, and $P \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} (P', \theta)$, then we call t an *execution path* of P. □

Definition 11 If S is a path expression and $S \overset{s_1}{\rightsquigarrow} \ldots \overset{s_n}{\rightsquigarrow} S'$, then we call $t = s_1 \ldots s_n$ an *execution path* of S. □

Any process expression simulates a path expression that can terminate successfully (by not making any transitions). Any process that cannot make any transitions simulates a path expression that can terminate unsuccessfully. For all execution paths

of a path expression, a process must have a matching execution path. A simulation is defined formally in the following definition.

Definition 12 Let P be a process expression and S a path expression. Then P *(strong) simulates* S, denoted by $P \vdash^s S$, if

1. *term(S)*;
2. *dead(S)* and $P \not\longrightarrow$,
3. $S \overset{\alpha}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\longrightarrow} P', \theta$ for some P' and θ, and $P'\theta \vdash^s S'\theta$,
4. $S \overset{X}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\longrightarrow} P', \theta$ for some P', θ and α, and $P'\theta \vdash^s S'[\alpha/X]$,
5. $S \overset{\chi}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\longrightarrow} P', \theta$ for some P', θ and α, and $P'\theta \vdash^s S'$,
6. $S \overset{!L}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\longrightarrow} P', \theta$ for some P', θ and $\alpha \notin L$, and $P'\theta \vdash^s S'$.

□

Strong simulation matches every action of the process expression with an action of the path expression — even for `tau` actions. A more relaxed requirement for `tau` actions is to require that each `tau` action be matched by zero or more `tau` actions.

Definition 13 If $\alpha \in Act$ then $P \overset{\alpha}{\Longrightarrow} (P', \theta)$ if

$$
\begin{array}{l}
P \overset{\texttt{tau}}{\longrightarrow} P_1, \theta_1 \\
P_1\theta_1 \overset{\texttt{tau}}{\longrightarrow} P_2, \theta_2 \\
\dots \\
P_{n-1}\theta_{n-1} \overset{\texttt{tau}}{\longrightarrow} P_n, \theta_n \\
P_n\theta_n \overset{\alpha}{\longrightarrow} P_{n+1}, \theta_{n+1} \\
P_{n+1}\theta_{n+1} \overset{\texttt{tau}}{\longrightarrow} P_{n+2}, \theta_{n+2} \\
\dots \\
P_{n+m}\theta_{n+m} \overset{\texttt{tau}}{\longrightarrow} P_{n+m+1}, \theta_{n+m+1}
\end{array}
$$

where $n, m \geq 0$, $\theta = \theta_{n+m+1}$ and $P' = P_{n+m+1}$. P' is called an *α-descendant* of P. □

The definition for *weak simulation* is similar to the definition of strong simulation

Definition 14 Let P be a process expression and S a path expression. Then P *weak simulates* S, denoted by $P \vdash^w S$, if

1. *term(S)*,
2. *dead(S)* and $P \overset{\mathtt{tau}}{\Longrightarrow} P', \theta$, $P'\theta \not\longrightarrow$,
3. $S \overset{\alpha}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\Longrightarrow} P', \theta$ for some P' and θ, and $P'\theta \vdash^w S'\theta$,
4. $S \overset{X}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\Longrightarrow} P', \theta$ for some P', θ and α, and $P'\theta \vdash^w S'[\alpha/X]$,
5. $S \overset{X}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\Longrightarrow} P', \theta$ for some P', θ and α, and $P'\theta \vdash^s S'$,
6. $S \overset{!L}{\rightsquigarrow} S'$ and $P \overset{\alpha}{\Longrightarrow} P', \theta$ for some P', θ and $\alpha \notin L$, and $P'\theta \vdash^w S'$.

□

Chapter 4

Examples

ACS may be used to design systems for a variety of programming paradigms. In this chapter we present two examples which demonstrate the power and expressibility of the language.

Dining Philosophers. This problem is used to demonstrate how design flaws can be found and corrected by using the path expressions.

Mobile Robots. This problem demonstrates how one can model the complete system, including the physical environment.

4.1 Dining Philosopher

The classical *dining philosophers* problem is well-known for illustrating a deadlock situation. The problem is depicted in Figure 4.1, where we have eight (N in general) philosophers and eight chopsticks. Each philosopher is required to pick up both

the chopstick on the left and on the right. A deadlock situation occurs when every philosopher picks up the left chopstick first then the right chopstick.

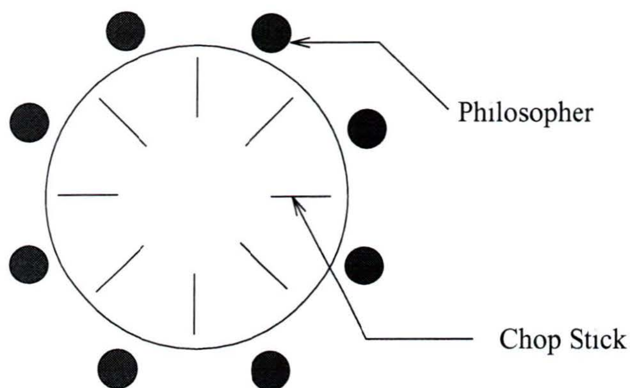


Figure 4.1: Eight Dining Philosophers

In designing the system we will have two main objects — a *chopstick* and a *philosopher*. A generic description of a chopstick could be

```
Stick(i) := up(i),down(i),Stick(i),
```

where `Stick(i)` represents the ith chopstick. Its main operations are requests to pick up the chopstick (`up(i)`) and to put down the chopstick (`down(i)`).

The definition of a *philosopher* is quite simple — pick up the right and left chopstick (in any order), eat and then put down the right and left chopstick (again, in any order).

```
Phil(i,n)  := (`up(i) | `up(i+1 mod n)),
              tau,
              (`down(i) | `down(i+1 mod n)),
              Phil(i,n).
```

The agent `Phil(i,n)` represents the ith philosopher around a table of `n`. Since the choice of picking up the left or right chopstick is independent, the parallel operator lends itself well. The same applies to putting the chopstick down. The `tau` represents the *eating* portion of the philosopher's duties. After putting the chopstick down, the philosopher can start again.

Now a generic table of `n` philosophers and chopsticks can be defined recursively as follows:

```
Table(n,n)   = SKIP
Table(i,n)   = ( Stick(i) | Phil(i,n) | Table(i+1,n) ) if i < n.
Table(n)     = Table(0,n)\{up,down}
```

The agent `Table(i,n)` represents the ith component of the table, and each component consists of a philosopher and a chopstick. `Table(n)` denotes the table of `n` philosophers and chopsticks. The restriction ensures that no external entity, other than a philosopher at the table, can pick up or put down a chopstick.

The dining philosopher problem is now just a table of size `n`.

```
DiningPhils(n)   = Table(n).
```

Now consider a table with two dining philosophers. If both request the left chopstick (i.e. philosopher 0 performs `'up(0)` and philosopher 1 performs `'up(1)`), neither can get the right chopstick (i.e. philosopher 0 cannot perform `'up(1)` and philosopher 1 cannot perform `'up(0)`). The sequence of events below illustrate this, where the command `strace` single steps through the transitions of a process expression. At each level the available transitions are displayed as numbered options. To distinguish the various internal communications that are possible, a parameter has been added to `tau` to indicate the action performing the communication/synchronization.

```
[> strace(DiningPhils(2))
```

```
**step tracing DiningPhils(2)
Level 0
1 ---tau(up(0))--> (down(0)  | up(0+1 mod 2) |
                   (Stick(1) | Phil(1,2)))\{up,down}
2 ---tau(up(0))--> (down(0)  | Phil(0,2) |
                   (Stick(1) | up(1)   ))\{up,down}
3 ---tau(up(1))--> (Stick(0) | Phil(0,2) |
                   (down(1)  | up(1+1 mod 2)  ))\{up,down}
4 ---tau(up(1))--> (Stick(0) | up(0)   |
                   (down(1)  | Phil(1,2)))\{up,down}
Which? 1
```

The first possible action of the system is to perform one of four communications. Either philosopher may pick up either chopstick. The first choice represents philosopher 0 picking up chopstick 0. After performing the first transition there are only two options left — either philosopher 0 or philosopher 1 may pick up the last chopstick. If philosopher 1 picks up the last chopstick we reach a deadlock situation as illustrated here.

```
Level 1 tau(up(0))
1 ---tau(up(1))--> (down(0)  | 'down(0)   |
                   (down(1)  | Phil(1,2)))\{up,down}
2 ---tau(up(1))--> (down(0)  | up(0+1 mod 2)   |
                   (down(1)  || up(1+1 mod 2)  )\{up,down}
Which? 2
Level 2 tau(up(0)),tau(up(1))
  <no transitions possible>
Which?
```

Rather than stepping through the derivation tree of a system, we can test certain path expressions. The deadlock situations could be checked by the path expression

```
(_*2),STOP
```

A sample trace of the results are given below. It is possible to locate every such trace that will end up in deadlock. The command `ssim` performs the strong simulation of a process expression against a path expression.

```
[> ssim(DiningPhils(2),(_*2),STOP)
**simulating  DiningPhils(2)
with path     (_*2),STOP
trace  tau(up(0)),tau(up(1))
Agent  (down(0),Stick(0) |
         up(0+1 mod 2),('down(0) | 'down(0+1 mod 2)),Phil(0,2)|
         (down(1),Stick(1) |
  up(1+1 mod 2),('down(1) | 'down(1+1 mod 2)),Phil(1,2) |
          Table(1+1,2)))\{up,down}
?;
```

One of the possible execution paths that satisfies the deadlock path expression is having chopstick 0 picked up by one philosopher, then having chopstick 1 picked up by the other philosopher, as represented by the above simulation. What is relevant is not whether philosopher 0 picks up chopstick 0 or chopstick 1, but only that both philosophers pick up one chopstick. At this point the path expression has been satisfied, however, the user has the option of examining the second of two execution paths that lead to deadlock — having chopstick 1 picked up before chopstick 0.

```
trace  tau(up(1)),tau(up(0))
Agent  (down(0),Stick(0) |
         up(0+1 mod 2),('down(0) | 'down(0+1 mod 2)),Phil(0,2) |
```

```
        (down(1),Stick(1) |
         up(1+1 mod 2),('down(1) | 'down(1+1 mod 2)),Phil(1,2) |
         Table(1+1,2)))\{up,down}
?,
```

The number 2 in (_*2),STOP for the number of repetitions is arbitrary, but should be sufficiently large to detect a deadlock For instance, for n dining philosophers use n as the number

An even shorter test is to use the notion of *weak* simulation, as in the following example which will immediately detect a deadlock

```
[> wsim(DiningPhils(2),STOP)
**weak simulating DiningPhils(2)
with path STOP
trace []
Agent DiningPhils(2)
```

The trace for weak simulation does not include any `taus` Therefore, the test does not show the execution path to deadlock, but rather that deadlock does exist

Often deadlock is not a desired property Although the specification does satisfy the vague initial requirements, it does not satisfy other specific requirements, such as

1 at any given time, some philosopher *must* be capable of eating (free of deadlock)

2 every philosopher must eventually eat, and

3 no single philosopher will eat more often than any other (fairness)

To achieve these requirements we will need to redesign the specification First, in order to meet the requirements, a scheduler is needed to determine when a philosopher is

capable of picking up the chopsticks. No specific order is required, but to be fair all philosophers must have had the chance to eat before any other can eat again. To ensure that no deadlock occurs, only one philosopher can be scheduled at any given time.

```
Semaphore      = REPEAT('take,give)
Schedule(0)    = SKIP
Schedule(n)    = (take,go(i),'done(i),'give | Schedule(n-1)
                    if (n>0)
Scheduler(n)    = (REPEAT(Schedule(n)) | Semaphore)\{take,give}
```

The agent `Semaphore` represents the usual *binary semaphore* primitive, with the operations `take` to reserve the semaphore, and `give` to relinquish the semaphore. The agent `Schedule(n)` represents a random scheduling of `n` objects, in this case, philosophers. The semaphore primitives are used to ensure that only one philosopher can be scheduled at a time. The action `go(i)` signals the ith philosopher to start eating and the action `'done(i)` indicates that the ith philosopher is done. The agent `Scheduler(n)` represents the repeated scheduling of the `n` philosophers.

There are two requirements that the scheduler must satisfy.

1. Only one item may be scheduled at any given time. A possible path expression that would disprove this is as follows:

   ```
   go(i),(!{done(i)})*10,go(j)
   ```

   This expression states that there exists a process execution path that contains two `go()` actions with no `done()` action between them. The length of the process execution path is at most 10. If an execution path greater than 10 is suspected to contain such a pattern, the number may be increased.

2. No item may be scheduled more often than the other items (fairness). There is no single simple general path expression that can be used to disprove this requirement. However, a collection of expressions of the form,

```
go(i),('{go(0)})*10,go(i)
...
go(i),('{go(i-1)})*10,go(i)
go(i),('{go(i+1)})*10,go(i)
...
go(i),('{go(n)})*10,go(i)
```

for all values of i, will disprove it. If any single path expression represents a valid process execution path, then the requirement is not met.

The philosopher must be modified such that he waits to be scheduled before picking up the chopsticks, and such that he must signal completion once the chopsticks have been picked up. In addition, the silent action tau, used to represent eating, will be replaced with the action eat(i) in order to observe when a particular philosopher has eaten.

```
Phil(i,n)  = 'go(i),
             ('up(i) | 'up(i+1 mod n)),
             done(i),
             eat(i),
             ('down(i) | 'down(i+1 mod n)),
             Phil(i,n)
```

Notice that by redefining the previous Phil(i,n) agent, the table definition Table(n) does not require redesigning. This new definition simply replaces the previous one.

The new dining philosopher can now be defined as a table of `n` philosophers and a scheduler of `n` objects

```
DiningPhils2(n)  = (Table(n) | Scheduler(n) )\{go,done}
```

Although this example is quite simplistic, it demonstrates how design flaws can be found and how desired properties can be verified

4.2 Mobile Robots

The example in this section is taken from [JF93] The idea is to design an autonomous robot that can manoeuvre on its own The robot is composed of the following devices

- a front and back bumper,
- two infrared sensors
- two photo-cells and
- two motors

A schematic diagram of the robot is given in Figure 4 2

The design incorporates the subsumption architecture The main idea is that there are no explicit geometric representations of the world from which the robot plans its actions Instead, there are a number of control loops for each action or behavior Each behavior is ranked in conjunction with other behaviors When two contradictory behaviors are triggered at the same time, the higher ranked one suppresses the lower one When the higher ranked behaviors are no longer activated, the lower ranking behaviors may resume control

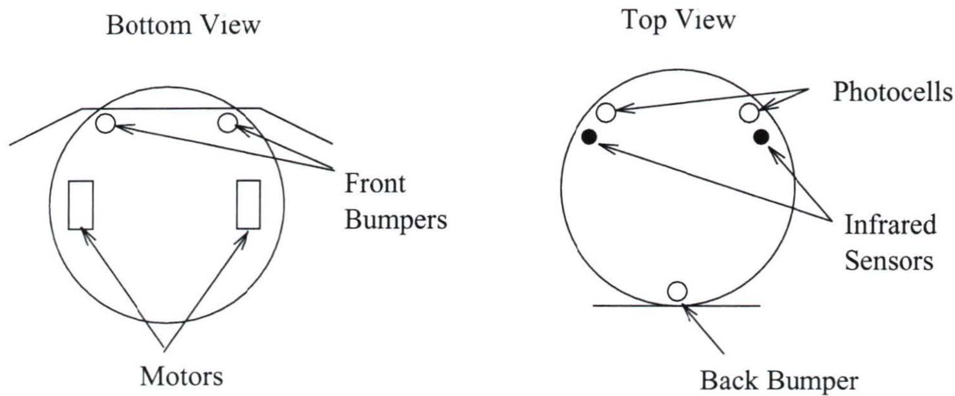


Figure 4.2: Schematic Diagram of Robot

We will consider a robot with four behaviors.

1. *cruise*: To constantly force the robot in the forward direction.
2. *follow*: To monitor the two photo-cells and follows the direction of brighter intensity.
3. *avoid*: Based on the infrared sensor information, to avoid contact with any obstacles.
4. *escape*: If the infrared sensors fail and the robot bumps into to object, to activate the *escape* behavior.

From lowest to highest, the behaviors are ranked as follows: *cruise*, *follow*, *avoid*, and *escape*. The interactions of the behaviors are depicted in Figure 4.3, where the rounded boxes represent the physical sensors and control drivers, the square boxes represent the actual behaviors, and the circles represent the suppressor node where the top behavior suppresses the lower behaviors.

Each behavior will be described in detail below.

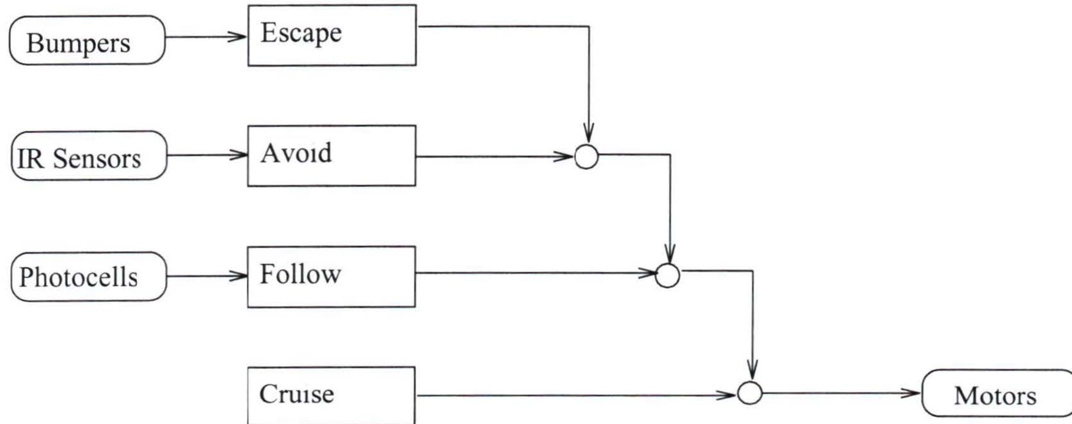


Figure 4 3 Robot Subsumption Architecture

Cruise The *cruise* behavior is straightforward, its purpose is to ensure that the robot is always moving The command sequence is issue a *forward* command, do some internal computation (i e possibly delay for some time), and repeat The definition is as follows

```
Cruise  = 'cruise_cmd('forward'),tau,Cruise
```

Follow The *follow* behavior monitors the two photo-cells, one of which is at the front left corner, the other at the front right corner When *follow* detects a difference between the two intensities which is greater than a given amount, it will issue a command to follow the more intense light If there is no discernible difference in intensity, *follow* will do nothing A description of the behavior is as follows

```
Follow  = photo_detect(val),F(val),tau,Follow
F('photo_none')   = 'follow_cmd('nothing')
F('photo_left')   = 'follow_cmd('left_turn')
F('photo_right')  = 'follow_cmd('right_turn')
```

`Follow` represents the agent that follows the path of greatest illumination, given that one photo-cell's intensity exceeds the other's. The action `photo_detect(val)` represents the difference between the two photo-cells. `val` is either `'photo_right'`, `'photo_left'`, or `'photo_none'`, indicating that either the right is more intense, left is more intense, or both have similar intensities. The agent `F(x)` issues the appropriate command based on the intensity difference `x`. If the right is more intense (`'photo_right'`) then turn right, similarly, if the left is more intense (`'photo_left'`) then turn left. If neither is the case, do nothing.

Avoid. *Avoid* handles the detection of obstacles from the infrared sensors. If the left sensor senses an obstacle, it will turn the robot to the right, if the right sensor senses an obstacle it will turn the robot to the left. Furthermore, the amount the robot turns is not significant, it tries to move the robot in an arc. If both sensors detect an obstacle it arbitrarily chooses to turn left. A description of this behavior is as follows:

```
Avoid  =ir_detect(val),A(val),tau,Avoid
A('both_ir')   = 'avoid_cmd('left_turn')
A('left_ir')   = 'avoid_cmd('left_arc')
A('right_ir')  = 'avoid_cmd('right_arc')
A('none_ir')   = 'avoid_cmd('nothing')
```

`Avoid` denotes the *avoid* behavior. It reads the infrared sensors (`ir_detect(val)`), issues an appropriate command (`A(val)`), performs some internal computation (`tau`) such as a possible delay, and then repeats. The value received from the sensors is assumed to be a string indicating which sensor(s) detected an object. The agent `A(x)` actually issues the commands.

Escape. If the infrared sensors fail to detect an obstacle and the robot runs into them, the bumpers will activate the *escape* behavior. Its main goal is to move the robot away from any obstacles. As previously mentioned, the escape commands are of highest priority. If both the left and right sides of the front bumper make contact, it will back up the robot and arbitrarily turn it to the left. If only the left front corner makes contact it will turn the robot right. If only the right front corner makes contact it will turn the robot left. Finally, if the back bumper makes contact it will command the robot to move forward. This behavior is described as follows.

```
Escape    = bump(val),E(val),tau,Escape
E('bump_lr')    = 'escape_cmd('backward'),tau,
                  'escape_cmd('left_turn')
E('bump_l')     = 'escape_cmd('right_turn')
E('bump_r')     = 'escape_cmd('left_turn')
E('bump_b')     = 'escape_cmd('left_turn')
E('bump_none') = 'escape_cmd('nothing')
```

The agent `Escape` is similar to the other behaviors, it requests the status of the bumpers (`bump(val)`), handles the type of bumper contact appropriately (`E(val)`), does some internal activity (`tau`), then repeats the whole process. Notice that in the case where both the left and right sides of the front bumper make contact (`E('bump_lr')`), two commands are issued with a possible delay or some other activity in between.

Arbitrator. Now that we have all the behaviors defined we need to design the *subsumption*, or the *suppressor nodes* of the system. One method is to design a global *arbitrator* that will handle the priority of the commands appropriately. In this case, its functionality is quite simple. If `Escape` issues a valid command, pass on that com-

mand, discarding the others. If Escape issues a null command 'escape_command('nothing'), and Avoid issues a valid command, then pass Avoid's message on to the motor. Similarly, the commands from Follow are passed on only if Escape and Avoid issue the null command, and the commands from Cruise are passed on to the motor only if the other three behaviors issue a null command. A description of the arbitrator is given as follows.

```
Arbitrate  = Arb('nothing','nothing','nothing','nothing')
Arb(c,f,a,e)  = cruise_cmd(c1),Arb(c1,f,a,e)
Arb(c,f,a,e)  = follow_cmd(f1),Arb(c,f1,a,e)
Arb(c,f,a,e)  = avoid_cmd(a1),Arb(c,f,a1,e)
Arb(c,f,a,e)  = escape_cmd(e1),Arb(c,f,a,e1)
Arb(c,f,a,e)  = ('motor_cmd(e),Arb(c,f,a,e)) if (e <> 'nothing')
Arb(c,f,a,'nothing')  = ('motor_cmd(a),Arb(c,f,a,'nothing'))
                   if (a != 'nothing')
Arb(c,f,'nothing','nothing')  =
  ('motor_cmd(f),Arb(c,f,'nothing','nothing')) if (f <> 'nothing')
Arb(c,'nothing','nothing','nothing')  =
    'motor_cmd(c),Arb(c,'nothing','nothing','nothing')
```

The agent Arb(c,f,a,e) represents the current state of the arbitrator, where the parameters c, f, a, and e are the last commands issued by the behaviors Cruise, Follow, Avoid and Escape, respectively. The first four definitions handle the inputs from the behaviors, and the last four handle the outputs to the motors. Arbitrate is the agent that denotes the actual *arbitrator*, starting in the initial state, where all behaviors have a null command ('nothing'). Notice that the commands *linger* until another command is issued to clear or change the command. Consequently, each behavior must issue a command even if it has nothing useful to do.

Controller System The controller portion of the system may be defined as a collection of processes all running independently

```
Controller  = (Arbitrate | Avoid | Escape | Follow | Cruise)
              \{escape_cmd,cruise_cmd,avoid_cmd,follow_cmd}
```

The only visible action of the agent `Controller` is `'motor_command()`

Environment The environment is assumed to consist of

- a photo-cell system that indicates which photo-cell is more intense through the output action `'photo_detect(val)`,
- a bumper system that indicates which bumper made contact, if any, through the output action `'bump(b)`,
- an infrared system that indicates which infrared sensor detected an obstacle through the output action `'detect_ir(val)`, and
- a motor controller system that takes as input a command to move in a certain direction though the input action `motor_command()`

We will present a very abstract description of the environment entities to show how a complete system may be described

Bumper The *bumper* entity produces the output action `'bump()` used by the `Avoid` agent There are five possible outputs

- `'bump_lr'` — both the left and right sensors of the front bumper detect contact,
- `'bump_l'` — only the left sensor of the front bumper detected contact,

- 'bump_r' — only the right sensor of the front bumper detected contact,
- 'bump_b' — only the back bumper detected contact, and
- 'bump_none' — no contact was detected by any bumper.

An abstract description of the bumper entity is given as follows:

```
Bumpers  = REPEAT(('bump('bump_lr') + 'bump('bump_l') +
                  'bump('bump_r' + 'bump('bump_b') +
                  'bump('bump_none'),tau ).
```

Infrared sensors. The infrared sensing system produces the output action 'detect_ir() with four possible parameter values:

- 'both_ir' — both of the infrared sensors are blocked,
- 'left_ir' — only the left of the infrared sensors is blocked,
- 'right_ir' — only the right of the infrared sensors is blocked, and
- 'none_ir' — neither of the infrared sensors are blocked.

An abstract description is given as follows:

```
Infrared  = REPEAT(('detect_ir('both_ir') + 'detect_ir('left_ir') +
                   'detect_ir('right_ir') + 'detect_ir('none_ir')),
                  tau ).
```

Photo-cells. The photo-cell system produces the output action 'photo_detect() with three possible parameter values.

- `'photo_none'` — the intensities of both photo-cells are similar,
- `'photo_left'` — the left photo-cell is more intense than the right, and
- `'photo_right'` — the right photo-cell is more intense than the left.

An abstract description is given as follows:

```
PhotoCells  = REPEAT((‘photo_detect('photo_none') +
                     ‘photo_detect('photo_left') +
                     ‘photo_detect('photo_right'),tau )
```

Motors. The motor system is responsible for driving the two motors in the appropriate direction to achieve the desired route. The abstract motor system accepts commands from the input action `motor_command()` and then sends the appropriate command(s) to the two motors. For our abstraction, the output action will represent this movement, and will be used to trace the robot's movements. The possible commands for the motor are:

- `'right_turn'` — turn the robot sharp right,
- `'right_arc'` — turn the robot in a gradual arc to the right,
- `'left_turn'` — turn the robot sharp left,
- `'left_arc'` — turn the robot in a gradual arc to the left,
- `'backward'` — move the robot backwards,
- `'forward'` — move to robot forwards, and
- `'stop'` — stop the robot.

```
Motors  = REPEAT(motor_command(c),'move(c))
```

The environment can now be specified as follows

```
Env  = (Motors | PhotoCells | Infrared | Bumpers)
```

Complete system The complete robot system can be specified as the controller composed with an environment

```
RobotSystem  = (Controller | Env)\{motor_command, ,photo_detect,
                                       detect_ir, bump}
```

The only observable action of the complete system is the output action `'move()`

Chapter 5

Implementation

In this chapter, we describe a Prolog implementation of the ACS language, CPE path expressions, and the simulation definitions introduced in this thesis. The reasons for choosing Prolog are:

1. The nondeterminism and backtracking capabilities of Prolog allow the semantics of ACS and CPE to be directly translated into Prolog.

2. A major characteristic of Prolog programming is that it allows the programmer to concentrate on specifying *what* the problem is, rather than *how* to solve it. The nondeterministic nature of the operational semantics would make the use of an imperative programming language very tedious.

5.1 Optimizations

The operational semantics defined in Chapter 2 are inherently inefficient. The main area of inefficiency is in the restriction formula. The recursive nature of the semantics

would cause a large amount of backtracking until a transition is found that is not in the restriction set. For example, the expression

```
P  = ( a | 'a)\{a}
```

might lead to trying the following transition sequences:

$$1.\quad \cfrac{\cfrac{\mathtt{a} \xrightarrow{\mathtt{a}} (\mathtt{SKIP}, \emptyset)}{\mathtt{a|`a} \xrightarrow{\mathtt{a}} (\mathtt{SKIP|`a}, \emptyset)}}{(\mathtt{a|`a})\backslash\{\mathtt{a}\} \stackrel{\mathtt{a}}{\not\longrightarrow}} \qquad 2.\quad \cfrac{\cfrac{\mathtt{`a} \xrightarrow{\mathtt{`a}} (\mathtt{SKIP}, \emptyset)}{\mathtt{a|`a} \xrightarrow{\mathtt{`a}} (\mathtt{a|SKIP}, \emptyset)}}{(\mathtt{a|`a})\backslash\{\mathtt{a}\} \stackrel{\mathtt{`a}}{\not\longrightarrow}}$$

$$3.\quad \cfrac{\cfrac{\mathtt{a} \xrightarrow{\mathtt{a}} (\mathtt{SKIP}, \emptyset) \quad \mathtt{`a} \xrightarrow{\mathtt{`a}} (\mathtt{SKIP}, \emptyset)}{\mathtt{a|`a} \xrightarrow{\mathtt{tau}} (\mathtt{SKIP|SKIP}, \emptyset)}}{(\mathtt{a|`a})\backslash\{\mathtt{a}\} \xrightarrow{\mathtt{tau}} ((\mathtt{SKIP|SKIP})\backslash\{\mathtt{a}\}, \emptyset)}$$

Notice that the first two sequences both have two extra levels of recursion that are unnecessary. An optimization to the operational semantics is to carry a restriction set such that at the base cases (i.e. actions), the transition is prohibited if it is in the restriction set. However, in doing so, the relabeling information is required as well. The new operational semantics is based on an environment, together with the original transition relation. The expression

$$\langle \Delta, \Phi \rangle \vdash P \xrightarrow{\alpha} (P', \theta)$$

means: given the environment $\langle \Delta, \Phi \rangle$, where Δ is the restriction set and Φ is the relabeling function, and process expression P, the transition $P \xrightarrow{\alpha} (P', \theta)$ is possible. Referring to the previous example, the complete initial transition sequence would be

as follows:

$$1.\quad \frac{\dfrac{\langle\{\mathtt{a}\}, Id\rangle \vdash \mathtt{a} \overset{\mathtt{a}}{\not\longrightarrow}}{\langle\{\mathtt{a}\}, Id\rangle \vdash \mathtt{a\,|\,`a} \overset{\mathtt{a}}{\not\longrightarrow}}}{\langle\emptyset, Id\rangle \vdash (\mathtt{a\,|\,`a})\backslash\{\mathtt{a}\} \overset{\mathtt{a}}{\not\longrightarrow}} \qquad 2.\quad \frac{\dfrac{\langle\{\mathtt{a}\}, Id\rangle \vdash \mathtt{`a} \overset{\mathtt{`a}}{\not\longrightarrow}}{\langle\{\mathtt{a}\}, Id\rangle \vdash \mathtt{a\,|\,`a} \overset{\mathtt{`a}}{\not\longrightarrow}}}{\langle\emptyset, Id\rangle \vdash (\mathtt{a\,|\,`a})\backslash\{\mathtt{a}\} \overset{\mathtt{`a}}{\not\longrightarrow}}$$

$$3.\quad \frac{\dfrac{\langle\emptyset, Id\rangle \vdash \mathtt{a} \overset{\mathtt{a}}{\longrightarrow} (\mathtt{SKIP}, \emptyset) \quad \langle\emptyset, Id\rangle \vdash \mathtt{`a} \overset{\mathtt{`a}}{\longrightarrow} (\mathtt{SKIP}, \emptyset)}{\langle\{\mathtt{a}\}, Id\rangle \vdash \mathtt{a\,|\,`a} \overset{\mathtt{tau}}{\longrightarrow} (\mathtt{SKIP\,|\,SKIP}, \emptyset)}}{\langle\emptyset, Id\rangle \vdash (\mathtt{a\,|\,`a})\backslash\{\mathtt{a}\} \overset{\mathtt{tau}}{\longrightarrow} ((\mathtt{SKIP\,|\,SKIP})\backslash\{\mathtt{a}\}, \emptyset)}$$

Notice that failures in the first two sequences occur at the bottom level rather than at the invocation of the restriction rule.

Relating this to the operational semantics in Chapter 2, we have the following relationship:

$$P \overset{\alpha}{\longrightarrow} (P', \theta) \text{ iff } \langle\emptyset, Id\rangle \vdash P \overset{\alpha}{\longrightarrow} (P', \theta),$$

where Id is the *identity* function. The complete optimized operational semantics are given in Figure 5.1.

The actual restriction and renaming is done in the **In** and **Out** rules. The restriction rule (**Res**) adds to the restriction set. The relabeling rule (**Ren**) applies the inverse function (f^{-1}) to the restriction set, since the set applies to the renamed labels, and uses the composition of functions ($f \circ \Phi$) as the relabeling function. Function composition $f \circ \Phi$ means the usual $\Phi(f(\alpha))$. Furthermore, the communication rule (**Com3**) does not require any environment for its condition transitions, since a communication (`tau`) is not affected by a relabeling or restriction combinator.

The translation from the semantic rules to actual Prolog implementation is given in the next section.

$$\textbf{In}\quad \frac{}{\langle\Delta,\Phi\rangle\vdash a(\vec{e}) \xrightarrow{\Phi(a(\vec{v}))} (\texttt{SKIP},\theta)}\; a\notin\Delta,\ [\![\vec{e}\theta]\!]=\vec{v}$$

$$\textbf{Out}\quad \frac{}{\langle\Delta,\Phi\rangle\vdash \text{‘}a(\vec{e}) \xrightarrow{\text{‘}\Phi(a(\vec{v}))} (\texttt{SKIP},\theta)}\; (a)\notin\Delta,\ [\![\vec{e}\theta]\!]=\vec{v}$$

$$\textbf{Tau}\quad \frac{}{\langle\Delta,\Phi\rangle\vdash \texttt{tau} \xrightarrow{\texttt{tau}} (\texttt{SKIP},\emptyset)}$$

$$\textbf{Seq1}\quad \frac{\langle\Delta,\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\texttt{,}Q \xrightarrow{\alpha} (P'\texttt{,}Q,\theta)} \qquad \textbf{Seq2}\quad \frac{\langle\Delta,\Phi\rangle\vdash Q \xrightarrow{\alpha} (Q',\theta)}{\langle\Delta,\Phi\rangle\vdash P\texttt{,}Q \xrightarrow{\alpha} (Q',\theta)}\; P{\downarrow}$$

$$\textbf{Sum}\quad \frac{\langle\Delta,\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\texttt{+}Q \xrightarrow{\alpha} (P',\theta)}$$

$$\textbf{If}\quad \frac{\langle\Delta,\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\ \texttt{if}\ be \xrightarrow{\alpha} (P',\theta)}\; [\![be]\!]=\texttt{true}$$

$$\textbf{Ren}\quad \frac{\langle f^{-1}(\Delta), f\circ\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\texttt{\#}f \xrightarrow{\alpha} (P'\texttt{\#}f,\theta)} \qquad \textbf{Res}\quad \frac{\langle\Delta\cup L,\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\backslash L \xrightarrow{\alpha} (P'\backslash L,\theta)}$$

$$\textbf{Com1}\quad \frac{\langle\Delta,\Phi\rangle\vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash P\,\texttt{|}\,Q \xrightarrow{\alpha} (P'\,\texttt{|}\,Q,\theta)} \qquad \textbf{Com2}\quad \frac{\langle\Delta,\Phi\rangle\vdash Q \xrightarrow{\alpha} (Q',\theta)}{\langle\Delta,\Phi\rangle\vdash P\,\texttt{|}\,Q \xrightarrow{\alpha} (Q',\theta)}\; P{\downarrow}$$

$$\textbf{Com3}\quad \frac{\langle\emptyset,Id\rangle\vdash P \xrightarrow{\text{‘}a(\vec{v})} (P',\emptyset),\ \langle\emptyset,Id\rangle\vdash Q\theta \xrightarrow{a(\vec{v})} (Q',\theta)}{\langle\Delta,\Phi\rangle\vdash P\,\texttt{|}\,Q \xrightarrow{\tau} (P'\,\texttt{|}\,Q',\theta)}$$

$$\textbf{Defn1}\quad \frac{\langle\Delta,\Phi\rangle\vdash P\theta_1 \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash A(\vec{e_1}) \xrightarrow{\alpha} (P',\theta)}\; A(\vec{e_2}) = P,\ [\![\vec{e_1}]\!]=\vec{v},\ [\![\vec{e_2}\theta_1]\!]=\vec{v}$$

$$\textbf{Defn2}\quad \frac{\langle\Delta,\Phi\rangle\vdash P\theta_1 \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle\vdash O(\vec{p_1}) \xrightarrow{\alpha} (P',\theta)}\; O(\vec{p_2}) = P,\ [\![\vec{p_1}]\!]=\vec{v},\ [\![\vec{p_2}\theta_1]\!]=\vec{v}$$

Figure 5 1 Efficient Operational Semantics for ACS

5.2 System Description

5.2.1 Expression Parsing

All syntactic expressions, whether ACS, CPE, or value expressions, are parsed into either unary or binary terms of the form

`op(oper,term)` or `op(oper,term1,term2)`

For example, the expression `a(x),((‘b if x < 0)) + c)` would translate into the following nested Prolog term

```
op(',',func(lid('a'),1,[X]),
        op('+',op(if,op('‘',lid('b')),op('<',X,num(0)),lid('c')))
```

The term `func(lid('a'),1,[X])` is used to express parameterized actions, where `X` is a Prolog variable; the first parameter of `func()` is the action label, the second the arity, and the third is a list of the formal parameter expressions, in this case, simply a variable. Notice that even the output action indicator (‘) is treated as a unary operator. Agent and operator definitions are stored as Prolog **defn**/3 clauses, where the first parameter is the agent identifier, the second is the actual expression, and the third parameter is a list of all the variable names associated to actual Prolog variables. For example, the agent definition

```
P(x)   = a(x),SKIP
```

is stored as the Prolog clause

```
defn(func(uid('P'),1,[X]),
      op(',',func(lid('a'),1,[X]),'SKIP'), [x=X])
```

Parameterized agents and user-defined operators are defined similarly as parameterized actions.

5.2.2 Implementation of ACS Operational Semantics

The operational semantics of the ACS process algebra translate directly into Prolog terms. The operational semantic expression

$$\langle \Delta, \Phi \rangle \vdash P \xrightarrow{\alpha} (P', \theta)$$

is translated into the Prolog relation

```
trans(P, Res, Ren, A, Q1),
```

where `Res` $= \Delta$, `Ren` $= \Phi$, `A` $= \alpha$, and `Q1` $= P'\theta$. The translation from the formal optimized operational semantics of ACS to Prolog clauses is described in detail, avoiding giving low level implementation details. All the source code for the given implementation is presented in Appendix C.

In. The input action semantic rule

$$\frac{}{\langle \Delta, \Phi \rangle \vdash a(\vec{e}) \stackrel{\Phi(a(\vec{v}))}{\longrightarrow} (\mathtt{SKIP}, \theta)} \; a \notin \Delta, \; [\![\vec{e}\theta]\!] = \vec{v}$$

is translated into the following clause:

```
trans( A, Res, Ren, B, 'SKIP') :-
          input_action(A),
          !, is_not_in(A, Res),
          eval(A,A1),
          renaming(A1, Ren, B).
```

The `input_action(A)` relation holds if `A` is an input action, and the `is_not_in(A, Res)` relation ensures that `A` is *not* in the restriction set `Res` and corresponds to the expression $a \notin \Delta$ of the above semantic rule. The *cut* (!) after the the `input_action` relation ensures that no other `trans/5` clause is evaluated. The relation `eval(A,A1)` performs the value expression semantic defined in Appendix A and corresponds to the expression $[\![\vec{e}\theta]\!] = \vec{v}$. For simplicity, the evaluation is applied to any process expression. Finally, `renaming(A1, Ren, B)` renames the action `A1` to `B` using the relabeling function `Ren`.

Out The **Out** semantic rule

$$\frac{}{\langle \Delta, \Phi \rangle \vdash `a(\vec{e}) \overset{`\Phi(a(\vec{v}))}{\longrightarrow} (\mathsf{SKIP}, \theta)} \; (a) \notin \Delta, \; [\![\vec{e}\theta]\!] = \vec{v}$$

is translated into the following clause:

```
trans( op('`',A), Res, Ren, op('`',B), 'SKIP') :-
        !, input_action(A),
        is_not_in(A, Res),
        eval(A,A1),
        renaming(A1, Ren, B).
```

An output action is expressed as a binary operator term, where the output action indicator (`) is the binary operator and the input action is the operand.

Tau The **Tau** semantic rule

$$\frac{}{\langle \Delta, \Phi \rangle \vdash \mathtt{tau} \overset{\mathtt{tau}}{\longrightarrow} (\mathsf{SKIP}, \emptyset)}$$

is translated into the following clause:

```
trans( tau, _, _, tau, 'SKIP') :- !.
```

If The **If** semantic rule

$$\frac{\langle \Delta, \Phi \rangle \vdash P \xrightarrow{\alpha} (P', \theta)}{\langle \Delta, \Phi \rangle \vdash P \texttt{ if } be \xrightarrow{\alpha} (P', \theta)} \; [\![be]\!] = \texttt{true}$$

is translated into the following clause

```
trans( op( if, P, Preds), Res, Ren, A, P1) :-
        !, eval(Preds, bool(true) ),
        trans( P, Res, Ren, A, P1).
```

The predicate `eval(Preds, bool(true))` corresponds to the expression $[\![be]\!] = \texttt{true}$.

Seq1 The **Seq1** semantic rule

$$\frac{\langle \Delta, \Phi \rangle \vdash P \xrightarrow{\alpha} (P', \theta)}{\langle \Delta, \Phi \rangle \vdash P; Q \xrightarrow{\alpha} (P'; Q, \theta)}$$

is translated into the following clause:

```
trans( op(',',P,Q), Res, Ren, A, V ) :-
        trans(P, Res, Ren, A, P1),
        combine(op(',',P1,Q),V).
```

The predicate `combine/2` removes any unnecessary `SKIP` process expressions. For example, the expression `SKIP,a` would be reduced to the expression `a`. This simplifies an expression to its minimal form.

Seq2 The **Seq2** semantic rule

$$\frac{\langle \Delta, \Phi \rangle \vdash Q \xrightarrow{\alpha} (Q', \theta)}{\langle \Delta, \Phi \rangle \vdash P; Q \xrightarrow{\alpha} (Q', \theta)} \; P\downarrow$$

is translated into the following clause:

```
trans( op(',',P,Q), Res, Ren, A, Q1 ) :-
        !, terminates(P),
        trans(Q, Res, Ren, A, Q1).
```

The predicate `terminates(P)` is an implementation of the termination predicate and corresponds to the expression $P\downarrow$.

Sum The **Sum** semantic rule

$$\frac{\langle\Delta,\Phi\rangle \vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle \vdash P+Q \xrightarrow{\alpha} (P',\theta)}$$

is translated into the following two symmetric clauses:

```
trans( op('+',P,_), Res, Ren, A, P1) :-
        trans(P, Res, Ren, A, P1).
trans( op('+',_,Q), Res, Ren, A, Q1) :-
        !, trans(Q, Res, Ren, A, Q1).
```

Note that the ordering of these clauses is important. Only the last clause must have the cut (!) since both rules apply to the same combinator expression.

Com1 The **Com1** semantic rule

$$\frac{\langle\Delta,\Phi\rangle \vdash P \xrightarrow{\alpha} (P',\theta)}{\langle\Delta,\Phi\rangle \vdash P\,|\,Q \xrightarrow{\alpha} (P'\,|\,Q,\theta)}$$

is translated into the following two symmetric clauses:

```
trans( op('|',P,Q), Res, Ren, A, V ) :-
        trans(P, Res, Ren, A, P1),
```

```
        combine(op('|',P1,Q),V).
trans( op('|',P,Q), Res, Ren, A, V ) :-
        trans(Q, Res, Ren, A, Q1),
        combine(op('|',P,Q1),V).
```

Com3. The **Com3** semantic rule

$$\frac{\langle\emptyset, Id\rangle \vdash P \xrightarrow{'a(\vec{v})} (P', \emptyset),\ \langle\emptyset, Id\rangle \vdash Q\theta \xrightarrow{a(\vec{v})} (Q', \theta)}{\langle\Delta, \Phi\rangle \vdash P \,|\, Q \xrightarrow{\tau} (P' \,|\, Q', \theta)}$$

is translated into the following two symmetric clauses:

```
trans( op('|',P,Q), _, _, tau(A), V ) :-
        trans(P, [], [], op('`',A), P1),
        trans(Q, [], [], A, Q1),
        combine(op('|',P1,Q1),V).
trans( op('|',P,Q), _, _, tau(A), V ) :-
        trans( Q, [], [], op('`',A), Q1),
        trans( P, [], [], A, P1),
        combine(op('|',P1,Q1),V).
```

The first clause handles the case where the first expression of the parallel expression performs the output action, the last clause handles the case where the second expression performs the output action. In order to trace the actions that are communicating, the action is added as a parameter to the silent action `tau`.

Com2. The **Com2** semantic rule

$$\frac{\langle\Delta, \Phi\rangle \vdash Q \xrightarrow{\alpha} (Q', \theta)}{\langle\Delta, \Phi\rangle \vdash P \,|\, Q \xrightarrow{\alpha} (Q', \theta)}\ P\!\downarrow$$

is translated into the following two symmetric clauses:

```
trans( op('|',P,Q), Res, Ren, A, Q1 ) :-
        terminates(P),
        trans(Q, Res, Ren, A, Q1).
trans( op('|',P,Q), Res, Ren, A, P1 ) :-
        !, terminates(Q),
        trans(P, Res, Ren, A, P1).
```

Ren. The **Ren** semantic rule

$$\frac{\langle f^{-1}(\Delta), f \circ \Phi\rangle \vdash P \xrightarrow{\alpha} (P', \theta)}{\langle \Delta, \Phi\rangle \vdash P\#f \xrightarrow{\alpha} (P'\#f, \theta)}$$

is translated into the following clause:

```
trans( op('#',P,F), Res, Ren, A, V ) :-
        !, f_inverse(Res, F, NewRes),
        fcn_compose(F,Ren,NewRen),
        !, trans( P, NewRes, NewRen, A, P1),
        combine(op('#',P1,F),V).
```

The `f_inverse(Res, F, NewRes)` relation computes the functional inverse of the restriction set `Res`, and corresponds to the expression $f^{-1}(\Delta)$ in the above semantic rule. `fcn_compose(F, Ren, NewRen)` computes the functional composition of `F` and `Ren`, and corresponds to $f \circ \Phi$.

Res. The **Res** semantic rule

$$\frac{\langle \Delta \cup L, \Phi\rangle \vdash P \xrightarrow{\alpha} (P', \theta)}{\langle \Delta, \Phi\rangle \vdash P\backslash L \xrightarrow{\alpha} (P'\backslash L, \theta)}$$

is translated into the following clause

```
trans( op('\',P,L), Res, Ren, A, V ) :-
        !, union(Res,L,NewRes),
        !, trans(P, NewRes, Ren, A, P1),
        combine(op('\',P1,L),V)
```

The relation `union(Res, L, NewRes)` is the usual set union.

Defn1 and Defn2. The semantic rules for agent definitions (**Defn1**) and operator definitions (**Defn2**) are treated the same, so only one clause is required. These two semantic rules

$$\frac{\langle \Delta, \Phi \rangle \vdash P\theta_1 \stackrel{\alpha}{\longrightarrow} (P', \theta)}{\langle \Delta, \Phi \rangle \vdash A(\vec{e_1}) \stackrel{\alpha}{\longrightarrow} (P', \theta)} \; A(\vec{e_2}) = P,\ [\![\vec{e_1}]\!] = \vec{v},\ [\![\vec{e_2}\theta_1]\!] = \vec{v}$$

$$\frac{\langle \Delta, \Phi \rangle \vdash P\theta_1 \stackrel{\alpha}{\longrightarrow} (P', \theta)}{\langle \Delta, \Phi \rangle \vdash O(\vec{p_1}) \stackrel{\alpha}{\longrightarrow} (P', \theta)} \; O(\vec{p_2}) = P,\ [\![\vec{p_1}]\!] = \vec{v},\ [\![\vec{p_2}\theta_1]\!] = \vec{v},$$

both translate into the following single clause:

```
trans(P, Res, Ren, A, P1) :-
        eval(P, P2),
        defn(P2, R, _),
        trans(R, Res, Ren, A, P1).
```

As before, the clause `eval(P,P2)` corresponds to the expressions $[\![\vec{e_1}]\!] = \vec{v}$ and $[\![\vec{p_1}]\!] = \vec{v}$ of the above semantic rules, with the exception that `eval/2` accepts process expressions rather than just value expressions. The `defn(P2, R, _)` predicate call does two things: it produces the substitution θ_1 by Prolog's instantiation properties, and it recalls an appropriate agent or operator definition. Note that this clause handles the case where a certain agent (operator) identifier has multiple definitions.

The relationship of the new semantics to the semantics described in Chapter 2,

$$P \xrightarrow{\alpha} (P', \theta) \text{ iff } \langle \emptyset, Id \rangle \vdash P \xrightarrow{\alpha} (P', \theta)$$

translates directly into the trans/3 relation

```
trans(P, A, P1) :- trans(P, [], [], A, P1).
```

5.2.3 CPE Semantics

The implementation of the path expression semantics is straightforward, and similar to the ACS semantics. The path_term(S) relation denotes the path termination predicate, *term(S)*. The path deadlock predicate is denoted by the relation path_dead(S). The path transition rule

$$S \overset{A}{\rightsquigarrow} S_1$$

is implemented as the relation path_step(S, A, S1). (See Appendix C for the implementation details.)

5.2.4 Simulation

The last implementation detail to be discussed is the simulation of process expressions with path expressions. The six cases given in Definition 12 of Chapter 3 can be grouped into three categories:

1. successful termination,
2. unsuccessful termination (deadlock), and
3. single action step.

The (strong) simulation $P \vdash^s S$ is implemented as the relation `strong_simulate(P, S, Path, P1)` where `Path` represents the execution path of `P` that satisfies the path expression `S`, and `P1` is the residual process expression of `P` after completion of the execution path `Path`.

Successful termination. The implementation of the first case is straightforward. If the path terminates, then any process expression satisfies the path expression by not performing any action, and the simulation is complete.

```
strong_simulate(P, S, [], P) :-
        path_term(S).
```

Unsuccessful termination. In the second case, if the path expression deadlocks and the process expression cannot perform any action, then the relation holds and the simulation is complete.

```
strong_simulate(P, S, [], P) :-
no_path_step(S),
\+ trans(P, _, _).
```

Single action step. The four cases of single action steps of the definition of simulation can be captured by the following single clause:

```
strong_simulate(P, S, [A|As], P2) :-
path_step(S, A, S1),
trans(P, A, P1),
instantiate(A),
strong_simulate(P1, S1, As, P2).
```

The first two calls ensure that whatever action the path expression may perform, a matching transition for the process expression exists. The `instantiate(A)` relation prompts for the appropriate user interaction to instantiate any unknown variables, whether action or parameter variables. The last call is the recursive call to complete the simulation of the path expression.

The implementation of the weak simulation is similar and is given in detail in Appendix C.

Chapter 6

Concluding Remarks

6.1 Contributions

Process algebra is one of the formalisms used to analyze and verify the designs of systems, particularly concurrent systems. Unfortunately, although theoretically sound, complex design specifications in current process algebraic formalisms are not straightforward. The simplicity of the syntax often complicates the specifications of designs.

In this thesis, a design specification language, ACS, has been introduced. It incorporates the desirable features of Milner's value passing CCS [Mil89] and Bergstra and Klops's ACP [BK85]. In addition, less restrictions are enforced to allow more concise specifications of complex systems. The language addresses four main issues:

1. better variable scoping and binding,

2. full pattern matching of action and agent identifier parameters,

3. sequential composition of processes, and

4. the addition of user-definable operators

A formal operational semantics is defined for this language.

Next, a concurrent path expression language, CPE, has been introduced to automate the debugging and simulation of ACS design specifications. The language is quite similar to *data path expressions* presented in [HK90]. CPE expressions allow for verification of properties such as deadlock, eventuality, and fairness in a concise notation.

Finally, ACS and CPE have been implemented in a tool for automated simulation and debugging of designs. The tool is an interactive interpreter that allows the user to simulate ACS designs against CPE specifications and step through the execution paths of ACS designs. The tool is a step towards a prototyping environment with the addition of other verification techniques.

6.2 Future Work

Although the results obtained in this thesis are encouraging, there are many interesting additions to our work that should be further investigated. The most important is the notion of symbolic interpretations with respect to simulation and other notions of verification.

An important aspect of producing formal specifications is the possibility of analysis and verification of the design. One form of verification is proving two designs are equivalent. Equivalence is an often-used criterion for a correct refinement of a specification. There are a variety of notions of equivalences, some of which are summarized and compared in [vG90, vG93]. Amongst these, the most prominent one is *bisim-*

ulation equivalence [1] For finite state specifications, the check for bisimulation can be done automatically Unfortunately, once data is introduced, a specification may not be finite state since the data may have infinitely many values (or unmanageably many) A *symbolic* interpretation of ACS is required to reduce the number of transitions However, the *symbolic* semantics require not only an action as the transition but also a boolean predicate that must hold for that transition to be possible For example, consider the expression

```
P(x)  = ('a(x) if x>= 0) + ('b(x) if x<0)
```

which outputs nonnegative numbers on the a channel and negative numbers on the b channel In symbolic interpretations, the transitions, $\mathtt{P(x)} \xrightarrow{\mathtt{'a(x)}} \mathtt{SKIP}$ and $\mathtt{P(x)} \xrightarrow{\mathtt{'b(x)}} \mathtt{SKIP}$ are possible However, this is not true for all x, in fact, for any given number, only *one* transition is possible Consequently, a way of describing this is

$$\mathtt{P(x)} \overset{\mathtt{'a(x),x>=0}}{\longmapsto} \mathtt{SKIP}$$
$$\mathtt{P(x)} \overset{\mathtt{'b(x),x<0}}{\longmapsto} \mathtt{SKIP},$$

meaning that P(x) can perform the transition 'a(x) provided that the predicate x>=0 holds

Some work in the area of symbolic interpretations has already been done for other process algebras in [JP89],[Sch92], and [HL92] These, however, are limited by their extension to pure precess algebras, especially in the area of variable scoping and binding, and pattern-matching An extension of these concepts provides exciting possibilities for further research

[1] In Appendix B we introduce the bisimulation equivalence for ACS

Bibliography

[BK85] J A Bergstra and J W Klop Algebra of communicating processes with abstraction *Theoretical Computer Science*, 37 77–121, 1985

[Bru93] G Bruns A Language and Translator for value-passing CCS Technical report, University of Edinburgh, 1993

[BW90] J C M Baeten and W P Weijland *Process Algebra* Cambridge Tracts in Theoretical Computer Science Cambridge University Press, Great Britain, 18 edition, 1990

[Che92] Mantis H M Cheng A tutorial on process algebraic specification Technical Report DCS-199-IR, University of Victoria, July 1992

[Che94] M H M Cheng Calculus of Communicating Systems a synopsis unpublished paper, 1994

[GP90] J F Groote and A Ponse The syntax and semantics of μCRL Technical Report Technical Report CS-R9076, Centre for Mathematics and Computer Science (CWI), 1990

[GP91] J F Groote and A Ponse Proof theory for μCRL Technical Report Technical Report CS-R9138, Centre for Mathematics and Computer Science (CWI), 1991

[HK90] W. Hseuh and G. Kaiser. Modeling Concurreny in Parallel Debugging. In *2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 11–20, Seattle, Washington, 1990.

[HL92] M. Hennessy and H. Lin. Symbolic bisimulation. Technical Report Computer Science Report 92:01, University of Sussex, April 1992.

[HL93] M. Hennessy and H. Lin. Proof systems for message-passing process algebras. In Eike Best, editor, *CONCUR'93*, pages 202–216. Springer-Verlag, 1993. LNCS 715.

[Hoa78] Hoare, C.A.R. Communicating sequential process. *Communications of the ACM*, 21:666–677, 1978.

[Hoa85] Hoare, C.A.R. *Communicating Sequential Process*. Prentice Hall International, Hertfordshire, England, 1985.

[JF93] Joseph L. Jones and Anita M. Flynn. *Mobile Robots: Inspiration to Implementation*. A K Peters Ltd, Wellesley, MA, 1993.

[JP89] B. Jonsson and J. Parrow. Deciding bisimulation equivalences for a class of non-finite state programs. In *The 6th Symposium on Theoretical Aspects of Computer Science*, pages 421–433, 1989.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980. LNCS 92.

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

[Sch92] Zvi Schreiber. Verification and Analysis of value-passing CCS Programs with infinite sorts. Technical Report DoC 92/9, Imperial College, June 1992.

[vG90] R.J. van Glabbeek. The linear time – branching time spectrum. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90*, pages 278–297. Springer-Verlag, 1990. LNCS 458.

[vG93] R.J. van Glabbeek. The linear time – branching time spectrum ii. In Eike Best, editor, *CONCUR'93*, pages 66–81. Springer-Verlag, 1993. LNCS 715.

[Wal87] D. Walker. A Introduction to a Calculus of Communicating Systems. Technical report, University of Edinburgh, 1987.

Appendix A

Value Expressions

This appendix formally describes the syntax and semantics of the value expressions used in ACS

A.1 Syntax

Let $\mathcal{D}$ denote the basic data values, which include *numbers*, *booleans*, *constants* (optionally parameterized with elements in $\mathcal{D}$) and *lists* of elements in $\mathcal{D}$. v ranges over $\mathcal{D}$. Formally, $\mathcal{D}$ is defined as follows:

Definition 15 Let n, b, and c denote numbers, booleans, and constants, respectively. Then $\mathcal{D}$ is the set generated by the following grammar, where $v_1, \ldots, v_n \in \mathcal{D}$:

$$v ::= n \mid b \mid c \mid c(v_1, \ldots, v_n) \mid [v_1, \ldots, v_n]$$

□

Value expressions include binary and unary operator expressions (usual arithmetic, boolean and relational expressions, as well as list membership, concatenation and

subtraction), lists of value expressions, and parameterized constants.

Definition 16 Let $\mathcal{V}$ be the set of data *variables* denoted by x. Let v range over $\mathcal{D}$. The set $\mathcal{E}$ of *value expressions* with typical element e is the set generated by the following grammar, where $e_1, \ldots, e_n \in \mathcal{E}$:

$$
\begin{array}{lll}
e & ::= & be \mid ae \mid le \mid c(e_1, \ldots, e_n) \\
be & ::= & x \mid \texttt{true} \mid \texttt{false} \mid ae\ rop\ ae \mid be\ bop\ be \mid \texttt{not}\ be \mid e_1\texttt{=}e_2 \mid e_1\texttt{<>}e_2 \\
ae & ::= & x \mid n \mid ae\ op\ ae \\
le & ::= & x \mid [e_1, \ldots, e_n] \mid [e_1, \ldots, e_n|le] \mid e\ \texttt{isin}\ le \mid le_1\ lop\ le_2
\end{array}
$$

$$
\begin{array}{lll}
op & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{mod} \\
bop & ::= & \texttt{and} \mid \texttt{or} \\
rop & ::= & \texttt{>} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{=<} \\
lop & ::= & \texttt{++} \mid \texttt{--}
\end{array}
$$

□

A.2 Data Evaluation

The semantics for most of the value expressions is the standard meaning. Informally,

1. the boolean operators `and`, `or`, and `not` take boolean operands and return the usual boolean result,

2. the arithmetic operators `+`, `-`, `*`, and `mod` take integer operands and return the usual integer result,

3. the relational operators `>=`, `=<`, `<`, and `>` are defined for only integer operands, and return the usual boolean comparison result,

4. the (in)equality operator `=(<>)` is defined on all similar operand types, and returns the usual boolean result,

5. `isin` is the typical *membership* function that returns a boolean result, `true` if the first operand is in the list represented by the second operand, `false` otherwise

6. the list operators `++` and `--` are the normal list concatenation and list subtraction, respectively, with the result being a list

The formal evaluation semantics are given by the *evaluation* function, as described in Figure A 1, where $Ap(o, v_1, v_2)$ is the result of applying the binary operator o to the two operands v_1 and v_2 Similarly, $Ap(o, v)$ is the result of applying the unary operator o to the operand v The evaluation function $[\![\ ]\!]$ is strict, in the sense that if any operand is undefined, the whole result is undefined

$$\frac{}{[\![n]\!] = n} \qquad \frac{}{[\![\texttt{false}]\!] = \texttt{false}} \qquad \frac{}{[\![\texttt{true}]\!] = \texttt{true}}$$

$$\frac{[\![e_1]\!] = v_1, \ldots, [\![e_n]\!] = v_n}{[\![c(e_1, \ldots, e_n)]\!] = c(v_1, \ldots, v_n)} \qquad \frac{[\![e_1]\!] = v_1, \ldots, [\![e_n]\!] = v_n}{[\![[e_1, \ldots, e_n]]\!] = [v_1, \ldots, v_n]}$$

$$\frac{[\![e_1]\!] = v_1, \ldots, [\![e_{n-1}]\!] = v_{n-1}, [\![le]\!] = [v_n, \ldots, v_m]}{[\![[e_1, \ldots, e_{n-1}|le]]\!] = [v_1, \ldots, v_{n-1}, v_n, \ldots, v_m]} \quad n \geq 2,\ m \geq n$$

$$\frac{[\![ae_1]\!] = n_1,\ [\![ae_2]\!] = n_2}{[\![ae_1\ op\ ae_2]\!] = Ap(op, n_1, n_2)} \qquad \frac{[\![ae_1]\!] = n_1,\ [\![ae_2]\!] = n_2}{[\![ae_1\ rop\ ae_2]\!] = Ap(rop, n_1, n_2)}$$

$$\frac{[\![be_1]\!] = b_1,\ [\![be_2]\!] = b_2}{[\![ae_1\ bop\ ae_2]\!] = Ap(bop, n_1, n_2)} \qquad \frac{[\![be]\!] = b}{[\![\texttt{not}\ be]\!] = Ap(\texttt{not}, b)}$$

$$\frac{[\![e_1]\!] = v_1,\ [\![e_2]\!] = v_2}{[\![e_1\ o\ e_2]\!] = Ap(o, v_1, v_2)} \quad o \in \{\texttt{<>}, \texttt{=}\}$$

$$\frac{[\![le_1]\!] = [v_1, \ldots, v_{n-1}],\ [\![le_2]\!] = [v_n, \ldots, v_m]}{[\![le_1\texttt{++}le_2]\!] = [v_1, \ldots, v_{n-1}, v_n, \ldots, v_m]} \quad n > 1,\ m \geq n$$

$$\frac{[\![le_1]\!] = [v_1, \ldots, v_{n-1}],\ [\![le_2]\!] = [v_n, \ldots, v_m]}{[\![le_1\texttt{--}le_2]\!] = [v_1, \ldots, v_{n-1}] \setminus [v_n, \ldots, v_m]} \quad n > 1,\ m \geq n$$

$$\frac{[\![e]\!] = v,\ [\![le]\!] = [v_1, \ldots, v_n]}{[\![e\ \texttt{isin}\ le]\!] = Ap(\texttt{isin}, v, [v_1, \ldots, v_n])}$$

Figure A.1: Evaluation Semantics

Appendix B

Bisimulation Equivalence

Although not directly related to this thesis, a (concrete) bisimulation equivalence is provided. It has the disadvantage that for value-passing process algebras where values may have infinite domains, the equivalence checking involves the comparisons of infinite transitions.

Definition 17 A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a *(strong) bisimulation* if $(P, Q) \in S$ then

1. $P\downarrow$ if and only if $Q\downarrow$.
2. $\forall \alpha \in Act$
   - if $P \xrightarrow{\alpha} P', \theta_1$ then $\exists\, Q',\ \theta_2,\ Q \xrightarrow{\alpha} Q', \theta_2$, and $(P'\theta_1, Q'\theta_2) \in S$, and
   - if $Q \xrightarrow{\alpha} Q', \theta_2$ then $\exists\, P',\ \theta_1,\ P \xrightarrow{\alpha} P'$, and $(P\theta_1', Q'\theta_2) \in S$.

For $(P, Q) \in S$, we write $P \sim Q$. □

An alternative, weaker equivalence is to determine if two processes are equivalent only up to a maximum of n steps. This is referred to as *n-bisimulation equivalence*.

Definition 18 A set of binary relations $S^n = \{S_i \mid S_i \subseteq \mathcal{P} \times \mathcal{P}, 0 \leq i \leq n\}$ is a (*strong*) *n-bisimulation* if

1. $(P, Q) \in S_0$ for all $P, Q \in \mathcal{P}$;
2. if $(P, Q) \in S_n$ implies
   - $P\downarrow$ if and only if $Q\downarrow$,
   - for all $\alpha \in Act$
     - if $P \xrightarrow{\alpha} P', \theta_1$ then $\exists\ Q',\ \theta_1,\ Q \xrightarrow{\alpha} Q', \theta_2$, and $(P'\theta_1, Q'\theta_2) \in S_{n-1}$; and
     - if $Q \xrightarrow{\alpha} Q', \theta_2$ then $\exists\ P',\ \theta_1,\ P \xrightarrow{\alpha} P', \theta_1$, and $(P'\theta_1, Q'\theta_2) \in S_{n-1}$

For $(P, Q) \in S^n$, we write $P \sim_n Q$ □

Now bisimulation equivalence $\sim$ can be defined as $\bigcap_{n \in N} \sim_n$ It is easy to see that $\sim_0 \supseteq \sim_1 \supseteq$ and if $\sim_n = \sim_{n+1}$ then $\sim_n = \sim$ There will always be such an n in the finite state case since there are only finitely many equivalence relations

Appendix C

Source Code

```
%
% Module: lts
%
% Written by: Philip J. Wiebe   (February 1993)
% Modified by: Philip J. Wiebe  (August 1993)
% Modified by: Philip J. Wiebe  (October 1993)
%
% Description:
%    This module is an implemenation of the Labelled Transition
%    System for the process alebra language, Algebra of Concurrent
%    Systems (ACS).
%
%

trans( P, A, P1 ) :-
        trans(P, [], [], A, P1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          New and improved labelled transition system               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
% Written by: Philip J. Wiebe   (November 1993)
```

```
%

trans( op( if, P, Preds), Res, Ren, A, P1) :-
        !,
        eval(Preds, bool(true) ),
        trans( P, Res, Ren, A, P1).

trans( op(',',P,Q), Res, Ren, A, V ) :-
        trans(P, Res, Ren, A, P1),
        combine(op(',',P1,Q),V).
trans( op(',',P,Q), Res, Ren, A, Q1 ) :-
        !,
        terminates(P),
        trans(Q, Res, Ren, A, Q1).

trans( op('+',P,_), Res, Ren, A, P1) :-
        trans(P, Res, Ren, A, P1).
trans( op('+',_,Q), Res, Ren, A, Q1) :-
        !,
        trans(Q, Res, Ren, A, Q1).

trans( op('|',P,Q), Res, Ren, A, V ) :-
        trans(P, Res, Ren, A, P1),
        combine(op('|',P1,Q),V).
trans( op('|',P,Q), Res, Ren, A, V ) :-
        trans(Q, Res, Ren, A, Q1),
        combine(op('|',P,Q1),V).

trans( op('|',P,Q), _, _, tau(A), V ) :-
        trans(P, [], [], op('~',A), P1),
        trans(Q, [], [], A, Q1),
        combine(op('|',P1,Q1),V).
trans( op('|',P,Q), _, _, tau(A), V ) :-
        trans( Q, [], [], op('~',A), Q1),
        trans( P, [], [], A, P1),
        combine(op('|',P1,Q1),V).

trans( op('|',P,Q), Res, Ren, A, Q1 ) :-
        terminates(P),
        trans(Q, Res, Ren, A, Q1).
trans( op('|',P,Q), Res, Ren, A, P1 ) :-
        !,
```

```
        terminates(Q),
        trans(P, Res, Ren, A, P1).

trans( op('#',P,F), Res, Ren, A, V ) :-
        !,
        f_inverse(Res, F, NewRes),
        fcn_compose(F,Ren,NewRen),
        !,
        trans( P, NewRes, NewRen, A, P1),
        combine(op('#',P1,F),V).

trans( op('\',P,L), Res, Ren, A, V ) :-
        !,
        union(Res,L,NewRes),
        !,
        trans(P, NewRes, Ren, A, P1),
        combine(op('\',P1,L),V).

trans( op('`',A), Res, Ren, op('`',B), 'SKIP') :-
        !,
        input_action(A),
        is_not_in(A, Res),
        eval(A,A1),
        renaming(A1, Ren, B).
trans( A, Res, Ren, B, 'SKIP') :-
        input_action(A),
        !,
        is_not_in(A, Res),
        eval(A,A1),
        renaming(A1, Ren, B).
trans( tau, _, _, tau, 'SKIP') :- !.

trans(P, Res, Ren, A, P1) :-
        eval( P, P2),
        defn( P2, R, _),
        trans( R, Res, Ren, A, P1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           Utilities for labelled transition system                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%
```

```
%% terminates(+Agent)
%  succeeds if Agent can terminate without performing an action.
% i.e. may become SKIP without performing an action.
terminates(A) :- terminates(A,[]).

terminates('SKIP',_) :-!.
terminates( op('+',X,_), L) :- terminates(X, L).
terminates( op('+',_,X), L) :- !, terminates(X, L).
terminates( op('|',X,Y), L) :- !, terminates(X, L), terminates(Y,L).
terminates( op(',',X,Y), L) :- !, terminates(X, L), terminates(Y,L).
terminates( op('#',X,_), L) :- !, terminates(X, L).
terminates( op('\',X,_), L) :- !, terminates(X, L).
terminates( op(if,X,C), L)  :- !, eval(C,bool(true)), terminates(X, L).
terminates( A, L ) :-
        eval(A,B),
        \+ member(B,L),
        !,
        defn(B,P,_),
        terminates(P, [B|L]).


%% initialActions(+P,-ActionList)
%       returns the list of immediate successors of the process P.
%
initialActions(P,Acts) :-
        setof(A, X^( trans(P,X,_), most_general_action(X,A) ), Acts),
        !.
initialActions(_,[]).

%% most_general_action(+Act,-Act')
%       returns the most general action of a given action.  That is,
%       all parameters have strictly variables instead of any value
%       expressions.
%
most_general_action(lid(A), lid(A)).
most_general_action(func(A,N,_), func(A,N,P) ) :- length(P,N).
most_general_action(op('`',A), op('`',MA) ) :- most_general_action(A, MA).
most_general_action(Tau,tau(_)) :- tau_action(Tau), !.

%% successors(+Agent, -ListOfTransitions)
%
successors(N, L) :-
```

```
        setof( pr(X,Y), trans(N, X, Y), L), !.
successors(_, []).

%% descendant(+Agent,-Action,-Agent',-Path)
%     returns an immediate descendant of Agent, including the path used
%     to reach the given expression.
%
descendant(P,A,Q,Path) :-
        descendant(P,A,Q,Path,[]).

descendant(P,A,Q,[A],_) :-
        trans(P,A,Q).
descendant(P,A,Q,[Tau|Path],ProcessedAgents) :-
        non_member(P, ProcessedAgents),
        tau_action(Tau),
        trans(P,Tau,P1),
        descendant(P1,A,Q,Path,[P|ProcessedAgents]).

%% descendants(+P,-List)
%   returns a list of immediate descendants of the expression P.
descendants(P, L) :-
        findall( pr(A,Q,Path), descendant(P,A,Q,Path), L).


%% renaming(+Action, +Function, -Action')
%
%       Renames Action to Action' by the function defined by Function.
%       Assumes Action is in canonical form, Action' is also in
%       canonical form.
%
% This is deterministic.
%
renaming(tau, _, tau)       :- !.
renaming(tau(A),_,tau(A))   :- !.
renaming(X, [], X)          :- !.
renaming( func(X,N,P), Ps, func(Y,N,P) ) :-
        !,
        renaming(X, Ps, Y).
renaming(op('`',X), Ps, Y) :- !,
        renaming(X,Ps,Y1),
        canonical_action(op('`',Y1), Y).
renaming(X, [P|_], Y)       :- % X =/= op('`',R)
```

```
        rename(X, P, Y1),
        !,
        canonical_action(Y1, Y), !.
renaming(X, [_|Ps], Y) :-
        renaming(X, Ps, Y).

%% rename(+Action, +ForPair, -Action')
%       Assumes that Action is ALWAYS a positive action
%       IE that it will never be an output action
rename(X, op('/',Y,X1), Y ) :-
        same_action_label(X,X1),
        !.
rename(X, op('/',Y,op('`',X1)), op('`',Y) ) :-
        same_action_label(X,X1).

%% is_not_in(+X, +L)
%       succeeds if X or op('`',X) is a member of L
%       (N.B., it is deterministic.
%
is_not_in(_,[])     :- !.
is_not_in(tau,_ )   :- !.
is_not_in(A,[B|X]) :-
        \+ same_action_label(A,B),
        !,
        is_not_in(A,X).

%% label(+Act,?Label)
%
label(func(A,_,_), A) :- !.
label(op('`',A), B)   :- !, label(A,B).
label(A, A).

%% same_action_label(?Act,?Act)
%       Normal usage is when both parameters are input, although
%       it can be used to generate all actions that are considered
%       to be the same. Actions are the same if, after being
%       stripped of all bars, and parameters they are unifyable.
%
same_action_label( A, A ).
same_action_label( A, func(A,_,_) ).
same_action_label( func(A,_,_), A ).
same_action_label( A, op('`',B) ) :- same_action_label(A,B).
```

```
same_action_label( op('`',A), B ) :- same_action_label(A,B).

%% atomic_action(+Action)
%       holds if Action has no parameters
%
atomic_action( lid(_) ).
atomic_action( op('`',lid(_)) ).

%% input_action(+Action)
%       True if parameter is an input action.
%
input_action( lid(_)          ) :- !.
input_action( func(lid(_),_,_) ) :- !.

output_action( op('`',Act) ) :- input_action(Act).

tau_action( tau ).
tau_action( tau(_) ).

%% action(+Action)
%       True if parameter is an action.
%
action( Act ) :- tau_action(Act), !.
action( Act ) :- input_action(Act), !.
action( Act ) :- output_action(Act), !.

%% canonical_action( +Action, -Action' ).
%       returns the canonical form of Action.
%
canonical_action(op('`', op('`',A)), A1) :-
        !,
        canonical_action(A,A1).
canonical_action(A, A).

%% complementary(+Action,+Action')
%       true if actions are complementary.
complementary(op('`',A),B) :- \+ \+ A = B.
complementary(B,op('`',A)) :- \+ \+ A = B.


%% combine( +ACSTerm, -ACSTerm')
%
```

```
%   Removes unnecessary 'STOP' and 'SKIP' from an ACS expression.
%

% rules for 'STOP'
combine( op(',','STOP',_)  , 'STOP' ) :- !.
combine( op('#','STOP',_)  , 'STOP' ) :- !.
combine( op('\','STOP',_)  , 'STOP' ) :- !.
combine( op(if,'STOP',_)   , 'STOP' ) :- !.
combine( op('+','STOP',R)  , R )      :- !.
combine( op('+',L,'STOP')  , L )      :- !.
% combine( op('|','STOP',R) , R )     :- !.
% combine( op('|',L,'STOP') , L )     :- !.

% rules for 'SKIP'
combine( op(',','SKIP',R)  , R )      :- !.
combine( op(',',L,'SKIP')  , L )      :- !.
combine( op('|','SKIP',R) , R )       :- !.
combine( op('|',L,'SKIP') , L )       :- !.
combine( op('#','SKIP',_)  , 'SKIP' ) :- !.
combine( op('\','SKIP',_)  , 'SKIP' ) :- !.
combine( op(if,'SKIP',_)   , 'SKIP' ) :- !.

% rules for restriction and renaming
combine( op('\',op('\',P,L),K), op('\',P,LK) )     :-
        union(L,K,LK), !.
combine( op('#',op('#',P,F1),F2), op('#',P,F12) ) :-
        fcn_compose(F1,F2,F12).

% default to original
combine( X, X ).


%% f_inverse(+Set, +F, -Set' )
%
% f_inverse(L) = { a in L | f(a) = a} union
%                { b in dom(f) | a in L, f(b) = a}.
%
f_inverse(L,F,NewL) :-
        f1_inv(L,F,L1),
        f2_inv(L,F,L2),
        union(L1,L2,NewL), !.
```

```
f1_inv([],_,[]).
f1_inv([A|As],F,Bs) :-
          % either input or output label may be in F
        ( member(op('/',_,A),F); member(op('/',_,op('`',A)), F) ),
        !,
        f1_inv(As,F,Bs).
f1_inv([A|As],F,[A|Bs]) :-
        % \+ ( member(op('/',_,A),F), member(op('/',_,op('`',A)), F) ),
        f1_inv(As,F,Bs).

f2_inv(_,[],[]).
f2_inv(L,[op('/',A,_)|F],L1) :-
        is_not_in(A,L),
        !,
        f2_inv(L,F,L1).
f2_inv(L,[op('/',A,B)|F],[C|L1]) :-
        % want the input action label
        (output_action(B) -> complementary(B,C);B=C),
        !,
        f2_inv(L,F,L1).


%% fcn_compose(F1, F2, F21)
%       computes (f2 o f1)x = f2(f1(a))
%
fcn_compose(F1,F2,F21) :- fcn_compose(F1,F2,[],F21).

fcn_compose([],[],F21,F21) :- !.
%
% f1: a-->b, f2: b-->c then f2 o f1: a-->c
%
fcn_compose( [op('/',B,A)|F1], F2, InF21, OutF21 ) :-
        member( op('/',C,B), F2),
        !,
        delete_one(F2, op('/',C,B), NF2),
        fcn_compose(F1, NF2, [op('/',C,A)|InF21], OutF21).
%
% f1: a --> b,  f2: b -/-> _)  then f2 o f1: a --> b.
%
fcn_compose( [op('/',B,A)|F1], F2, InF21, OutF21 ) :-
        !,
        fcn_compose(F1, F2, [op('/',B,A)|InF21], OutF21).
```

```
%
% f2: a-->b, but f2 o f1: a-->c already, so drop a-->b
%
fcn_compose( [], [op('/',_,A)|F2], InF21, OutF21) :-
        member(op('/',_,A), InF21),
        !,
        fcn_compose([], F2, InF21, OutF21)
fcn_compose( [], [op('/',B,A)|F2], InF21, OutF21) :-
        fcn_compose([], F2, [op('/',B,A)|InF21], OutF21).

%% instantiate(?Action)
%       instantiates all variables in Action, prompting the user
%       for input, if required.
%
instantiate( op('`',Act) ) :- ground_term( Act ), !.
instantiate( Act ) :- ground_term( Act ), !.
instantiate( Term ) :-
        repeat,
          echo_expression( Term ),
          ask_input( ' = ', L ),
          get_next_expression( L, Input ),
          Term = Input, !.


%% ground_term(+Term)
%     holds if Term does not contain variables.
%
ground_term( X          ) :- var(X), !, fail.
ground_term( uid(_)     ) :- !.
ground_term( lid(_)     ) :- !.
ground_term( const(_) ) :- !.
ground_term( num(_)     ) :- !.
ground_term( bool(_)    ) :- !.
ground_term( tau        ) :- !.
ground_term( tau(_)     ) :- !.
ground_term( [A|As]     ) :- !, ground_term( A ), ground_term( As ).
ground_term( []         ) :- !.
ground_term( func(_,_,Args) ) :- ground_term( Args ).


%
% Module: paths
```

```
%
% Written by Philip J Wiebe  (October 1993)
%
% Description:
%    This module is an implemenation of the Labelled Transition
%    System for the path expression language, Concurrent Path
%    Expressions (CPE), and the simulation code.
%
%

%% strong_simulate(+Agent, +Path, -Trace, -Agent').
%
strong_simulate(Ag, 'SKIP', [], Ag) :- !.
strong_simulate(Ag, P, [], Ag) :-
        no_path_step( P ),
        \+ trans(Ag,_,_).
strong_simulate(Ag, P, [Act|Ts], RAg) :-
        path_step(P,Act,P1),
        trans(Ag,Act,Ag1),
        instantiate(Act),
        strong_simulate(Ag1, P1, Ts, RAg).

%% weak_simulate(+Agent, +Path, -Trace, -Agent').
%
weak_simulate(Ag, 'SKIP', [], Ag) :- !.
weak_simulate(Ag, P, [], Ag) :-
        no_path_step( P ),
        \+ descandant(Ag,_,_).
weak_simulate(Ag, P, NewT, RAg) :-
        path_step(P,Act,P1),
        descendant(Ag,Act,Ag1,APath),
        instantiate(Act),
        weak_simulate(Ag1, P1, Ts, RAg),
        append(APath,Ts,NewT).

%% no_path_step(+PathExp) holds if PathExp cannot perform any action.
%  This is an implementation of 'path_dead'.
%
no_path_step( 'STOP' ) :- !.
no_path_step(op(';',P1,_))  :- no_path_step(P1), !.
no_path_step(op(';',P1,P2)) :- !, path_term(P1), no_path_step(P2).
no_path_step(op('+',P1,_))  :- no_path_step(P1), !.
```

```
no_path_step(op('+',_,P2)) :- !, no_path_step(P2).
no_path_step(op('|',P1,_)) :- no_path_step(P1), !.
no_path_step(op('|',_,P2)) :- !, no_path_step(P2).

%% path_step(+Path,?Action,?Path')
% holds if Path can perform Action and then become Path'.
%
path_step(X, X, 'SKIP') :- var(X), !.
path_step(anyvar,_,'SKIP') :- !.    % any action
path_step(op(',',P1,P2),A,PP2) :-
        path_term(P1),
        path_step(P2,A,PP2).
path_step(op(',',P1,P2),A,V) :-
        path_step(P1,A,PP1),
        combine(op(',',PP1,P2),V).

path_step(op('+',P1,_),A,PP1) :- path_step(P1,A,PP1).
path_step(op('+',_,P2),A,PP2) :- path_step(P2,A,PP2).

path_step(op('|',P1,P2),A,PP2) :- path_term(P1), path_step(P2,A,PP2).
path_step(op('|',P1,P2),A,PP1) :- path_term(P2), path_step(P1,A,PP1).
path_step(op('|',P1,P2),A,V)   :- path_step(P1,A,PP1),
                                   combine(op('|',PP1,P2),V).
path_step(op('|',P2,P1),A,V)   :- path_step(P1,A,PP1),
                                   combine(op('|',P2,PP1),V).

path_step(op('^',P,num(1)),A,P1) :- path_step(P,A,P1).
path_step(op('^',P,num(N)),A,P1) :-
        N > 1, N1 is N - 1,
        path_step(op(',',P,op('^',P,num(N1))), A, P1).

path_step(op('*',P,num(1)),A,P1) :- path_step(P,A,P1).
path_step(op('*',P,num(N)),A,P1) :-
        N > 1, N1 is N - 1,
        path_step(op(',',P,op('*',P,num(N1))), A, P1).

path_step(op('!',ASet),B,'SKIP') :-
        freeze(B, not_in_action_set(B,ASet) ).

path_step(A,A,'SKIP') :- action(A), !.

%% not_in_action_set(+Action,+ActionSet) holds if Action is not in
```

```
%       ActionSet
%
not_in_action_set(_,[]) :- !.
not_in_action_set(Act, [A|As]) :-
        \+ Act = A,
        not_in_action_set(Act,As).

%% path_term(+PathExp) holds if PathExp terminates.
%
path_term( op('*',_,_)      ) :- !.
path_tern( 'SKIP'           ) :- !.
path_term( op('+',A,_)      ) :- path_term(A), !.
path_term( op('+',_,B)      ) :- !, path_term(B).
path_term( op('|',A,B )     ) :- !, path_term(A), path_term(B).
path_term( op(';',A,B)      ) :- !, path_term(A), path_term(B).
path_term( op('^',_,num(0)) ) :- !.
path_term( op('^',P,_)      ) :- !, path_term(P).

%% semantic_check_path(+PathExp,-PathExp',+VarList,-VarList')
% Checks the semantics of a path expression.
%
% P ::= Var | act | P+P | P|P | P^N | P*N | !{a1,...,an} | STOP | SKIP
%
semantic_check_path(op(';',P1,P2), op(';',V1,V2), InVars, OutVars) :-
        !,
        semantic_check_path(P1,V1,InVars,OutV1),
        semantic_check_path(P2,V2,OutV1,OutVars).
semantic_check_path(op('|',P1,P2), op('|',V1,V2), InVars, OutVars) :-
        !,
        semantic_check_path(P1,V1,InVars,OutV1),
        semantic_check_path(P2,V2,OutV1,OutVars).
semantic_check_path(op('^',P,num(N)), op('^',V,num(N)), InVars, OutVars) :-
        !,
        N > 0,
        semantic_check_path(P,V,InVars,OutVars).
semantic_check_path(op('*',P,num(N)), op('*',V,num(N)), InVars, OutVars) :-
        !,
        N > 0,
        semantic_check_path(P,V,InVars,OutVars).
% cannot introduce new variables
semantic_check_path(op('!',ASet), op('!',VASet), InVars, InVars) :-
        !,
```

```
        semantic_check_path_action_set(ASet, VASet, InVars, InVars).
semantic_check_path('SKIP','SKIP', E, E) :- !.
semantic_check_path('STOP','STOP', E, E) :- !.
semantic_check_path(A,A1,InVars,OutVars) :-
        semantic_check_path_action(A,A1,InVars,OutVars), !.

semantic_check_path_action_set([], [], E, E) :- !.
semantic_check_path_action_set([A|As], [V|Vs], E1, E3) :-
        semantic_check_path_action(A,V,E1,E2),
        semantic_check_path_action_set(As,Vs,E2,E3).

semantic_check_path_action(tau   , tau   , E, E) :- !.
semantic_check_path_action(tau(A), tau(A), E, E) :- !.
semantic_check_path_action(lid(A), lid(A), E, E) :- !.
semantic_check_path_action(var('_'),anyvar, E, E) :- !. % 'any' variable
semantic_check_path_action(var(X), V, E1, E2) :- % higher-order action
        !,
        new_var(E1,E2,X,V).
semantic_check_path_action(op('`',A), op('`',T), E1, E2) :- !,
        !,
        semantic_check_path_action(A, T, E1, E2).
semantic_check_path_action(func(lid(A),N,Ts), func(lid(A),N,Ts1), E1, E2) :-
        semantic_check_term(Ts, Ts1, E1, E2).



%
% Module: eval
%
% Written by:  Philip J. Wiebe   (February, 1993)
% Modified by: Mantis H.M. Cheng (August,   1993)
% Modified by: Philip J. Wiebe   (October,  1993)
%
% Description:
%  This module implements the value and process evaluation for ACS type
%  expressions. Implementation of Appendix A of Philip J. Wiebe's Thesis.
%

%% eval( +Term, -Value )
%       Value may be 'error' if Term cannot be evaluated.
%       Value is either 'bool(true)' or 'bool(false)' for boolean Term.
%
```

```
% Notes: It NEVER fails!
%        All evaluable and constructor functions are STRICT!
%        (i.e., f(error) = error -- It preserves error.)
%
eval( X,       Y      ) :- var(X), !, Y = error.
eval( error, error ) :- !.
eval( [],      []     ) :- !.
eval( [T|Ts],     [V|Vs] ) :-
        eval( T, V ),
        V \== error,
        eval( Ts, Vs ),
        Vs \== error,
        !.
eval( [_|_],     error )  :- !.
eval( bool(T),    V  )    :- !, V = bool(T).
eval( true,       V  )    :- !, V = bool(true).
eval( false,      V  )    :- !, V = bool(false).
eval( num(T),     V  )    :- !, V = num(T).
eval( lid(T),     V  )    :- !, V = lid(T).
eval( uid(T),     V  )    :- !, V = uid(T).
eval( 'STOP',     V  )    :- !, V = 'STOP'.
eval( 'SKIP',     V  )    :- !, V = 'SKIP'.
eval( const(T),   V  )    :- !, V = const(T).
eval( X,          V  )    :- atomic(X), !, X = V.

eval( func(Id,N,Ts), V ) :-
        eval( Ts, Vs ),
        Vs \== error,
        !,
        V = func(Id,N,Vs).
eval( op('+',T1,T2), num(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        V is V1 + V2.
% '+' is also used for agents.
eval( op('+',T1,T2), op('+',V1,V2) ) :-
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
        V2 \== error,
        !.
```

```
eval( op('-',T), num(V) ) :-
        eval( T, num(V1) ),
        !,
        V is -V1.
eval( op('-',T1,T2), num(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        V is V1 - V2.
eval( op('*',T1,T2), num(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        V is V1 * V2.
eval( op(mod,T1,T2), num(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        V is V1 mod V2.
eval( op('<',T1,T2), bool(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        apply( '<', V1, V2, V ).
eval( op('>',T1,T2), V ) :- !,
        eval( op('<',T2,T1), V ).
eval( op('=<',T1,T2), bool(V) ) :-
        eval( T1, num(V1) ),
        eval( T2, num(V2) ),
        !,
        apply( '=<', V1, V2, V ).
eval( op('>=',T1,T2), V ) :- !,
        eval( op('=<',T2,T1), V ).
eval( op('<>',T1,T2), bool(V) ) :-
        eval( T1, V1 ),
        eval( T2, V2 ),
        !,
        apply( '<>', V1, V2, V ).
eval( op(not,T), bool(V) ) :-
        eval( T, bool(V1) ),
        !,
        apply( not, V1, V ).
```

```
eval( op(and,T1,T2), bool(V) ) :-
        eval( T1, bool(V1) ),
        eval( T2, bool(V2) ),
        !,
        apply( and, V1, V2, V ).
eval( op(or,T1,T2), bool(V) ) :-
        eval( T1, bool(V1) ),
        eval( T2, bool(V2) ),
        !,
        apply( or, V1, V2, V ).
eval( op('=',T1,T2), error ) :-
        var(T1),
        var(T2), !.
eval( op('=',T1,T2), bool(true) ) :-
        var(T1),
        nonvar(T2),
        eval( T2, V2 ),
        V2 \== error,
        !,
        T1 = V2.
eval( op('=',T1,T2), bool(true) ) :-
        var(T2),
        nonvar(T1),
        eval( T1, V1 ),
        V1 \== error,
        !,
        T2 = V1.
eval( op('=',T1,T2), bool(V) ) :-
        nonvar(T2),
        nonvar(T1),
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
        V2 \== error,
        !,
        apply( '=', V1, V2, V ).
eval( op(isin,T1,T2), bool(V) ) :-
        nonvar(T2),
        nonvar(T1),
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
```

```
        V2 \== error,
        !,
        apply( isin, V1, V2, V ).
eval( op('++',T1,T2), V ) :-
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
        V2 \== error,
        !,
        append( V1, V2, V ).
eval( op('--',T1,T2), V ) :-
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
        V2 \== error,
        !,
        list_diff( V1, V2, V ).
eval( op(O,T1,T2), op(O,V1,V2) ) :-  % not builtin binary op
        eval( T1, V1 ),
        V1 \== error,
        eval( T2, V2 ),
        V2 \== error, !.
eval( op(O,T), op(O,V) ) :-          % not builtin unary op
        eval( T, V ),
        V \== error, !.
eval( _, error ).                    % everything else is an error

%
% evaluable( +Op, +Arity) holds if Op/Arity is a valid operator of ACS.
%
evaluable( '--', 2 ).
evaluable( '++', 2 ).
evaluable( '=', 2 ).
evaluable( '=<', 2 ).
evaluable( '>=', 2 ).
evaluable( '<>', 2 ).
evaluable( '<', 2 ).
evaluable( '>', 2 ).
evaluable( '*', 2 ).
evaluable( '+', 2 ).
evaluable( '-', 2 ).
evaluable( '-', 1 ).
```

```
evaluable( mod, 2 ).
evaluable( isin, 2 ).
evaluable( not, 1 ).
evaluable( and, 2 ).
evaluable( or, 2 ).

%
% For booleans only
%
apply( not, true,  false ) :- !.
apply( not, false, true )  :- !.

apply( and, true,  true,  true )  :- !.
apply( and, _,     _,     false ) :- !.
apply( or,  false, false, false ) :- !.
apply( or,  _,     _,     true )  :- !.
%
% For numbers only
%
apply( '<', V1, V2, true )   :- V1 < V2, !.
apply( '<', V1, V2, false )  :- V1 >= V2, !.
apply( '=<',V1, V2, true )   :- V1 =< V2, !.
apply( '=<',V1, V2, false )  :- V1 > V2, !.
%
% For numbers and symbols
%
apply( '=', V1, V2, B )  :-  !,
        valueof(V1,S1),
        valueof(V2,S2),
        (S1 = S2 -> B = true, B = false).
apply( '<>', V1, V2, B )  :-  !,
        valueof(V1,S1),
        valueof(V2,S2),
        (S1 \== S2 -> B = true, B = false).
%
% For U X lists only
%
apply( isin , V1, V2, V )  :- member_con(V1,V2,V), !.

%
% valueof( +Term1, -Term2) holds if Term1 is a ACS data type.  Then
% Term2 is the actual data value.
```

```
%
valueof( num(T),   T) :- !.
valueof( const(T), T) :- !.
valueof( lid(T),   T) :- !.
valueof( bool(T),  T) :- !.
valueof( uid(T),   T).

%
% member_con(+Element, +List, -Bool) is the member function for
%    Arbutus style lists.
%
member_con( _, [],              false )  :- !.
member_con( X, op(':',X,_),     true )   :- !.
member_con( X, op(':',_,L),     V ) :-
        member_con( X, L, V ).



%
% Module:        semantic
%
% Written by:    Mantis H.M. Cheng (August, 1993)
% Modified by:   Philip J. Wiebe   (October, 1993)
%
%
% Description:
%   This modules checks the semantics of an ACS expression.
%

%% semantic_check(+Expression, -Expression', ?Environment)
%


%
% Definitions must be of the form:
%
%  <Defn>  ::= <Con> ':=' <Agent>
%
% where free variables in <Agent> must be defined in <Con>
%
semantic_check(op(':=',Con, Agent),op(':=',Con1,Agent1),Env2) :-
        semantic_check_con(Con, Con1, [], Env1),
        semantic_check_agent(Agent, Agent1, [], Env1, Env2 ), !.
```

```
%
% <Con>   ::= <Uid>                          -- agent constant
%           | <Uid> '(' <Term> ... <Term> ')'
%           | <Term> <Op> <Term>
%           | <Op> <Term>
%           | <Term> <Op>
%
% N.B. Every <Con> is also a <Term>, but not vice-versa.
%
semantic_check_con( uid(T), uid(T), Env, Env ) :- !.
semantic_check_con( func(uid(F),N,Ts), func(uid(F),N,Ts1), E1, E2 ) :-
        !,
        semantic_check_term( Ts, Ts1, E1, E2 ).
semantic_check_con( op(O,E), op(O,T), Env1, Env2 ) :-
        !,
        semantic_check_term( E, T, Env1, Env2 ).
semantic_check_con( op(O,E1,E2), op(O,T1,T2), Env1, Env3 ) :-
        semantic_check_term( E1, T1, Env1, Env2 ),
        semantic_check_term( E2, T2, Env2, Env3 ).

%
% <Term>  ::= <Constant>
%           | <Boolean>
%           | <Number>
%           | <Variable>
%           | 'STOP'
%           | 'SKIP'
%           | <lid>
%           | <lid> '(' <Term> ... <Term> ')'
%           | '[' <Term> ... <Term> ']'
%           | <Con>
%
% N.B. A <Term> is a more general <Agent>
%
semantic_check_term( bool(T)  , bool(T)  , E, E ) :- !.
semantic_check_term( const(T), const(T), E, E ) :- !.
semantic_check_term( num(T)   , num(T)   , E, E ) :- !.
semantic_check_term( lid(X)   , lid(X)   , E, E ) :- !.
semantic_check_term( uid(X)   , uid(X)   , E, E ) :- !.
semantic_check_term( 'STOP'   , 'STOP'   , E, E ) :- !.
```

```
semantic_check_term( 'SKIP'  , 'SKIP'   , E, E ) :- !.
semantic_check_term( var('_'), _        , E, E ) :- !.
semantic_check_term( var(X)  , V        , E1, E2 ) :-       % variable
        !,
        new_var( E1, E2, X, V ).
semantic_check_term( func(lid(X),N,Ts), func(lid(X),N,Ts1), E1, E2) :-
        !,
        semantic_check_term(Ts, Ts1, E1, E2 ).
semantic_check_term( op('|',T,Ts), [T1|Ts1], E1, E3) :-
        !,
        semantic_check_term( T, T1, E1, E2 ),
        semantic_check_term( Ts, Ts1, E2, E3 ).
semantic_check_term( [], [], E, E ) :- !.
semantic_check_term( [T|Ts], [T1|Ts1], E1, E3 ) :-
        !,
        semantic_check_term( T, T1, E1, E2 ),
        semantic_check_term( Ts, Ts1, E2, E3 ).
semantic_check_term( E, T, E1, E2 ) :-
        semantic_check_con( E, T, E1, E2 ).

%
% <Agent> ::= <Variable>
%           | 'STOP'
%           | 'SKIP'
%           | <Agent>    'where' <Eqns>
%           | <Agent>    '#'     <Identifier>
%           | <Agent>    '\'     <Identifier>
%           | <Agent>    'if'    <Cond>
%           | <Agent>    '+'     <Agent>
%           | <Agent>    '|'     <Agent>
%           | <Agent>    ','     <Agent>
%           | <Variable> ':'     <Agent>    % <Variable> MUST be a new var.
%           | <Action>
%           | <Con1>
%
% Every <Identifier> must be defined in the <Eqns>.
% <Cond> may introduce new <Variable> before its use.
% Only input <Action> is allowed to introduce new <Variable>.
%
semantic_check_agent( 'STOP', 'STOP', _, E, E ) :- !.
semantic_check_agent( 'SKIP', 'SKIP', _, E, E ) :- !.
semantic_check_agent( var(X), V     , _, E, E ) :-
```

```
        % variable must be defined
        member( X=V, E ), !.
semantic_check_agent( var(X), _       , _, E, E ) :-  % variable undefined
        !,
        errormsg('Undefined agent variable',X),
        fail.
semantic_check_agent( op(where,L,Eqns), L1, EqnEnv1, E1, E2 ) :-
        !,
        semantic_check_eqns( Eqns, [], EqnEnv ),
        append( EqnEnv, EqnEnv1, EqnEnv2 ),
        semantic_check_agent( L, L1, EqnEnv2, E1, E2 ).
semantic_check_agent( op('#',A,I), op('#',A1,F), Eqn, E1, E2 ) :-
        identifier( I ),
        !,
        member( I=F, Eqn ), !,
        semantic_check_agent( A, A1, Eqn, E1, E2 ).
semantic_check_agent( op('#',A,F), op('#',A1,F), Eqn, E1, E2 ) :-
        !,
        semantic_check_agent( A, A1, Eqn, E1, E2 ).
semantic_check_agent( op('\',A,I), op('\',A1,L), Eqn, E1, E2 ) :-
        identifier( I ),
        !,
        member( I=L, Eqn ), !,
        semantic_check_agent( A, A1, Eqn, E1, E2 ).
semantic_check_agent( op('\',A,L), op('\',A1,L), Eqn, E1, E2 ) :-
        !,
        semantic_check_agent( A, A1, Eqn, E1, E2 ).
semantic_check_agent( op(if,L,R), op(if,L1,R1), Eqn, E1, E3 ) :-
        !,
        semantic_check_predicates(R, R1, E1, E2),
        semantic_check_agent(L, L1, Eqn, E2, E3).
semantic_check_agent( op('+',L,R), op('+',L1,R1), Eqn, E1, E3 ) :-
        !,
        semantic_check_agent(L, L1, Eqn, E1, E2),
        semantic_check_agent(R, R1, Eqn, E2, E3).
semantic_check_agent( op('|',L,R), op('|',L1,R1), Eqn, E1, E4 ) :-
        !,
        semantic_check_agent(L, L1, Eqn, E1, E2),
        semantic_check_agent(R, R1, Eqn, E1, E3),
        % L and R CANNOT have same variables
        ( disjoint_env(E2,E3,E1) ->
                union(E2,E3,E4),
```

```
                errormsg('Shared variables by par: ',op('|',L,R)),fail)
        ).

semantic_check_agent( op(',',L,R), op(',',L1,R1), Eqn, E1, E3 ) :-
        !,
        semantic_check_agent(L, L1, Eqn, E1, E2),
        semantic_check_agent(R, R1, Eqn, E2, E3).
semantic_check_agent( op(':',var(X),P), op(':',V,P1), Eqn, E1, E2) :-
        !,
        (member(X=_, E1) ->
             errormsg('Variable used in recursion already used: ', X) ),
        semantic_check_agent( P, P1, Eqn, [X=V|E1], E2 ).
semantic_check_agent( A, A1, _, E1, E2 ) :-
        semantic_check_action( A, A1, E1, E2 ),
        !.
semantic_check_agent( A, A1, _, E1, E2 ) :-
        semantic_check_con( A, A1, E1, E2 ).  % Should be con1 or have E1=E2
      % But use this to allow


%
% N.B. Same as <Con> except that the operator usages cannot introduce
%        new variables.
%
semantic_check_con1( uid(T), uid(T), Env, Env ) :- !.
semantic_check_con1( func(uid(F),N,Ts), func(uid(F),N,Ts1), E1, E2 ) :-
        !,
        semantic_check_term1( Ts, Ts1, E1, E2 ).
semantic_check_con1( op(O,E), op(O,T), Env1, Env2 ) :-
        !,
        semantic_check_term1( E, T, Env1, Env2 ).
semantic_check_con1( op(O,E1,E2), op(O,T1,T2), Env1, Env3 ) :-
        semantic_check_term1( E1, T1, Env1, Env2 ),
        semantic_check_term1( E2, T2, Env2, Env3 ).

%
% N.B. Same as <Term> with the exception of introducing new variables
%
semantic_check_term1( func(lid(X),N,Ts), func(lid(X),N,Ts), E, E) :- !.
semantic_check_term1( bool(T) , bool(T) , E, E ) :- !.
semantic_check_term1( const(T), const(T), E, E ) :- !.
semantic_check_term1( num(T)  , num(T)  , E, E ) :- !.
```

```
semantic_check_term1( lid(X)  , lid(X)  , E, E ) :- !.
semantic_check_term1( uid(X)  , uid(X)  , E, E ) :- !.
semantic_check_term1( 'STOP'  , 'STOP'  , E, E ) :- !.
semantic_check_term1( 'SKIP'  , 'SKIP'  , E, E ) :- !.
semantic_check_term1( var(X)  , V       , E, E ) :-  % predefined variable
        member( X=V, E),
        !.
semantic_check_term1( var(X)  , _       , E, E ) :-  % undefined variable
        !,
        errormsg('Undefined variable: ',X),
        fail.
semantic_check_term1( [], [], E, E ) :- !.
semantic_check_term1( [T|Ts], [T1|Ts1], E1, E3 ) :-
        !,
        semantic_check_term1( T, T1, E1, E2 ),
        semantic_check_term1( Ts, Ts1, E2, E3 ).
semantic_check_term1( E, T, E1, E2 ) :-
        semantic_check_con1( E, T, E1, E2 ).


%
%
% <Cond>          ::= <Term1> <Connective> ... <Connective> <Term1>
% <Connective>    ::= and | or
%
semantic_check_predicates(op(and,T,Ts), op(and,T1,Ts1), E1, E3) :-
        !,
        semantic_check_term1( T, T1, E1, E2 ),
        semantic_check_predicates( Ts, Ts1, E2, E3 ).
semantic_check_predicates(op(or,T,Ts), op(or,T1,Ts1), E1, E3) :-
        !,
        semantic_check_term1( T, T1, E1, E2 ),
        semantic_check_predicates( Ts, Ts1, E2, E3 ).
semantic_check_predicates(R, R1, E1, E2) :-
        semantic_check_term1( R, R1, E1, E2 ).


%
% <Action>  ::= 'tau'
%             | <Variable>
%             | <lid>
%             | '`' <Action>
%             | <Func> '(' <Term> ... <Term> ')'
```

```
%
% Note: Output <Action> is not allowed to introduce new
%       <Variable>.
%       An <Action> may also be a higher-order variable.
%       <Func> must be lower case identifier
%
semantic_check_action(tau    , tau    , E, E) :- !.
semantic_check_action(lid(A), lid(A), E, E) :- !.
semantic_check_action(var(X), V, E, E) :- % higher-order action
        member( X=V, E ), !.
semantic_check_action(var(X), _, E, E) :-
        % higher-order action, but undefined
        !,
        errormsg('Undefined action variable: ', X),
        fail.
semantic_check_action(op('`',A), op('`',T), E1, E1) :- !,
        semantic_check_action(A, T, E1, E1).     % see note above

semantic_check_action(func(lid(A),N,Ts), func(lid(A),N,Ts1), E1, E2) :-
        semantic_check_term(Ts, Ts1, E1, E2).


%
% <Eqns>     ::= <Eqn> 'and' <Eqn> ... 'and' <Eqn>
% <Eqn>      ::= <Id> '=' <Actions>
% <Actions>  ::= '{' <Id> ... <Id> '}'
%              | '[' <Id>'/'<Id> ... <Id>'/'<Id> ']'
%
semantic_check_eqns( op('=',Id,Val), Env, [Id=Val|Env] ) :-
        identifier( Id ),
        actions_set( Val ),
        non_member( Id=_, Env ), !.  % should not have been defined
semantic_check_eqns( op('and',Eqn,Eqns), Env1, Env3 ) :-
        semantic_check_eqns( Eqn, Env1, Env2 ),
        semantic_check_eqns( Eqns, Env2, Env3 ).

actions_set( [lid(_)|As] )                   :- pure_actions( As ), !.
actions_set( [op('/',lid(_),lid(_))|As] ) :- rename_actions( As ), !.
actions_set( T ) :- errormsg( 'Invalid action set: ', T).

pure_actions( [] ).
pure_actions( [lid(_)|As] ) :- pure_actions( As ).
```

```
rename_actions( [] ).
rename_actions( [op('/',lid(_),lid(_))|As] ) :- rename_actions( As ).


new_var( [],        [X=V],     X, V ) :- !.
new_var( [X=U|E],   [X=U|E],   X, U ) :- !.
new_var( [Y=U|E1], [Y=U|E2], X, V ) :-
        X \==Y,
        new_var( E1, E2, X, V ).


%% disjoint(+Env1,+Env2,+Env3)
% holds if (Env1-Env3) and (Env2-Env3) are disjoint environments
%
disjoint_env([],_,_) :- !.
disjoint_env(_,[],_) :- !.
disjoint_env( [X=_|E1], [X=_|E2], E3 ) :-
        member(X=_, E3),
        !,
        disjoint_env( E1, E2, E3 ).
disjoint_env( [X=_|E1], E2, E3 ) :-
        \+ member(X=_, E2),
        !,
        disjoint_env( E1, E2, E3 ).

identifier( uid(_) ).
identifier( lid(_) ).



%
% Module:       stmt
%
% Written by:   Philip J. Wiebe (July, 1993)
% Modified by:  Mantis H.M. Cheng (August, 1993)
% Modified by:  Philip J. Wiebe (October, 1993)
%
%
% Description:
%   This module parses a list of tokens into an nested operator
%   expression.
```

```
%


% operator declarations
statement(op_defn(O,T,A,P))    -->
        op_type(T), op_assoc(A), [num(P)],
        {P>=0,P=<1200,!}, op_name(O), [end].

statement(end_of_file)        --> [end_of_file].

% any other input expression defined by operators
statement(E)                   --> expression(E), [end].

%=======================================================================
% Treating terms as ops, an op is declared as
%     op(N,T,O)
% where N is the precedence between 0 and 1200,
%       T is the type of operator: fx, fy, xf, yf, xfx, xfy, yfx, and
%       O is name of the op.
%=======================================================================
%
expression(T)                  --> term(1200,_,T).

term(N,M,T)                    --> term1(N,M1,L), term3(N,M1,L,M,T).

%
% non left-recursive terms
%
term1(_,0,func(F,N,[A|As]))    -->
       function_name(F), ['('], term(999,_,A), arguments(As), [')'],
       {!, length([A|As],N)}.
term1(_,0, PT )                --> ['('], {!}, expression(PT), [')'].
term1(_,0, [A|As] )            -->
       ['{'], term(999,_,A), arguments(As), ['}'], {!}.
term1(_,0, prolog(T))          --> [prolog(T)].
term1(_,0, num(T))             --> [num(T)].
term1(_,0, [])                 --> ['[]'].
term1(_,0, const(T))           --> [const(T)].
term1(_,0, uid(T))             --> [uid(T)].
term1(_,0, var(T))             --> [var(T)].
term1(_,0, lid(T))             --> [lid(T)].
term1(_,0, bool(true))         --> [true].
```

```
term1(_,0, bool(false))          --> [false].
term1(_,0, T)                    --> [T], {keyword(T)}.

term1(_,0,[E|Es])                -->
        ['['], expression(E), expression_list(Es), [']'].
term1(B,N,T)                     --> [O],
        {op_decl(O,fx,N), N=<B,N1 is N-1},
        term(N1,_,T1), {check_sign(O,T1,T)}.
term1(B,N,T)                     --> [O],
        {op_decl(O,fy,N),N=<B},
        term(N,_,T1), {check_sign(O,T1,T)}.

%
% left recursive terms
%
term3(N,LLP,LLPT,M,PT)            -->
        term2(N,LLP,LLPT,LP,LPT),
        term3(N,LP,LPT,M,PT).
term3(_,M,PT,M,PT)                --> [].

term2(N,M1,LPT,M,op(O,LPT))       --> [O],
        {op_decl(O,xf,M),M1<M,M=<N}.
term2(N,M1,LPT,M,op(O,LPT))       --> [O],
        {op_decl(O,yf,M), M1 =< M, M =< N}.
term2(N,M1,LPT,M,op(O,LPT,RPT)) --> [O],
        {op_decl(O,xfx,M),M1 < M, M =< N, N1 is M - 1},
        term(N1,_,RPT).
term2(N,M1,LPT,M,op(O,LPT,RPT)) --> [O],
        {op_decl(O,yfx,M), M1 =< M, M =< N, N1 is M - 1},
        term(N1,_,RPT).
term2(N,M1,LPT,M,op(O,LPT,RPT)) --> [O],
        {op_decl(O,xfy,M), M1 < M, M =< N},
        term(M,_,RPT).


arguments([A|As])        --> [','], term(999,_,A), arguments(As).
arguments([])            --> [].

function_name(uid(F))    --> [uid(F)].
function_name(lid(F))    --> [lid(F)].

expression_list(E)       --> ['|'], {!}, expression(E).
```

```
expression_list([E|Es]) --> [','], {!}, expression(E), expression_list(Es).
expression_list([])     --> [].

op_type(infix)          --> [lid(infix)].
op_type(prefix)         --> [lid(prefix)].
op_type(postfix)        --> [lid(postfix)].

op_assoc(non)           --> [lid(non)].
op_assoc(assoc)         --> [lid(assoc)].
op_assoc(left)          --> [lid(left)].
op_assoc(right)         --> [lid(right)].

op_name(O)              --> [uid(O)].
op_name(O)              --> [lid(O)].
op_name(O)              --> [const(O)].


%
% change the internal sign of a number if necessary.
%
% N.B. If T is a variable then we could get definitions of the form
%      op('-',X) that would have to be evaluated at the time X
%      becomes known.
%
check_sign( O   , T,      op(O,T) ).
```

VITA

Surname: Wiebe | Given Names: Philip John

Place of Birth: Vanderhoof, BC | Date of Birth: July 9, 1969

Educational Institutions Attended

University of Victoria	1987 to 1994
College of New Caledonia	1986 to 1987

Degrees Awarded

B.Sc. (Honours-Major)	University of Victoria	1992

Honours and Awards

NSERC Scholarship	1992–1994
University of Victoria President's Award	1992–1994
ASI Scholarship	1992

Publications

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A TOOL FOR PROTOTYPING CONCURRENT DESIGN SPECIFICATIONS

Author:

Philip John Wiebe

April 15, 1996

Date