CS415
Advanced Software Engineering: Formal Methods

Hugh Anderson
Maths and Computing Science, USP
anderson_h@usp.ac.fj

March 8, 2000

Preface

The FOLDOC[1] dictionary of computing defines software engineering as:

> A systematic approach to the analysis, design, implementation and maintenance of software. It often involves the use of CASE tools. There are various models of the software life-cycle, and many methodologies for the different phases.

The key word in software engineering is *'engineering'* - a largely neglected aspect of software production. Software developers start off with good intentions, but get mired down with administrivia and practical limitations. Very quickly the original *engineered* components of a product become warped, designs deviating from the original intent.

In this part of the course we focus on the use of rigorous methods (*formal methods*) in the specification, construction and checking of software. The structure of the course is a series of topics each exploring some aspect of formal methods.

[1]The Free-On-Line-Dictionary-Of-Computing is found at http://wombat.doc.ic.ac.uk/foldoc/index.html.

Contents

Chapter 1

Introduction

Let us start by looking at the following warranty information:

> *PC Manufacturer warrants that (a) the SOFTWARE will perform substantially in accordance with the accompanying written materials for a period of ninety (90) days from the date of receipt, and (b) any Microsoft hardware accompanying the SOFTWARE will be free from defects in materials and workmanship under normal use and service for a period of one (1) year from the date of receipt.*

and

> *ACCTON warrants to the original owner that the product delivered in this package will be free from defects in material and workmanship for the lifetime of the product.*

The first warranty applies to a large *software* product (Windows NT), the product of hundreds of person-years of software development effort, and containing millions of separate components.

The second warranty applies to a large *hardware* product (An intelligent hub), the product of tens of person-years of software and hardware development effort, and containing millions of separate components.

Taken together, it is clear that the reliability of software is viewed as poor - even by the largest software companies in the world - in comparison with hardware. One explanation often given for this sad state of affairs is that the software is *more complicated* than the hardware, and hence there is more opportunity for failure. However, this explanation is not particularly good - it is clear that even small[1] software is inherently unreliable. We have to recognize that most commercial software in use today contains large numbers of errors, and the software industry *must* improve. Another way of restating this is say that *software engineers* must improve.

Consider the following problem:

[1]A *small* software structure in this context might be one with up to 2,000 lines of code, 200 variables and four programming interfaces.

> The sum of the ages of my three sons is five times the age of the smallest son. The sum of the ages of the two youngest sons is six years younger than the sum of the ages of the two oldest sons. The oldest son has the next birthday, and when he has it, he will be two years older than twice the age of the youngest son. *What are the ages of my three sons?*

If a school child was presented with this problem, he or she would immediately reach for a piece of paper, and represent the problem in an abstract manner[2]:

$$\begin{aligned} a+b+c &= 5*a \\ a+b &= (b+c)-6 \\ (2*a)+2 &= c+1 \end{aligned}$$

Having done this, the set of equations are then reduced using well known methods. The general strategy taught at school is that "*if a problem is too complicated to understand all at once, rewrite it in an abstract notation and then use a set of rules to solve it.*"

For contrast, consider the following problem:

> A file consists of two sorts of records - unprocessed records and processed records. It contains at least one record, and we cannot tell the type of a record without first accessing it. A program randomly accesses two of the records. If they are of different types, the program amalgamates the data in the records, deletes them, and writes a new unprocessed record back to the file. If the records are of the same type, the program processes the data in the records, deletes them, and writes a new processed record back to the file. This process continues until the program cannot get two records from the file. Given a starting state of the file: *Does the program terminate? What is in the file when it terminates?*

If a software engineer is presented with this problem, he or she might '*model it using a small Pascal program*', '*Run it a few times and see what happens*', or perhaps '*Start with a file with one record of each type, then try a bigger file until a pattern emerges*'. (Actual responses!) In this case the software engineer is completely wrong - the problem can be easily abstracted and solved.

When chemists, physicists or biologists meet a problem that is too large or difficult to understand, they turn to mathematics for help. When software engineers meet a problem that is too large or difficult to understand, they try to code it a few different ways to see what happens, or perhaps develop a few subassemblies, or ...

1.1 Errors or bugs?

For many years the software industry has referred to software *errors* as *bugs*. This trivializes the issue, and gives the software developer a false sense of security. ("*Once I've removed these last*

[2]Actually, only my oldest son would do this...

few bugs, it will be perfect!”) In reality, the software developer may stand on very shaky ground: *“Once I’ve removed these last few bugs, it will be:”*

- *a system that has been known to work for 11 minutes continuously... (!)*
- *a system that has no written documentation describing it’s architecture...*
- *a system that no-one else in the world could repair without spending months looking at my code...*
- *a system that still contains code that I put in last month that I don’t know why it is there...*
- *a system that stops occasionally, but I can’t reproduce it so lets call it hardware...*
- *a system that is still using legacy code that noone has checked for about 12 years, but it has worked for a really long time so it must be OK...*
- *a system that is using a commercial windowing package that should be really reliable - well certainly better than code I would produce myself...*

The Internet newsgroup **comp.risks** reports on *“Risks to the Public in the Use of Computer Systems and Related Technology”*, and makes interesting reading. In the January 1999 risks digest we find that:

- *From the Toronto ’Globe and Mail’ (Canada) on 22 Jan 1998: Supposedly confidential records of up to 50,000 Canadians were accidentally left accessible to the general public on the Website of Air Miles, Canada’s second largest customer loyalty program.*

Previous digests have reported that:

- The Arizona lottery never picked the number 9 (a bad random number generator)
- The LA counties pension fund has lost US$1,200,000,000 through programming error.
- Windows NT corrupts filespace and deletes directories under some circumstances.
- A Mercedes 500SE with graceful no-skid brake computers left 200 metre skid marks. A passenger was killed.
- A woman killed her daughter, and attempted to kill herself and her son, after a computer error led to her being told that all her family had an incurable cancer.
- A computer controlled elevator door in Ottawa killed two people.
- An automated toilet seat in Paris killed a child.

- The Amsterdam air-freight computer software crashed, killing giraffes.
- Two Russ Hamilton's had the same birthdate. The wrong one was jailed as a result of a software error.
- A software patch to an unrelated system miraculously added \$10 to \$30 to 300,000 auto tax bills in Georgia.
- The Panasonic children's Internet watchdog software regularly spews out vulgarities.
- General Motors recalled over 1,000,000 cars for a software change to stop erratic air bag deployment.
- The 22nd of August 1999 might be a very dangerous date - don't rely on GPS or the time on that day.

There are thousands more of these sort of stories documented over the last 20 or so years, and in the following section we look at *just one*.

1.2 Therac 25

The Atomic Energy Commission Limited's Therac-25 was a machine programmed to dispense the right amount of radiation to fight cancer. The machine was based on a linear particle accelerator and was first delivered in 1985. It was used until January 1987, during which time there were six instances of extreme harm or death. The Therac-25 was a development over earlier Therac-6 and Therac-20 models, and was the first to rely completely on a computer (the earlier models had hardware interlocks). The software for Therac machines was originally produced by a French company called CGR, and some of the software faults were present in the early models, but gave no problems due to the hardware interlocks.

On the Therac-25, when the software failed, the machine caused severe radiation burns - and instead of normal 200 rad doses, people received an estimated 25,000 rad dose[3].

Occasional peculiar behaviour of the system was observed immediately after commissioning of the first Therac-25s, and even resulted in lawsuits from unhappy patients. AECL responded with modifications to the systems in September 1985, which "*resulted in an improvement in safety of 10,000,000%*" (according to AECL).

Four months later the first death occurred after the following incident at the East Texas Cancer Clinic in Tyler, Texas.

[3]Note that a whole body dose of 500 rad is fatal in 50% of cases.

He ... said something was wrong. She asked him what he felt, and he replied, "fire" on the side of his face... something had hit him on the side of his face, he saw a flash of light, and he heard a sizzling sound reminiscent of fried eggs... He was very agitated and asked "What happened to me, what happened to me?"

Other deaths followed and the systems were eventually shutdown for extensive hardware and software modifications.

The deaths are directly attributable to poor programming. In particular, the software consisted of multiple tasks that accessed shared variables in memory. In the time between testing and setting the value of a shared variable (commonly called a critical section), other tasks could modify it's value. It is likely that the software writers were completely unaware of this type of problem and general methods for their solution.

In Leveson's analysis of the Therac-25 software failure [1], the following general lessons are emphasized:

- Don't put too much confidence in software
- Don't confuse *reliability* with *safety*.
- Design defensively - that is, for the *worst* case.
- Don't focus on *bugs*.
- Be vigilant.

She also outlined the following good software engineering practices:

- Don't do specifications and documentation as an afterthought.
- KISS - Keep It Simple Stupid.
- Testing and analysis at all levels, rigorous quality assurance.
- Careful design of user interfaces.
- Don't assume that re-used software is safe.

1.3 Improving software quality

The following points may help in improving the quality of delivered software:

- **Design software before building it** - You would not pay much for a house without plans or that had not been designed first - why would you pay for software without similar sureties?

- **Reduce the complexity** - By localizing variables, use of modularity, well defined interfaces and so on.
- **Enforce compatibility with design** - Easier to say than to do.

In each of the above points, we can use mechanical (or formal) methods. Other engineering disciplines have embraced formal methods[4], but the software industry (perhaps it should still be called a homecraft) has not yet to any great extent.

Why is this?

- Software engineers are afraid of mathematically based methods.
- A steep learning curve.
- Ignorance of the benefits.
- A belief that there are few tools available.

The use of formal methods twenty years ago was seen as unworkable outside of the classroom, but now there are many well established cases of successful use of formal methods, resulting in products delivered under budget and under time[5].

Hall [2] outlines the following seven myths relating to formal methods:

- Guarantee perfect software
- All about proving programs correct
- Only useful in safety critical systems
- Needs mathematicians
- Increases development cost
- Unacceptable to users
- Not used on large scale systems

Bowen [3] adds seven more:

- Delays development
- No tools available
- Mustn't be mixed with traditional methods
- Only applies to software
- Not needed
- Not supported
- FM people always use FM

[4]When you design a bridge or a multi-storey building, extensive stress and pressure analyses are performed. Hardware engineers regularly use simulators and testers for proposed designs. Most modern VHDL chips are designed using tools that automatically check and verify desired behaviour expressed using mathematical methods.

[5]There are also of course examples of products developed using formal methods delivered over budget and over time, but in general these developments foundered due to other factors - not the 'use of formal methods'.

The above myths are fully recognized as such these days, and formal methods are gradually becoming more common.

Formal software specification techniques are replacing flow charts and pseudo-code. The distinct advantages of formal specifications are:

- The formal software specification can easily feed back modifications to the requirements specification as needed.
- The formal software specification can be mechanically progressed through to a design and even implementation.
- The formal software specification can be analysed, modelled and checked with a range of automated checkers.

In summary: formal techniques reduce the time to product delivery, and improve the products. This is true even without tool support.

1.4 Formal methods in industry

The use of formal methods in industry is at least 19 years old. In at least two cases, products have been developed by competing teams of developers - one set using formal methods, and the other not. In both of these cases, the design phase took about 25% longer with the team using formal methods, but the overall development time was reduced (despite having to learn a new paradigm), and the resultant products were considered qualitatively better.

In the following sections we look at some examples of successful use of formal methods in industry.

1.4.1 CICS

CICS (Customer Information Control System) is a large IBM software product fully licensed at over 40,000 major organizations around the world for managing and processing their business. CICS is in use in the banking industry, airline reservations, stock control and so on. CICS provides OLTP (On Line Transaction Processing) support, and a well developed API for customer applications. The first versions of CICS were in use over 30 years ago, and it has undergone continual improvement and refinement over the years.

From 1982 through to the present day, mathematical methods have been applied to the development of CICS, including heavy use of the Z specification language. Z is used in the specification of all new CICS modules. This process invariably results in extra work in the specification stages of the new products, but this extra work is completely offset by the later reduction in detailed design and coding times. Early CICS Z development was largely done without tool support, but even so the final products are significantly better than other CICS modules - requiring low software service cost after delivery.

Analyses of IBM's use of specification in CICS clearly show that:

- Total development time is reduced
- Product service time is dramatically reduced.

And perhaps more significantly, IBM continue to broaden their use of formal methods.

1.4.2 Chip development

Various groups have performed specification, design and verification of microprocessor systems and subassemblies.

At Inmos, a group developed the T800 FPU (Floating Point Unit) using formal methods. While doing this they uncovered a hitherto unknown fault in the specification of IEEE floating point numbers.

The overall time from beginning design to delivery was reduced from that of similar chips delivered by the same engineers previously. During testing of the first prototype of the T800, a single error was identified - caused by a (human) transcription failure, and no other faults have been found in the chip.

This should be put in perspective - the testing phase of similar chips normally identifies 10 to 100 errors, and lead to 4 to 10 chip mask iterations spread over a year or so. The Inmos T800 was tested, and the second iteration of the mask was the one used in production.

1.4.3 NASA

NASA's **DEEP SPACE 1** mission to Mars launched in December 1998 is heavily reliant on computer software. The *Millenium Remote Agent* software is an AI system written in various languages that sit on top of LISP, and manage the control, guidance and autonomous robotic systems on board. The software is considered mission critical, and has been developed over many years with an essentially unlimited budget, and with *extremely* experienced software engineers.

In 1997, a small inexperienced[6] group used formal methods to analyze components of the software. Most of their time was spent trying to understand the complex systems and find useful ways to model them. Once they had a working model, they quickly found 5 errors, four of which were considered important, and one of these revealing a significant design flaw in the system.

NASA's software engineers:

[6] They were inexperienced in that they had not done this sort of formal analysis before, and had not used the particular formal checker before.

> *You've found a number of bugs that I am fairly confident would not have been found otherwise. One of the bugs revealed a major design flaw... so I'd say you have had a substantial impact...*
>
> *I used to be very skeptical of the utility of formal methods... I believed (and still believe) that it is impossible to prove correctness of software systems. However, what you have been doing is finding places where software violates design assumptions...*
>
> *To me you have demonstrated the utility of this approach beyond any question.*

Lets hope DEEP SPACE 1 makes it!

Chapter 2

Mechanical methods

In this section we briefly look at simple mechanical methods for producing software that exactly match a specification given in some form or other.

2.1 Recursive descent parsing from (E)BNF

The recognition of correct sentences in formal languages (i.e. computer languages) is well understood. The definition of correct sentences is called the *grammar*, and the process of recognition is called *parsing*.

If we express our grammar formally, we can generate programs that recognize correct sentences automatically from the formal specification. In the following example, we limit our grammar to "*one which can be recognized by just looking at the next symbol*". Our language is one which specifies mathematical expressions such as $(3 * 4) + 2$, or $a - (b/3)$. We express this in (E)BNF [5] like this:

- **<math-exp> ::= <term> { '+' <term> | '-' <term> }**

  A <math-exp> is a <term> followed by zero or more occurences of either a '+' followed by a <term> or a '-' followed by a <term>.

- **<term> ::= <factor> { '*' <term> | '/' <term> }**

  A <term> is a <factor> followed by zero or more occurences of either a '*' followed by a <term> or a '/' followed by a <term>.

- **<factor> ::= '(' <math-exp> ')' | <identifier>**

  A <factor> is either a '(' followed by a <math-exp> followed by a ')', or an <identifier>.

The EBNF definition language uses the special symbols '**{**', '**}**', '|', '**::=**' and '**<....>**' to represent various things.

- **The '|' bar represents choice:** A|B means A or B

- **<....> represents a non-terminal:** <A> means that <A> must appear as the left hand side of the BNF expression.

- **<lhs> ::= ..... represents a rule of the grammar:** the <lhs> can be replaced by the right hand side wherever it appears.

- **{ ... } represents iteration:** You may have **...** repeated zero or more times.

This can be automatically translated to:

```
procedure math-exp;
begin
   term;
   iterateexp1 := true;
   while iterateexp1 do
      if nextsym in ['+','-'] then
         case getsym of
            '+' : term;
            '-' : term
         end
      else
         iterateexp1 := false
end;

procedure term;
begin
   factor;
   iterateterm1 := true;
   while iterateterm1 do
      if nextsym in ['*','/'] then
         case getsym of
            '*' : term;
            '/' : term
         end
      else
         iterateterm1 := false
end;

procedure factor;
begin
   if nextsym in ['('] then
      case getsym of
         '(' : begin math-exp; getsym end
      end
   else
      identifier
end;
```

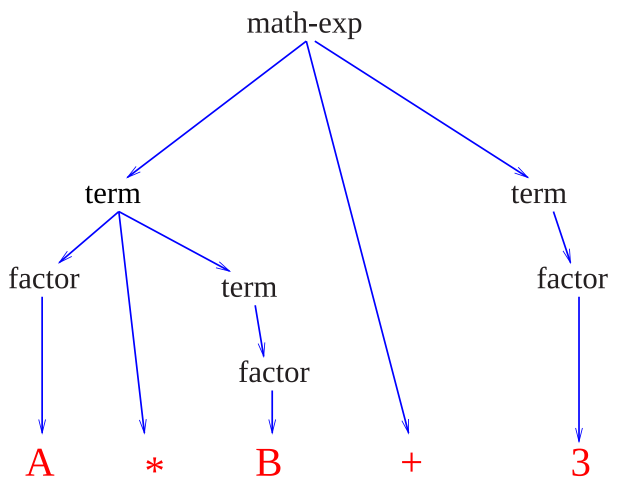


Figure 2.1: Parse tree for arithmetic expression.

Our rules for producing this code are given loosely[1] as follows:

- Create a procedure for each non-terminal (or left hand side).
- A sequence of terms on the right hand side of of a BNF expression generate a sequence of code within the corresponding procedure.
- If a component of the right hand side of of a BNF expression is { ... }, generate code like the following. The termination condition is when none of the internal items are possible.

```
iteration := true;
while iteration do
    ...
```

- If a component of the right hand side of of a BNF expression is 'a' <A> | 'b' <B>, generate code like this:

```
if nextsym in [ 'a','b'] then
   case getsym of
      'a' : A;
      'b' : B
   end
```

- Function *getsym* consumes the next symbol, *nextsym* just looks at it without consuming it.

This automatic technique for generating code from BNF is very easy to do, and generates a *recursive descent* parser. In languages, such as *Pascal*, that require you to declare procedures before using them, you may need to restructure the code:

[1]Yes - I know - I tried a more defined set of rules and it took four pages. In the context of this introduction, this will do...

```
procedure math-exp;
   procedure factor;
      procedure term;
      begin
         <body of term>
      end;
   begin
      <body of factor>
   end;
begin
   <body of math-exp>
end;
```

You can also note that the rules given here correctly parse arithmetic expressions according to the arithmetic rules of precedence - In figure 2.1 A*B+3 correctly evaluates to (A*B)+3.

The principal applications for this technique would be in any system recognizing a 'language' - for example:

- Compilers,
- The acceptable keystrokes on a cash register,
- An input text file format,

2.2 Encoding STDs

The state transition diagram (STD) is another method used for (hopefully) unambiguously describing small systems.

State transition diagrams consist of:

- A set of states, and
- A set of transitions, consisting of
  - events which allow you to change states, and
  - A actions which may be performed when a particular pairing of states and events occur.

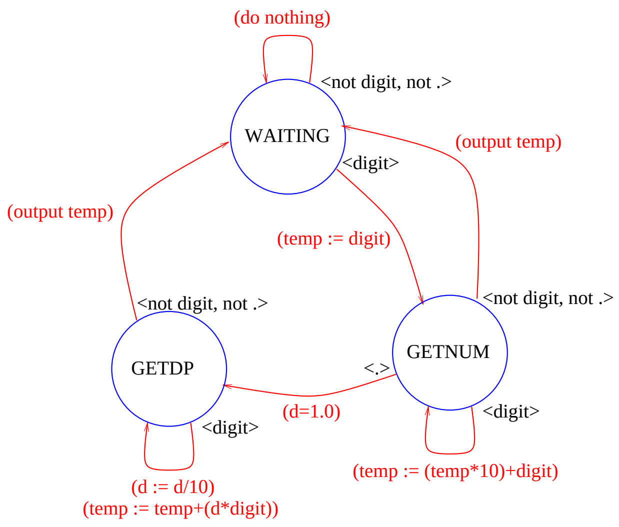


Figure 2.2: State transition diagram for real numbers.

In figure 2.2, we see a state transition diagram. The circles represent the *states*, so our diagram has three *states*. The lines indicate the allowable transitions - in this case there are seven allowable transitions. Each transition arc is marked with an *event* label in diamond brackets which indicates the *event* that will cause the transition, and an *action* which indicates what to do.

We may specify the STD in the following way:

State	**Event**	**Action**	**Next state**
WAITING	digit	temp := digit	GETNUM
	not digit, not .	do nothing	WAITING
GETNUM	digit	temp := (temp*10)+digit	GETNUM
	.	d := 1.0	GETDP
	not digit, not .	output temp	WAITING
GETDP	digit	d := d/10.0; temp := temp+(d*digit)	GETDP
	not digit, not .	output temp	WAITING

We may translate the STD to code by adopting the following set of rules:

- Write a CASE statement for each of the *states*:

```
case state of
   WAITING : ...
   GETNUM  : ...
   GETDP   : ...
end;
```

- Within each *state* item, write a case statement for each of the *events*:

```
GETNUM : case event of
            digit : ...
            '.'   : ...
         otherwise
            ...
         end;
```

- Within each event item, write the *actions* and then set the *state* variable:

```
GETNUM : case event of
            digit : begin
                       temp := (temp*10)+digit;
                       state := GETNUM
                    end;
```

Again we have a simple way to translate a reasonably formal description directly to code. However, the descriptions are fairly detailed, and can themselves be difficult to understand, or be open to misinterpretation.

In the following sections, more abstract formal descriptive methods are introduced, leading to more useful ways of manipulating and verifying software.

Chapter 3

Steps towards abstraction

Much of modern formal specification and proof is performed with the use of propositional and predicate calculi. In this section, we review some relevant terminology. You should keep in mind that the calculi here are no more complex than simple mathematics, and the relevant laws are similar. It may be useful to compare expressions in various logics - only the symbols change:

	Arithmetic	**Propositional**	**Command**	**Set theory**
Commutivity	$a+b=b+a$	$a \wedge b \Leftrightarrow b \wedge a$	$a;\ skip \equiv skip;\ a$	$a \bigcup b = b \bigcup a$
Identity	$a+0=a$	$a \wedge TRUE \Leftrightarrow a$	$a;\ skip \equiv a$	$a \bigcup \{\} = a$

3.1 Propositional logic

In propositional logic we construct expressions which evaluate to one of two values: TRUE or FALSE.

Definition: A proposition P is a statement that is either TRUE or FALSE:

Statement	**Value**	**Type**
All ravens are green	FALSE	proposition
$27 > 1$	TRUE	proposition
$x > y$	?	not a proposition
This sentence is FALSE	?	not a proposition

Compund propositions are constructed from simple propositions using the following symbols:

Name	**Short term**	**Symbol**
Conjunction	AND	$\wedge$
Disjunction	OR	$\vee$
Negation	NOT	$\neg$
Implication	implies	$\Rightarrow$
Equivalence	iff	$\Leftrightarrow$
Equality	equals	$=$

And obey the following laws:

	Law	**Dual Law**
Commutative	$P \wedge Q \Leftrightarrow P \wedge Q$	$P \vee Q \Leftrightarrow P \vee Q$
Associative	$P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$	$P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$
Distributive	$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$	$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$
	$P \wedge TRUE \Leftrightarrow P$	$P \vee FALSE \Leftrightarrow P$
	$P \wedge \neg P \Leftrightarrow FALSE$	$P \vee \neg P \Leftrightarrow TRUE$
De Morgan	$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$	$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
Absorption	$P \wedge (P \vee Q) \Leftrightarrow P$	$P \vee (P \wedge Q) \Leftrightarrow P$

In the propositional calculus, our formulae are either TRUE or FALSE, and can be evaluated using truth tables.

3.2 Predicate logic

In propositional calculus, we are unable to express things such as classes, classes of objects, or set membership. We can for instance state that *all machines are operating* , and that *pc0131 is a machine* , but the obvious conclusion that *pc0131 is operating* can not be drawn, because the calculus cannot express the concept that pc0131 belongs to the class of machines.

In predicate logic, we introduce two quantifiers:

1. The *existential* quantifier - $\exists$ - which means "*There exists*"
2. The *universal* quantifier - $\forall$ - which means "*For all*"

We construct expressions using these quantifiers with the following syntax:

$\forall x : S \bullet P$ - For all x of type S, the predicate P holds.

$\exists x : S \bullet P$ - There exists an x of type S such that the predicate P holds.

These expressions look like simple propositions, but contain variables. If you can replace the variables in an expression and get a proposition, the original expression is called a predicate.

We also have syntaxes for substitution - where we substitute an expression for x wherever it occurs in the body of the predicate:

$P[T/x]$ - The substitution of T for x in predicate P.

The term *bound variable* is used for x in the expressions given above.

We have:

$$\forall x : S \bullet P \wedge Q \Leftrightarrow (\forall x : S \bullet P) \wedge (\forall x : S \bullet Q)$$

And if x doesn't occur in N:

$$\forall x : S \bullet N \Leftrightarrow \exists x : S \bullet N \Leftrightarrow N$$
$$\forall x : S \bullet P \wedge N \Leftrightarrow (\forall x : S \bullet P) \wedge N$$

Along with all the duals of course...

We can also remove quantifiers if x does not occur in T:

$$\exists x : S \bullet x = T \wedge P \Leftrightarrow (T \in S) \wedge P[T/x]$$
$$\forall x : S \bullet x = T \Rightarrow P \Leftrightarrow (T \in S) \Rightarrow P[T/x]$$

3.3 Sets and relations

The concept of a set is a familiar one, with immediate natural understanding of such terms as

- the set of integers, or
- the set of library books.

A set is a defined aggregation of values, and the members of the set are not in any particular order. In formal methods, we limit our use of sets and set variables to *typed* values, in order to avoid Russell's paradox[1].

[1]Simple sets may be defined by just listing their members. However, earlier this century Bertrand Russell observed that some sets are difficult to assess. For example, "*the set of all sets that do not include themselves*". Should it include itself or not? (If it does, then it has an extra member - if it doesn't, then it is incomplete).

Sets:

The following base sets are defined:

Set	Notation	Example
Null or empty set	$\{\}$ *or* $\varnothing$	$\{\}$
Non negative integers	$\mathbb{N}$	$\{0\,1\,2\,3\,...\}$
Integers	$\mathbb{Z}$	$\{...\,-2\,-1\,0\,1\,2\,...\}$

It is common to use capital letters to refer to sets, with small letters used for set variables. Sets may be specified using '$\{item1, item2, ...\}$', or just identified: $[BOOKS]$. A set variable x of type *BOOKS* may be written $x : BOOKS$.

This table introduces some elemental features of set theory:

Feature	Notation	Comment
Membership	$t \in S$	t is an element of S
Inclusion	$S \subseteq T$	$\forall x : S \bullet x \in T$
Strict inclusion	$S \subset T$	$S \subseteq T \wedge S \neq T$
Cartesian product	$S \times T$	$\{(x\,y) \mid x \in S \wedge y \in T\}$
Powerset	$\mathbb{P}\,S$	Set of all subsets of the set S
Intersection	$S \cap T$	$\{x : X \mid x \in S \wedge x \in T\}$
Union	$S \cup T$	$\{x : X \mid x \in S \vee x \in T\}$
Size	$\sharp S$ *or* $\mid S \mid$	

We may tabulate the properties of the new set operations, but the table may soon become quite large. For the first few features:

	Union	**Intersection**	**...**
Commutivity	$A \cup B = B \cup A$	$A \cap B = B \cap A$	...
Associativity	$A \cup (B \cup C) = (A \cup B) \cup C$	$A \cap (B \cap C) = (A \cap B) \cap C$	...
Distribution	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$	...

Relations:

A relation can be modelled by a set of ordered pairs, and so operators defined on sets may be used on relations. There are various notations used for representing relations. We may just list the pairs $\{a_1 \mapsto b_1, a_2 \mapsto b_2, ..., a_n \mapsto b_n\}$, or we can write:

$R : X \leftrightarrow Y$

$X \leftrightarrow Y$ is the set of relations from X to Y - $\mathbb{P}(X \times Y)$

To express the idea that a variable x is related to y by a relation R, we write xRy or $x \mapsto y$, and we write the domain of the relation:

$$\operatorname{dom} R \equiv \{x : X \mid (\exists\, x : X \bullet xRy)\}$$

Set comprehension (or comprehensive specification), may be used to specify a set:

$\{D \mid P \bullet E\}$ *or* $\{D \mid P\}$
D is the signature, or declaration part,
P is the predicate, or constraint part, and
E is the term,

For example:

$\{x : \mathbb{N} \mid x \leq 5 \bullet x^2\} = \{0\,1\,4\,9\,16\,25\}$
$\{x : \mathbb{N} \mid x \leq 5\} = \{0\,1\,2\,3\,4\,5\}$
$\{x : \mathbb{N} \bullet x^2\} = \{0\,1\,4\,9\,16\,25...\}$

We can also use existential and universal quantifiers, with a syntax similar to that given before:

$\exists\, D \mid P \bullet Q$ and $\forall\, D \mid P \bullet Q$

We may negate a quantifier as follows:

$\neg(\exists\, D \mid P \bullet Q) \Leftrightarrow (\forall\, D \mid P \bullet (\neg Q))$, and
$\neg(\forall\, D \mid P \bullet Q) \Leftrightarrow (\exists\, D \mid P \bullet (\neg Q))$

Theorem notation:

A common activity in any formal mathematical transformation is the proof of theorems. Mathematicians start with *givens* (which they call *hypotheses*[2], and assume are *true*), and progress towards conclusions, using deductive rules. Again there are various notations for this. It is common to use the turnstile ($\vdash$) to separate a hypothesis and a conclusion:

[hypothesis]$\vdash$ conclusion

[2]Note that the mathematicians use of the term hypothesis is a little different from that in other areas of science. In other areas, the hypothesis is the *thing you are trying to prove*.

If the conclusion has free variables, they must be declared in the hypothesis - if not it is unnecessary:

$$\vdash \forall x : N \mid x > 5 \bullet x^2 > 25$$

A *proof* is a sequence of theorems derived from hypotheses establishing the conclusion. Each step in the proof must be either a fundamental property, or follow on from previous theorems by using a *rule of deduction* . We use the following notation[3]:

$$\frac{PREMISS(ES)}{CONCLUSION}$$

If our hypothesis is H, and P and Q are predicates, the following rules (and others) may be used:

Modus ponens:	$\frac{H \vdash P \quad H \vdash P \Rightarrow Q}{H \vdash Q}$
(or)	$\frac{[D] \vdash P \Rightarrow Q}{[D \mid P] \vdash Q}$
Reductio ad absurdum:	$\frac{[D \mid P] \vdash (\neg Q) \Rightarrow (\neg P)}{[D \mid P] \vdash Q}$
Induction:	$\frac{H \vdash P(0) \quad H \vdash (\forall k : \mathbb{N} \bullet (P(k) \Rightarrow P(k+1)))}{H \vdash \forall n : \mathbb{N} \bullet P(n)}$

Schema:

The schema notation is a signature/predicate pair, either written as $[S \mid P]$ if the signature and predicate are small, or inside the schema construct:

```
┌─ FirstSchema ──────────────────────────────
│ S
├──────────────
│ P
└────────────────────────────────────────────
```

If we wanted to declare a boolean set BOOL, we can use the following shorthand notation, or the schema (which declares much the same thing).

$$BOOL ::= TRUE \mid FALSE$$

or:

$$[BOOL]$$

[3]The word premiss may also be spelt premise.

Boolean

$true : BOOL$
$false : BOOL$

$true \neq false$
$BOOL = \{true, false\}$

3.4 Lambda calculus

The lambda (λ) calculus is a notation for relating functions and their arguments. It has rules for combining functions.

Definition: A lambda (λ) expression is made up of variables and the three symbols λ, (and).

1. A single variable is a λ-expression.
2. If A is a λ-expression, then λxA is also one.
3. If B and C are λ-expressions, then so is (BC).
4. In λxA, λx is the *bound variable* part, and A is called the *body*.
5. In (BC), B is the *operator* and C is the *operand*.

Associated with this notation is a set of rules (a calculus). The main rule is similar to "*Parameter passing using call by name*".

Definition: In $((\lambda xB)A)$, the effect of the application of the operator λxB to the operand A is the substitution of A for all free occurences of x in B. This rule is also called reduction.

The bound variables in λ-calculus do not imply memory locations as variables in Von-Neumann languages do. They are just used to make expressions easier to read. One of the core theorems of λ-calculus demonstrates that all such variables may be removed. The λ-calculus encompass the range of computable functions:

$\lambda x\,(x+1)$ is the successor function,

$\lambda x\,(0)$ is the zero function

$\lambda xyz\,(y)$ is the identity function

Programming languages based on λ-calculus are called *reduction*, *functional* or *applicative* languages.

3.5 Assertions, declarations, specifications, code...

Each of the above mathematical formalisms has been used as the basis for one or more programming languages. Common to many of these languages is a reduction in the gap between the specification and the implementation of a problem. It is perhaps easier to show this by example, so here are a few examples.

3.5.1 Predicate logic: prolog

There are various automatic ways of transforming predicates into statements[4]. One of these automatic methods [4] is the basis of the *prolog engine* - a different sort of computer, and a different sort of computer language.

When you use a *prolog* system, your program statements are turned into predicate logic theorems, which are then solved if possible:

	Program statement	**Internal form**	**Machine response**	**Comment**
1	f(hugh,boyd)	hugh F boyd	OK	Accepted a TRUE proposition
2	f(hugh,ford)	hugh F ford	OK	Accepted a TRUE proposition
3	f(hugh,judge)	hugh F judge	OK	Accepted a TRUE proposition
4	f(hugh,*x)	hugh F *x	boydfordjudge	Solved for x
5	f(ford,*x)	ford F *x	FALSE	Solved for x
6	gf(*x,*z) if f(*x,*y) and f(*y,*z)	$\forall x \forall y \forall z \bullet xFy \wedge yFz \Rightarrow xGz$	OK	Paternal grandfather proposition
7	gf(*x,*z) if f(*x,*y) and m(*y,*z)	$\forall x \forall y \forall z \bullet xFy \wedge yMz \Rightarrow xGz$	OK	Maternal grandfather proposition
8	f(bob,hugh)	bob F hugh	OK	Accepted a TRUE proposition
9	gf(*x,judge)	*x G judge	bob	Solved for x

In this example, we have

- written nine lines of *prolog*, of which
- four of the lines created data (1,2,3 & 8),
- two lines described the grandfather relationship (6 & 7), and
- the other three lines queried the resulting structure.

There are no extra variables or constants or program constructs except for those needed in the system. Here is another example of equally terse code implementing a stack in *prolog*. This code can be viewed as a *specification* of a stack[5]:

[4]This could perhaps be phrased as "*solving predicate logic theorems*".

[5]Can you think of any other way to specify the behaviour of a stack?

```
top(push(*x,*y),*x);
is(pop(push(*x,*y)),*y);
empty(S,TRUE);
empty(push(*x,*y),FALSE);
```

Then, if we query the specification:

```
empty(S,*x)-w("The new stack is empty: ")-w(*x)-line;
empty(push(apples,S),*x)-w("If we push apples: ")-w(*x)-line;
empty(*y,*x)-is(pop(push(apples,S)),*y)-w("And then pop them: ")-w(*x)-line;
top(*y,*x)-is(push(apple,push(pear,S)),*y)-
w("Push a pear then an apple, on top is: ")-w(*x)-line;
top(*y,*x)-is(pop(push(apple,push(pear,S))),*y)-
w("Push a pear then an apple, pop once, on top is: ")-w(*x)-line;
```

We get the following results:

```
The new stack is empty: TRUE
If we push apples: FALSE
And then pop them: TRUE
Push a pear then an apple, on top is: apple
Push a pear then an apple, pop once, on top is: pear
```

In this example, we can see that there is very little difference between a specification of a stack, and its implementation. Compare it with the following partial stack implementation in a conventional Von-Neumann style language:

```
Const
   StackSize = 100;
Type
   Stack : array [ 1..StackSize ] of integer;
Var
   S : Stack;
   index : 1..StackSize;

procedure push(item:integer);
begin
   index := index+1;
   S[index] := item
end;
...
```

A summary of comparisons:

Von-Neumann Program	**Prolog**
Assure correctness by mentally executing	Correct by definition
Extra variables (i)	No extra variables
Names data and results (stack,index)	Works on *any* stack
Must know size of stack	Works on *any size* stack
Can only stack integers	Stacks anything
Familiar	Not familiar

3.5.2 Lambda calculus: FP

One example of a language based on the λ-calculus is Backus's FP language. The following code defines a function which adds together two arrays. It uses two other functions - AA (Apply to all) and trans (Transpose):

```
def Add = (AA,+).trans
```

Read backwards - say "transpose the arrays and add each item".

If we apply this function to two arrays, our FP machine successively rearranges the expression:

```
{Add:(1,2,3,4),(4,3,5,9)
{((AA,+).trans):(1,2,3,4),(4,3,5,9)}
{(AA,+):trans:(1,2,3,4),(4,3,5,9)}
{(AA,+):(1,4),(2,3),(3,5),(4,9)}
{+:(1,4),+:(2,3),+:(3,5),+:(4,9)}
5,5,8,13
```

Note that at each stage of the computation of the exression, we have a complete readable FP expression. Compare this with the following Von-Neumann program:

```
Procedure Add;
Var
   i : Integer;
begin
   for i:=1 to size do
      C[i] := A[i]+B[i]
end;
```

A summary of comparisons:

Von-Neumann Program	**Reduction Language Program**
Assure correctness by mentally executing	Correct by definition
Extra variables (i)	No extra variables
Names data and results (A,B,C)	Works on *any* data
Must know size of data	Works on *any size* data
Machine execution not related to program	Machine execution related to program
Parallelism not easily identified	Parallelism easily identified
Familiar	Not familiar
Wordy	Terse

3.5.3 Lambda calculus: LISP

Another functional programming language is LISP (Short for **Lis**t **P**rocessing language). LISP expressions look just like λ-expressions except that the λ is replaced by the word LAMBDA, and *infix* expressions must be given in *prefix* form:

λ-expression	LISP
$\lambda x\,(x+x)$	(LAMBDA x (PLUS x x))
$(\lambda x\,(x+x))\ 10$	(LAMBDA x (PLUS x x)) 10

LISP is often hard to read due to the large number of brackets in LISP expressions, and more readable alternatives to LISP are often used:

English	LISP	Burge's language	Pascal

The HCF of x and y
is y if x is 0, or else it is
the HCF of the
remainder of $\frac{y}{x}$ and x
	(DEF HCF(X Y)
	(COND (EQ X 0) Y
	(HCF (REM (X Y) X))))
		$hcf\,(x,y) \equiv x = 0 \rightarrow y;$
		hcf(y mod x,x)
			function hcf(x,y:integer):integer;
			begin
			if x=0 then
			hcf := y
			else
			hcf := hcf(y mod x,x)
			end

The FAC of a number x
is 1 if x is 0, or else it is
x times the FAC of $x-1$
	(DEF FAC(X)
	(COND (EQ X 0) 1
	(TIMES X FAC (MINUS X 1))))
		$fac(x) \equiv x = 0 \rightarrow 1;$
		x*fac(x-1)
			function fac(x:integer):integer;
			begin
			if x=0 then
			fac := 1
			else
			fac := x*fac(x-1)
			end

We can observe the reduction of the LISP expression for the factorial of 2:

```
FAC 2
COND ((EQ X 0) 1)(TIMES X FAC (MINUS X 1)) 2                                Simplification/Use definition
COND ((EQ 2 0) 1)(TIMES 2 FAC (MINUS 2 1))                                  Reduction
COND (FALSE 1) (TIMES 2 FAC (1))                                            Simplification
(TIMES 2 FAC (1))                                                           Simplification
(TIMES 2 COND ((EQ X 0) 1)(TIMES X FAC (MINUS X 1))1)                       Simplification/Use definition
(TIMES 2 COND ((EQ 1 0) 1)(TIMES 1 FAC (MINUS 1 1)))                        Reduction
(TIMES 2 COND (FALSE 1) (TIMES 1 FAC (0)))                                  Simplification
(TIMES 2 (TIMES 1 FAC (0)))                                                 Simplification
(TIMES 2 (TIMES 1 COND ((EQ X 0) 1)(TIMES X FAC (MINUS X 1))0)))            Simplification/Use definition
(TIMES 2 (TIMES 1 COND ((EQ 0 0) 1)(TIMES 0 FAC (MINUS 0 1)))               Reduction
(TIMES 2 (TIMES 1 COND (TRUE 1) (TIMES 0 FAC (-1)))                         Simplification
(TIMES 2 (TIMES 1 1))                                                       Simplification
(TIMES 2 1)                                                                 Simplification
2                                                                           Canonical or normal form
```

Note that:

- At each point of the execution of the LISP machine we have a LISP expression.

- There are no von-Neumann style variables
- There are only two operations - reduction and simplification

We can also perform program transformations with the same ease found in the program execution. If we assumed that we had the following three LISP functions:

S fgx = fx(gx)

B fgx = f(gx)

C fgx = f x g

And the following pure function (i.e. it has no variables or parameters):

(DEF FACV

S (C (B COND (EQ 0)) 1 0 (S TIMES (B FAC (C MINUS 1))))

We can transform it into the LISP FAC function as follows:

```
FACV N
S (C (B COND (EQ 0)) 1 0 (S TIMES (B FACV (C MINUS 1 ))) N
C (B COND (EQ 0)) 1 N (S TIMES (B FACV (C MINUS 1 )) N
(B COND (EQ 0) N 1 )(TIMES N (B FACV (C MINUS 1 )) N
(COND ((EQ 0 N) 1 )(TIMES N (FACV (MINUS N 1 ))) N
```

What have we just done?

- We have demonstrated the equivalence of two different LISP programs
- Our original program was a pure function (no variables), our final one has an N variable.
- The proof was done without stepping outside of the language.

Chapter 4

Traditional program proof

Most software engineers reason about their programs in some form or other. In this section, we develop the form of a program proof.

We must first make clear what is meant by *correctness*. In this context, we always mean "*correctness with respect to a specification*". The program may have glaring errors which cause all output to be corrupt, but if the program matches the specification, it is deemed to be *correct*.

We break down our definition of program correctness into two sorts:

Partial correctness: The implementation matches the specification. When our program runs, it gives a correct result *if it gives a result*. However it is possible for our program to run and just never give an *incorrect* result.

Total correctness: The implementation is partially correct, *and* it terminates.

The proof methods for termination are normally quite different from the proof methods for partial correctness, and so breaking our concept of correctness into two parts allows us to prove parts of our program independantly.

Sometimes it is extremely hard to prove either.

```
while x>1 do
   if even(x) then
      x := x/2
   else
      x := (3*x)+1
```

Question: Under what initial conditions does the above program segment terminate?

4.1 Informal reasoning about programs

With a *mathematical* theorem we argue that “Given a specified hypothesis P, the conclusion Q is true” ($P \vdash Q$). We use the same style for reasoning about the correctness of programs, using the terminology *precondition* to refer to P, and *postcondition* to refer to Q.

The precondition and postcondition of a program segment can be viewed as a *specification* of the program segment.

```
sample := SomeNumberLargerThan2;
while sample>=2 do
   sample := sample-2
```

Informally, we reason that:

> *If* `sample` *starts off even, it remains even. If* `sample` *starts off odd, it remains odd.*
> *We can say that the evenness or oddness (parity) is an invariant of the loop.*
> *After executing the loop,* `sample` *will be less than 2, and will be of the same parity as the original.*

4.2 Formal reasoning about programs

If we have $\{P\}$ $segment1$ $\{Q\}$ $segment2$ $\{R\}$[1], then:

- If $\{R\}$ is our correct final state, then we are interested mostly in $\{Q\}$, the set of states that result in $\{R\}$.
- We are concerned also with $\{P\}$, the weakest precondition for $(segment1;\ \ segment2)$ which results in $\{R\}$.

We can define the weakest precondition *wp* as the set of initial states that guarantee the postcondition.

$$\begin{array}{rcl} wp.(segment2).\{R\} & = & \{Q\} \\ wp.(segment1).\{Q\} & = & \{P\} \\ wp.(segment1;\ \ segment2).\{R\} & = & \{P\} \end{array}$$

- if $\{P\}$ is $\{TRUE\}$, then all states lead to $\{R\}$.

[1] In this section, the terminology derives from Hoare, Floyd and Dijkstra.

```
{even}                          {P}
while sample>=2 do                while (R)
   sample := sample-2                Z  (where {P} Z {P})
                                  endwhile
{even ∧ sample < 2}            {P ∧ ¬R}
```

Most programmers reason about their programs in this way even if they have not met Hoare's terminology, but with larger programs, the reasoning becomes much harder. It takes quite a while to ensure yourself of the correctness or otherwise of the following program fragment for ensuring that two processes do not simultaneously enter a critical section.

```
procedure P(boolean i);
begin
   want[i] := true;
   exit := false;
   while not exit do
      begin
         if turn<>i then
            begin
               while not want[i] do;
               turn := i;
            end
         else
            if turn=i then
               exit := true
      end;
   want[i] := false
end;
```

4.3 Simple imperative language

The following simple imperative language is used when reasoning about programs.

Program	::=	*abort*	
		$\mid$ *skip*	
		$\mid x := E$	(Assignment)
		$\mid seg;\ seg$	(Sequence)
		$\mid$ *if B then seg*1 *else seg*2 *fi*	(Choice)
		$\mid seg1 \sqcap seg2$	(Non-deterministic choice)
		$\mid (\mu X.C)$	(Recursion)

We have not included iteration, as it is just a simple case of recursion:

- $do\,B \rightarrow seg\,od \equiv (\mu X.if\ B\ then\ (seg;\ \ X)\ else\ skip\ fi)$

We also commonly shorten:

- $x := E_1 \sqcap x := E_2 \sqcap$ ... to $x \in \{E_1, E_2, ...\}$.

The above language has associated with it a weakest precondition semantics:

$$
\begin{array}{lcl}
wp.abort.Q & \equiv & false \\
wp.skip.Q & \equiv & Q \\
wp.(x := E).Q & \equiv & Q[x := E] \\
wp.(seg1;\ seg2).Q & \equiv & wp.seg1.(wp.seg2.Q) \\
wp.(if\ B\ then\ seg1\ else\ seg2\ fi).Q & \equiv & (B \wedge wp.seg1.Q) \bigvee (\neg B \wedge wp.seg2.Q) \\
wp.(seg1\ \sqcap\ seg2).Q & \equiv & wp.seg1.Q \wedge wp.seg2.Q \\
\text{... and so on ...} & &
\end{array}
$$

One case associated with the use of the iterative construct is commonly used in proofs, and is commonly called the *loop invariance theorem:*

- $\{P\}\, do\, B \rightarrow seg\, od\, \{P \wedge \neg B\} \Leftrightarrow wp.(do\, B \rightarrow seg\, od).\{P \wedge \neg B\} := \{P\}$

4.4 Example

In RSA encryption and authentication, we often have to solve large integer equations. The numbers involved may easily be larger than the number of molecules in the universe (about 10^{80}).

Question - What is the remainder after you divide 111 into 67^{77}?

One solution to this sort of problem is the following code fragment, which calculates $C = mod(P^Q, N)$ for all $P, Q, N > 0$.

```
c := 1;
x := 0;
while x<>Q do
   x:=x+1;
   c:=mod(c*P,N)
end
```

We can write this in our language as:

```
Line 1:  c := 1;
Line 2:  x := 0;
Line 3:  do x<>Q → x:=x+1; c:=mod(c*P,N) od
```

We would like our final state to be $c = mod(P^Q, N)$, and looking at the final line, we notice that it is a *do...od*, so we rewrite the final state to follow the form found in the loop invariance theorem:

$$\{c = mod(P^x, N) \,\wedge\, x = Q\} \,\Leftrightarrow\, \{c = mod(P^Q, N)$$

If the assignments in line 3 of our program leave $c = mod(P^x, N)$ unchanged[2], then using the loop invariance theorem:

$$wp.(line\,3).\{c = mod(P^Q, N)\} \,\equiv\, \{c = mod(P^x, N)\}$$

Now we know the weakest precondition for line 3, we can find the weakest precondition for line 2:

$$wp.(line\,2).\{c = mod(P^x, N)\} \,\equiv\, c = \{mod(P^0, N)\}$$

Now we know the weakest precondition for line 2, we can find the weakest precondition for line 1, and hence the whole program fragment:

$$wp.(line\,1).\{c = mod(P^0, N)\} \,\equiv\, \{1 = mod(P^0, N)\}$$

and of course, $\{1 = mod(P^0, N)\} \equiv \{true\}$. This argument shows that the weakest precondition for our whole program segment is $\{true\}$, but there is one step that we have not justified - we must be sure that line 3 leaves $c = mod(P^x, N)$ unchanged.

We can just apply the same reasoning to the smaller program fragment within the *do..od* loop:

```
                        {c = mod(P^x, N)}
x := x+1;
                        {c = mod(P^{x-1}, N)}
c := mod(c*P,N)
                        {c = mod(mod(P^{x-1}, N) * P, N)}   ⇒   {c = mod(mod(P^x, N) + (K * N), N)}
                                                                {c = mod(P^x, N)}
```

We have proved each step of the program and demonstrated that the desired postcondition will be reached from all initial states ($\{true\}$). This is partial (not total) correctness.

[2]We'll come back to this step in the argument a little later...

Chapter 5

Probabilistic proof

Recently, the traditional predicate transformer approach given in Chapter 4 has been extended to include *probabilistic* choice. The probabilistic imperative language is summarized below.

$$\begin{array}{lcll}
\textit{Program} & ::= & \textit{abort} & \\
 & & \mid \textit{skip} & \\
 & & \mid x := E & \text{(Assignment)} \\
 & & \mid \textit{seg};\ \textit{seg} & \text{(Sequence)} \\
 & & \mid \textit{if } B \textit{ then seg}1 \textit{ else seg}2 \textit{ fi} & \text{(Choice)} \\
 & & \mid \textit{seg}1 \sqcap \textit{seg}2 & \text{(Non-deterministic choice)} \\
 & & \mid \textit{seg}1\ {}_p\!\bigoplus_{1-p}\ \textit{seg}2 & \text{(Probabilistic choice)} \\
 & & \mid (\mu X.C) & \text{(Recursion)}
\end{array}$$

Probabilistic choice between three or more choices can be expressed as a nested binary choice:

$$\textit{prog}_0\ {}_{p0}\!\bigoplus (\ \textit{prog}_1\ {}_{\frac{p1}{1-p0}}\!\bigoplus\ \ldots\ )$$

or with this shorthand: $\textit{prog}_0$ @ p_0, $\textit{prog}_1$ @ p_1, ...

Traditional predicates may be viewed as functions from program states to $\{0, 1\}$ (or $\{\textit{false}, \textit{true}\}$ if you prefer). To deal with probabilistic choice, we generalize the predicates as functions from states to $\Re_{\geq}$.

In a traditional predicate:

$$wp.(n := 1 \sqcap n := 2).(n = 2) \quad \equiv \quad \textit{false}$$

In our generalized predicates:

$$wp.(n := 1\ {}_{\frac{1}{2}}\!\bigoplus n := 2).(n = 2) \quad \equiv \quad \tfrac{1}{2}$$

The weakest precondition semantics for our probabilistic language:

$$
\begin{array}{lcl}
wp.abort.Q & \equiv & \varnothing \\
wp.skip.Q & \equiv & Q \\
wp.(x := E).Q & \equiv & Q[x := E] \\
wp.(seg1;\ seg2).Q & \equiv & wp.seg1.(wp.seg2.Q) \\
wp.(if\ B\ then\ seg1\ else\ seg2\ fi).Q & \equiv & (B * wp.seg1.Q) + (\neg B * wp.seg2.Q) \\
wp.(seg1 \sqcap seg2).Q & \equiv & wp.seg1.Q \sqcap wp.seg2.Q \\
wp.(seg1\ {}_P\!\bigoplus seg2).Q & \equiv & P * wp.seg1.Q + (1-P) * wp.seg2.Q \\
\text{... and so on ...} & &
\end{array}
$$

5.1 Example

The principal use of the Monty Hall problem is to demonstrate how angry people get when they do not agree with the answer (!)

Statement of the problem:

- You are a contestant in a TV show asked to select one of three curtains - we refer to your curtain as *cc* - the contestant curtain.
- Behind one of the curtains is a prize - we refer to this curtain as *pc* - the prize curtain.
- The announcer for the show knows where the prize is located, and opens one of the other two curtains, showing you that the prize is not behind it. We refer to this curtain as *ac* - the announcer curtain.
- You are then asked if you want to change your mind - you can either stick to your original curtain, or change to the other curtain. Which[1] should you do?

A program model of Monty Hall[2]:

line 1:		$pc :\in \{A, B, C\}$
line 2:		$cc := (A@\frac{1}{3}, B@\frac{1}{3}, C@\frac{1}{3})$
line 3:		$ac :\notin \{pc, cc\}$
line 4:		$if\ change\ then\ (cc :\notin \{cc, ac\})\ else\ skip\ fi$

If we are to *win* the Monty Hall game, the desired post condition is $[pc = cc]$, and we just follow the same sort of argument as given before.

We work backwards through the program establishing the wp for line 4, then line 3 and so on:

[1]To cut a long story short, your probability of winning increases from $\frac{1}{3}$ to $\frac{2}{3}$ if you change curtains. No arguments please.

[2]This program is from Carroll Morgan [7].

$$wp.(line\,4).[pc = cc]$$

$$\Rightarrow \quad wp.(if\ change\ then\ (cc :\notin \{cc, ac\})\ else\ skip\ fi).[pc = cc]$$

$$= \quad [change] * [\{ac, cc, pc) = \{A, B, C\}] + [\neg change] * [pc = cc]$$

$$and\ then \quad wp.(line\,3).([change] * [\{ac, cc, pc) = \{A, B, C\}] + [\neg change] * [pc = cc])$$

$$\Rightarrow \quad wp.(ac :\notin \{pc, cc\}).([change] * [\{ac, cc, pc) = \{A, B, C\}] + [\neg change] * [pc = cc])$$

$$= \quad [change] * [pc \neq cc] + [\neg change] * [pc = cc]$$

$$and\ then \quad wp.(line\,2).([change] * [pc \neq cc] + [\neg change] * [pc = cc])$$

$$= \quad [\tfrac{change}{3}] * ([pc \neq A] + [pc \neq B] + [pc \neq C]) + \tfrac{2}{3} * [\neg change] * ([pc = A] + [pc = B] + [pc = C])$$

$$= \quad [\tfrac{change}{3}] * 2 + \tfrac{1}{3}[\neg change]$$

$$= \quad \tfrac{1+change}{3}$$

The initial non-deterministic choice of line 1 leaves this *wp* unchanged, so we can say that the weakest precondition for the program segment is $\frac{1+change}{3}$[3].

[3]We should *really* change curtains!

Chapter 6

Refinement

In Chapter 4, the Floyd-Hoare notation was introduced:

$$\{P\}\, prog\, \{Q\}$$

In this notation, $\{P\}$ represents a precondition and $\{Q\}$ the postcondition. The pair $\{P\}$,$\{Q\}$ may be viewed as a specification of *prog*, and the (new) notation $[P, Q]$ is used for a specification of this form.

An example of this might be the following specification:

$$[true, c = mod(P^Q, N)]$$

The program given in Section 4.4 (page 32) has been shown to meet this specification. In this chapter we introduced the reverse process - starting from a specification, we derive a program which implements the specification.

Refinement is a process of refining a specification S to an implementation I by applying a series of transformations that preserve the specification. We write:

$$S \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq \ldots \sqsubseteq S_n \sqsubseteq I$$

The $\sqsubseteq$ symbol refers to refinement, and our resultant implementation I is guaranteed to match its specification S.

At each stage of the process, we have a choice of possible refinements, but not all refinements will lead to an implementation. However, the process is a mechanical one, and various heuristics can be applied to improve the choice of refinements at each stage.

6.1 Refinement laws

The refinement process exhibits the following two properties:

Safety: $pre\,S \vdash pre\,I$ - We can rephrase this as:

> "The precondition for S must also be an acceptable precondition for I".
>
> The *safety* property indicates that refinement can safely *weaken*[1] the precondition.

Liveness: $(pre\,S) \wedge I \vdash S$ - We can rephrase this as:

> "Given the precondition for S, the behaviour of I must be allowed by S".
>
> The *liveness* property indicates that refinement can safely *strengthen* the postcondition.

These are our proof obligations if we assert that $S \sqsubseteq I$. We must ensure both *safety* and *liveness*.

The refinement table below gives some sample laws that preserve the specification and can be used to derive a program.

Law	**Specification**	**refined by ...**	**Code**	**Caveat**
Skip	$[P,\ P]$	$\sqsubseteq$	`skip`	
Assignment	$[P[E/V],\ P]$	$\sqsubseteq$	`V := E`	
Derived assignment	$[P,\ Q]$	$\sqsubseteq$	`V := E`	if $P \Rightarrow Q[E/V]$
Precondition weakening	$[P,\ Q]$	$\sqsubseteq$	$[R,\ Q]$	if $P \Rightarrow R$
Postcondition strengthening	$[P,\ Q]$	$\sqsubseteq$	$[P,\ R]$	if $R \Rightarrow Q$
Sequencing	$[P,\ Q]$	$\sqsubseteq$	$[P,\ R]$; $[R,\ Q]$	

While	$[P,\ P \wedge \neg S]$	$\sqsubseteq$	`while` S `do`
			 `begin`
			 $[P,\ P]$
			 `end`	

If	$[P,\ Q]$	$\sqsubseteq$	`if` S `then`
			 $[P \wedge S,\ Q]$
			 `else`
			 $[P \wedge \neg S,\ Q]$	

6.2 Example refinement

The specification $[X \geq 0,\ f = fac(X)]$ specifies a factorial function, and an implementation can be derived as follows:

[1] If $P \Rightarrow Q$, we say that P is *stronger* than Q, and Q is *weaker* than P.

	Derived code	**Specification obligations remaining**	**Operation**
		$[X \geq 0,\ f = fac(X)]$	Rewrite
$\sqsubseteq$		$[X \geq 0,\ f = fac(x) \wedge x = X]$	Sequencing
$\sqsubseteq$		$[X \geq 0,\ f = 1 \wedge X \geq \varnothing]$; $[f = 1 \wedge X \geq \varnothing,\ f = fac(x) \wedge x = X]$	Assignment
$\sqsubseteq$	`f:=1;`	$[f = 1 \wedge X \geq \varnothing,\ f = fac(x) \wedge x = X]$	Sequencing
$\sqsubseteq$	`f:=1;`	$[f = 1 \wedge X \geq \varnothing,\ f = 1 \wedge X \geq \varnothing \wedge x = 0]$; $[f = 1 \wedge X \geq \varnothing \wedge x = 0,\ f = fac(x) \wedge x = X]$	Assignment
$\sqsubseteq$	`f:=1;` `x:=0`	$[f = 1 \wedge X \geq \varnothing \wedge x = 0,\ f = fac(x) \wedge x = X]$	Precondition weakening
$\sqsubseteq$	`f:=1;` `x:=0`	$[f = fac(x) \wedge X \geq \varnothing,\ f = fac(x) \wedge x = X]$	Postcondition strengthening
$\sqsubseteq$	`f:=1;` `x:=0`	$[f = fac(x) \wedge X \geq \varnothing,\ f = fac(x) \wedge x = X \wedge X \geq \varnothing]$	While
$\sqsubseteq$	`f:=1;` `x:=0;` `while x<>X do` `begin` `end`	$[f = fac(x) \wedge X \geq \varnothing,\ f = fac(x) \wedge X \geq \varnothing]$	Sequencing
$\sqsubseteq$	`f:=1;` `x:=0;` `while x<>X do` `begin` `end`	$[f = fac(x) \wedge X \geq \varnothing,\ f = fac(x-1) \wedge X \geq \varnothing]$; $[f = fac(x-1) \wedge X \geq \varnothing,\ f = fac(x) \wedge X \geq \varnothing]$	Assignment
$\sqsubseteq$	`f:=1;` `x:=0;` `while x<>X do` `begin` `x:=x+1` `end`	$[f = fac(x-1) \wedge X \geq \varnothing,\ f = fac(x) \wedge X \geq \varnothing]$	Derived assignment
$\sqsubseteq$	`f:=1;` `x:=0;` `while x<>X do` `begin` `x:=x+1;` `f:=x*f` `end`		

In a similar fashion, we could derive other implementations using a different ordering of refinements.

Chapter 7

Z

Z is a specification language developed by the Programming Research Group at Oxford University around 1980. It can describe and model a wide range of computing systems.

Z is based on axiomatic typed set theory and first order predicate logic. It is written using many non-ASCII symbols, and Z specifications should normally be properly typeset using a typesetting language like LaTeX[1].

Tools for analysis and animation of Z specifications normally work directly on the LaTeX files, ignoring extra non-Z parts of the file.

7.1 LaTeX

A first LaTeX document is given below:

```
\documentclass[12pt]{article}
\begin{document}
A very short document. Tradition dictates that we include “Hello World”!
\end{document}
```

This document may be processed into a device independant form (a dvi file) with the following command:

```
opo 427% latex first.tex
   This is TeX, Version 3.14159 (C version 6.1)
   (first.tex
      LaTeX2e <1996/06/01> Hyphenation patterns for english, german, loaded.
      (/usr/local/teTeX/texmf/tex/latex/base/article.cls
      Document Class: article 1996/05/26 v1.3r Standard LaTeX document class
      (/usr/local/teTeX/texmf/tex/latex/base/size12.clo))
      No file first.aux.
      [1] (first.aux) )
   Output written on first.dvi (1 page, 244 bytes).
   Transcript written on first.log.
opo 428%
```

[1]Yes - it is written LaTeX!

This creates a file (first.dvi) which may be directly viewed using a dvi viewer:

```
opo 428% xdvi first.dvi
Note: overstrike characters may be incorrect.
opo 429%
```

We can also convert this dvi file to a postscript file for printing:

```
opo 429% dvips -o first.ps first.dvi
This is dvipsk 5.58f Copyright 1986, 1994 Radical Eye Software
' TeX output 1999.04.30:1056' -> first.ps <tex.pro>. [1]
opo 430%
```

And then we can view this:

```
opo 430% gv first.ps
```

7.2 LaTeX and Z

When using Z in a document, we use special Z style files which contain standard styles for each of the Z constructs. One of the main constructs is the Z schema, which ties together declarations with predicates that constrain them:

```
\documentclass[12pt]{article}
\usepackage{oz}
\begin{document}
  First we introduce the basic type we will use:
  \begin{zed}
     [Student]
  \end{zed}

   And then our first schema:
  \begin{schema}{Class}
     enrolled, tested: \power Student
  \where
     \# enrolled \leq size \\
     tested \subseteq enrolled
  \end{schema}
\end{document}
```

When we run this through LaTeX, we get something like the following:

First we introduce the basic type we will use:

$[\mathit{Student}]$

And then our first schema:

Class

$\mathit{enrolled}, \mathit{tested} : \mathbb{P}\,\mathit{Student}$

$\#\mathit{enrolled} \leq \mathit{size}$
$\mathit{tested} \subseteq \mathit{enrolled}$

Many of the special symbols used in Z specifications are written with short mnemonics. The following table shows some of the more common ones.

Notation	**LaTeX**	**Symbol**
Power Set	`\pset`	$\mathbb{P}$
Finite Subset	`\finset`	$\mathbb{F}$
Natural numbers	`\nat`	$\mathbb{N}$
Not equal	`\neq`	$\neq$
For all	`\all`	$\forall$
There exists	`\exists`	$\exists$
Union	`\uni`	$\cup$
Mapping	`\map`	$\mapsto$
Domain	`\dom`	dom
Subset	`\subseteq`	$\subseteq$

Paul King's "*Printing Z and Object-Z LaTeX documents*" contains a complete list.

7.3 Z specifications

A Z specification normally consists of free format text descriptions, and the following formal descriptions:

1. Declarations of the basic types (or sets) along with global variables, and general rules (axioms) related to the system to be specified. First - a basic type:

   $[\mathit{Student}]$

   and an axiomatic definition for the system:

   $\mathit{maxstudents} : \mathbb{N}$

   $\mathit{maxstudents} \leq 100$

2. Abstract state schema describing the major system components:

   Class

   $\mathit{enrolled}, \mathit{tested} : \mathbb{P}\,\mathit{Student}$

   $\#\mathit{enrolled} \leq \mathit{size}$

   $\mathit{tested} \subseteq \mathit{enrolled}$

3. Schemas for initialization:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{ClassInit}} \\ \hline
\textit{Class}' \\ \hline
\textit{enrolled}' = \varnothing \\
\textit{tested}' = \varnothing \\ \hline
\end{array}
$$

4. Operational schemas describing aspects of the normal operation of the system:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Enrolok}} \\ \hline
\Delta \textit{Class} \\
s? : \textit{Student} \\
r! : \textit{Response} \\ \hline
s? \notin \textit{enrolled} \\
\#\textit{enrolled} < \textit{size} \\
\textit{enrolled}' = \textit{enrolled} \cup \{s?\} \\
\textit{tested}' = \textit{tested} \\
r! = \textit{success} \\ \hline
\end{array}
$$

7.3.1 Z declarations

Here are some sample Z declarations, given with the LaTeX source. We can declare simple types as:

$[\textit{Student}]$

```
\begin{zed}
  [Student]
\end{zed}
```

Free types as:

$\textit{Position} ::= \textit{off} \mid \textit{on}$

```
\begin{syntax}
  Position ::= off | on
\end{syntax}
```

A declaration:

$$x : \mathbb{N}$$

```
\begin{zed}
  x : \nat
\end{zed}
```

Another declaration:

$$\forall n : \mathbb{N} \bullet n + n \in \mathit{even}.$$

```
\begin{zed}
  \all n: \nat \dot n+n \mem even.
\end{zed}
```

An axiomatic definition:

$$\begin{array}{|l}
\mathit{size} : \mathbb{N} \\
\hline
\mathit{size} = 6
\end{array}$$

```
\begin{axdef}
  size : \nat
\where
  size = 6
\end{axdef}
```

Schemas describe the state of the system components, and the operations on them:

$$\begin{array}{|l}
\hline
\textit{Enrolok} \\
\Delta \mathit{Class} \\
s? : \mathit{Student} \\
r! : \mathit{Response} \\
\hline
s? \notin \mathit{enrolled} \\
\# \mathit{enrolled} < \mathit{size} \\
\mathit{enrolled}' = \mathit{enrolled} \cup \{s?\} \\
\mathit{tested}' = \mathit{tested} \\
r! = \mathit{success} \\
\hline
\end{array}$$

```
\begin{schema}{Enrolok}
\Delta Class \\
  s?: Student \\
  r!: Response
\where
  s? \notin enrolled \\
  \# enrolled < size \\
  enrolled' = en-
rolled \cup \{ s? \} \\
  tested' = tested \\
  r! = success
\end{schema}
```

7.3.2 Example specification

Here is a partial specification of a video library system. We begin by introducing the given sets:

$$[\mathit{Video}, \mathit{Borrower}]$$

And we can then specify the maximum number of videos that an individual may borrow, in this case 4:

$$\begin{array}{|l} maxvids : \mathbb{N} \\ \hline maxvids = 4 \end{array}$$

We can then show the abstract state specification of our video library:

$$\begin{array}{|l} \textit{VideoLibrary} \\ \hline stock : \mathbb{F}\, Video \\ instock : \mathbb{F}\, Video \\ issued : Video \pfun Borrower \\ borrowers : \mathbb{F}\, Borrower \\ \hline instock \cup \operatorname{dom} issued = stock \\ instock \cap \operatorname{dom} issued = \varnothing \\ \operatorname{ran} issued \subseteq borrowers \\ \forall b : borrowers \bullet \#(issued \rhd \{b\}) \leq maxvids \\ \hline \end{array}$$

We interpret this specification as a declaration of four sets - the set of stock items, those items in stock, those videos that have been issued and the set of borrowers. The *issued* declaration is a partial function from the videos to the borrowers. We constrain the state space by asserting that:

$instock \cup$ **dom** $issued = stock$ - the union of the *instock* set and the domain of the *issued* function is just the total *stock*.

$instock \cap$ **dom** $issued = \varnothing$ - the intersection of the *instock* set and the domain of the *issued* function is the empty set.

ran $issued \subseteq borrowers$ - the range of the *issued* function is a subset of the *borrowers*.

$\forall\, b : borrowers\ \#\bullet (issued \rhd \{\, b\,\}) \leq maxvids$ - Each *borrower* must have borrowed less than or equal to *maxvids*.

Finally we can start specifying the operational schema. A first one might specify the operation of issuing a video:

$$\begin{array}{|l} \textit{IssueVideo} \\ \hline \Delta VideoLibrary \\ v? : Video \\ b? : Borrower \\ \hline v? \in instock \\ b? \in borrowers \\ \#(issued \rhd \{b?\}) < maxvids \\ issued' = issued \oplus \{v? \mapsto b?\} \\ stock' = stock \\ borrowers' = borrowers \\ \hline \end{array}$$

7.4 Some points

The following points are made about the Z specifications given here:

Implied AND: When we have a specification such as

Class

$enrolled, tested : \mathbb{P}\, Student$

$\#enrolled \leq size$
$tested \subseteq enrolled$

We can also write it as:

Class

$enrolled, tested : \mathbb{P}\, Student$

$(\#enrolled \leq size) \land (tested \subseteq enrolled)$

There is an implied conjunction of the lines in the predicate part of the schema.

Domain and range: The domain and range of a function may be specified by using `\dom` and `\ran` respectively.

Input and output: Input and output variables are identified by `?` and `!` respectively. For example, the declaration
$x? : \mathbb{N}$
marks x as an *input* variable.

Prime character: The prime character "`'`" indicates the new value of a changed item. In our example, we refer to `issued'` - the changed value of the function after the *IssueVideo* operation.

Delta and Xi: The Δ character is used to suggest that the state is changed by the schema in which it is included. The Ξ character is used to suggest that the state is unchanged by the schema in which it is included.
Δ is coded with `\Delta`,
Ξ is coded with `\Xi`.

Size: The number of items in a set S is given by $\#S$.

Empty set: The empty set may be written in three ways:

$x =$
$x = \varnothing$
$x = \{\}$

Range and domain restriction: In our declarations, we may restrict the function: $\mathit{issued} \rhd \{b?\}$ is domain of the function restricted to the range $\{b?\}$ (a set of videos). We also may do the same thing by restricting the domain: $\{v?\} \lhd \mathit{issued}$ is the range of the function restricted to the domain $\{v?\}$ (a set of borrowers). The symbols $\rhd$ and $\lhd$ are coded as `\rres` and `\dres` respectively.

Functional updating: To update a function, we use the $\oplus$ symbol (`\fovr`).

Functions: There are many sorts of functions that can be expressed in Z. Here is a partial list:

Name	**Symbol**	**LaTeX**	**dom** f	*OneToOne*	**ran** f
Total function	→	`\tfun`	$= X$		$\subseteq Y$
Partial function	⇸	`\pfun`	$\subseteq X$		$\subseteq Y$
Total injection	↣	`\tinj`	$= X$	Y	$\subseteq Y$
Partial injection	⤔	`\pinj`	$\subseteq X$	Y	$\subseteq Y$

7.5 Code generation

When implementing systems from specifications, the predicates in the abstract state specification correspond to system invariants in the code. Some programming languages[2] support the concept of system invariants.

The predicates in operational schema may also correspond to the preconditions and postconditions of our code. We can improve our code, and ensure that it never deviates from the specification, by encoding the predicates as checks on our code.

- Those predicates that indicate requirements on the input states lead to *require* checks before executing the code.
- Those predicates that indicate requirements on the output states lead to *ensure* checks after executing the code.

If we had the following schema:

[2]Notably Eiffel.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Testok}} \\
\hline
\Delta Class \\
s? : Student \\
r! : Response \\
\hline
s? \in enrolled \\
s? \notin tested \\
tested' = tested \cup \{s?\} \\
enrolled' = enrolled \\
r! = success \\
\hline
\end{array}
$$

Our code might be something like:

```
function ClassManager.Testok( s : Student ):Response;
begin
   { 'requires' section                    }
   require( s in enrolled );
   require( not (s in tested) );

   { 'implementation' section              }
   ... your code ...

   { 'ensures' section                     }
   ensure( ... an independant check on tested' ... );
   ensure( ... an independant check on enrolled' ... )
   Testok := success
end;
```

The language *Eiffel* supports this sort of check directly in the language. The *pre* and *post* conditions may be specified for each feature, and the class declaration has an *invariant* clause.

```
class TEST feature
   scale is
      require
         ...
      do
         ...
      ensure
         ...
      end;
invariant
   ...
end -- class TEST
```

7.6 Tool support

There are numerous commercial Z tools, including RoZeLink[3], Zola[4] and FuZZ[5]. There are also active research activities at many universities around the world, embedding Z within theorem proving systems such as HOL and Isabelle with a view to in-depth analysis and refinement of Z specifications.

As a result of all this activity, there are freely available tools for a range of platforms.

7.6.1 ZTC

ZTC[6] is a Z type checker available for UNIX, Win95 and NT. It accepts Z specifications in either LaTeX format:

```
  \begin{schema}{Class}
     enrolled, tested: \power Stu-
dent
  \where
     \# enrolled \leq size \\
     tested \subseteq enrolled
  \end{schema}
```

Class

$enrolled, tested : \mathbb{P}\, Student$

$\#enrolled \leq size$

$tested \subseteq enrolled$

or ZSL - an ASCII only format:

```
schema Class
  enrolled, tested : P Student
where
  # enrolled <= size;
  tested subseteq enrolled
end schema
```

```
-------- Class ----------------
|  enrolled, tested : P Student
|---------------------------
|  # enrolled <= size;
|  tested subseteq enrolled
-------------------------------
```

ZTC is also able to translate from each of these formats to the other. It can find errors in Z specifications - the type checker ensures that the specification is consistent in its use of types, and that the syntax of the components are correct.

7.6.2 ZANS

ZANS[7] is a Z animation system available for UNIX, Win95 and NT. Like ZTC, it accepts Z specifications in either

[3]http://www.calgary.shaw.wave.ca/headway/RozeLink.htm.
[4]http://www.ist.co.uk/products/zola.html.
[5]http://www.comlab.ox.ac.uk/oucl/software/fuzz.html.
[6]http://saturn.cs.depaul.edu/~fm/ztc.html.
[7]http://saturn.cs.depaul.edu/~fm/zans.html.

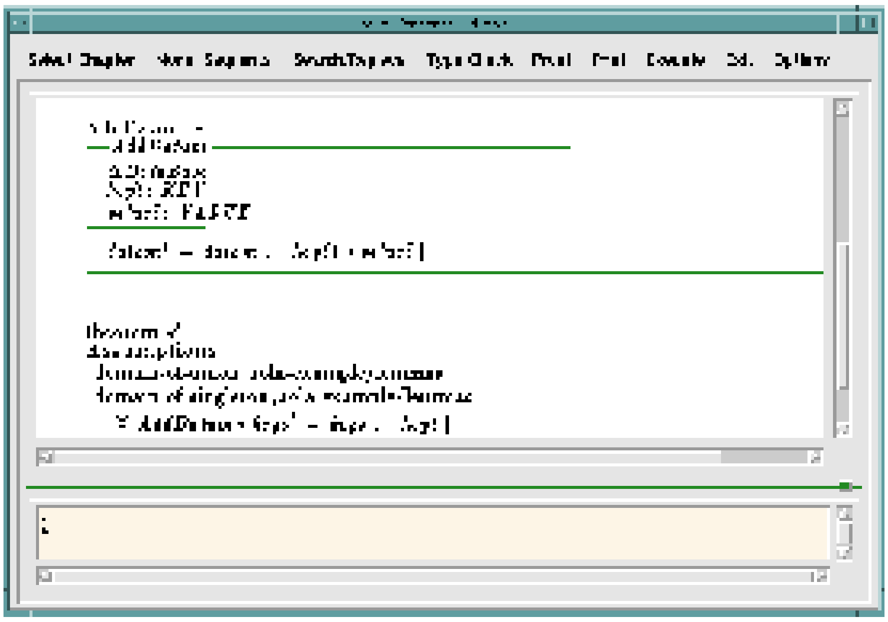


Figure 7.1: The Zola Z tool.

- LaTeX format, or
- ZSL - the ASCII only format.

When a Z specification is input to ZANS, it type checks the specification, and then allows an interactive animation process to be performed.

In the ZANS tutorial, a specification for a “*Class Manager’s assistant*” is animated - adding students to the class, testing them and so on. In the process, an error in the original specification is discovered.

7.7 Reading on Z

The following papers are provided.

1. Peter Lindsay’s “*A Tutorial Introduction to Formal Methods*”.
2. Xiaoping Jia’s “*A Tutorial of ZANS*”.
3. Paul King’s “*Printing Z and Object-Z LaTeX documents*”.
4. The “*Class Manager’s Assistant*” specification.
5. The “*Z FAQ*”.

Chapter 8

Verification

The language *promela* is for:

- constructing *models*, with
- attached *specifications*.

The *spin* tool can *compare* the model with the specification - This is called *verification*.

The language *promela* is 'C' like, with an initialization procedure. The formal basis for *promela* is that it is a guarded command language similar to that presented in the chapters on program proof, with extra statements which either make general assertions or test for reachability.

It can be used to model asynchronous or synchronous, deterministic or non-deterministic systems, and has a special data type called a channel which assists in modelling communication between processes.

8.1 Spin

Spin can be used in three basic modes:

1. Simulator - for rapid prototyping with a random, guided, or interactive simulation.
2. State space analyzer - exhaustively proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search).
3. Bit-state space analyzer - validates large protocol systems with maximal coverage of the state space (a proof approximation technique).

8.2 Promela language

The language is a general purpose one, not restricted to modelling of boolean or digital systems. We would be able to use it to model systems with *analog* inputs for example.

Statements block until they can be executed - so for example the test $a = b$ would block until a was equal to b (It does not return a boolean). The *atomic* construct allows sequences of code to be treated as an atomic unit.

8.2.1 Data types

The language has a small set of data types, with similar meaning to those found in C:

- **int** - an integer,
- **bool** - a boolean,
- **byte** - byte value,
- **short** - a short unsigned integer, and
- **chan** - a FIFO channel.

8.2.2 Channels

The channel data type is declared as follows:

```
chan name = [size] of { .... };
```

We can use variables of this type in various ways:

- **`ch!expr`** - puts expr into a channel
- **`ch!expr,expr,...`** - puts expr,expr,... into a channel
- **`ch!expr(expr)`** - puts expr,expr into a channel
- **`ch?expr`** - waits for something from channel
- **`ch?expr,expr,...`** - waits for something from channel
- **`ch?expr(expr)`** - waits for something from channel

8.2.3 Specifications

We can pepper our code with assertions to test the correctness of our model. Given a specification in $[pre, post]$ form, we can just put one assertion for every component of the precondition, and one assertion for every component of the postcondition.

```
assert(some_boolean_condition);
```

If the asserted condition is not TRUE then the simulation or verification fails, indicating the assertion that was violated.

8.2.4 Temporal claims

We may also make temporal claims - for example a claim such as "*we got here again without making any progress*". The support for temporal claims takes the form of:

- Endstate labels - for determining valid endstates
- Progress labels - claim the absence of non-progress cycles
- Never claims - express impossible temporal assertions

8.2.5 Control structures

We have a fairly normal set of structures for model/simulation control. The selection structure is:

```
if
:: (a != b) -> .....
:: .....
fi
```

Repetition is modelled by:

```
do
:: ... option 1 ...
:: ... option 2 ...
od
```

There is even a bad goto:

```
goto label
```

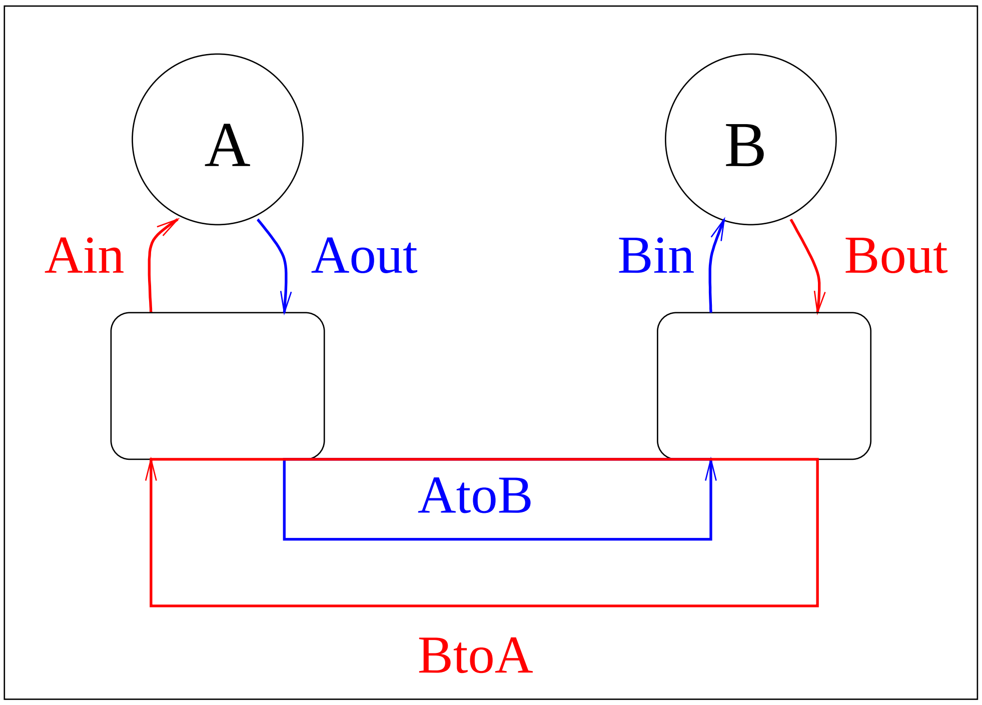
8.2.6 Processes

There are facilities to declare and create processes:

> **`proctype A () { }`** - process A, with parameters as needed
>
> **`init {}`** - the mainline
>
> **`run A ()`** - start up process A with parameters as needed
>
> **`atomic {}`** - enclosed items are atomic

8.3 Simple example

In this example, we model a simple system with two application processes (A and B), communicating with each other using a protocol implemented with two other *transfer* processes. We may visualize this as follows:



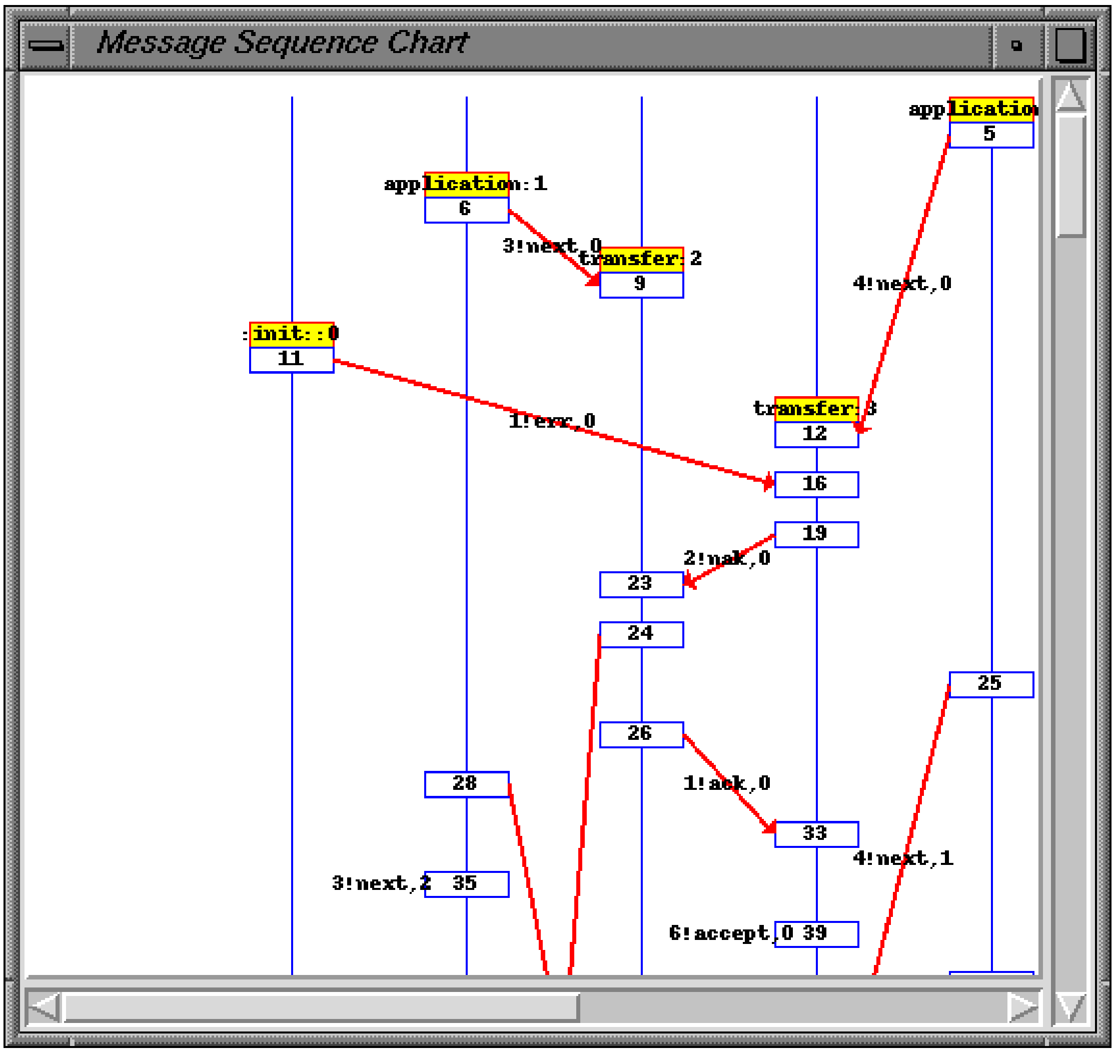
8.3.1 Code

The following code segment specifies the simple protocol, and a simple application to 'exercise' the protocol.

```
#define MAX 10
mtype = { ack, nak, err, next, accept }

proctype transfer( chan in, out, chin, chout )
{
   byte o,i;
   in?next(o);
   do
     :: chin?nak(i) -> out!accept(i); chout!ack(o)
     :: chin?ack(i) -> out!accept(i); in?next(o); chout!ack(o)
     :: chin?err(i) -> chout!nak(o)
   od
}

proctype application( chan in, out )
{
   int i=0, j=0, last_i=0;
   do
     :: in?accept(i) ->
           assert( i==last_i );
           if
             :: (last_i!=MAX) -> last_i = last_i+1
             :: (last_i==MAX)
           fi
     :: out!next(j) ->
           if
             :: (j!=MAX) -> j=j+1
             :: (j==MAX)
           fi
   od
}

init
{
   chan AtoB = [1] of { mtype,byte };
   chan BtoA = [1] of { mtype,byte };
   chan Ain  = [2] of { mtype,byte };
   chan Bin  = [2] of { mtype,byte };
   chan Aout = [2] of { mtype,byte };
   chan Bout = [2] of { mtype,byte };
   atomic {
      run application( Ain,Aout );
      run transfer( Aout,Ain,BtoA,AtoB );
      run transfer( Bout,Bin,AtoB,BtoA );
      run application( Bin,Bout )
   };
   AtoB!err(0)
}
```

8.3.2 Simulation

The spin tool may then be used to either exhaustively check the model, or to simulate and display the results:


Message Sequence Chart
application:5
5
application:1
6
3!next,0
transfer:2
9
4!next,0
:init::0
11
transfer:3
12
1!err,0
16
19
2!nak,0
23
24
25
26
28
1!ack,0
33
4!next,1
3!next,2
35
6!accept,0
39

Chapter 9

Abstract data types

There are four types of specification:

- Algebraic for sequential systems - (Larch)
- Algebraic for concurrent systems - (LOTOS)
- Model-based for sequential systems - (Z)
- Model-based for concurrent systems - (CSP)

The *model-based* systems may use algebraic notation as in Z, but the underlying view is that a model of a system is being constructed. In an *algebraic* specification, the sub-systems are defined in terms of their interfaces to other sub-systems.

Common to all the methods is some terminology for defining an *abstract data type* (ADT).

For each ADT we define:

- the **sort** - the type of the contained objects
- the **interface syntax**
- the **axioms** - the defined operations on the ADT

9.1 Example ADT

In an earlier chapter, we met a specification of a *stack* given using the axioms for the *stack* operation.

Here is a more formal specification of a *stack* using an ADT terminology:

Type definition	**sort**: stack
Description	Stacks are data structures that can be created, and grow or diminish in size when items are placed on, or removed from them.
	A Stack has a minimum of 0 items on it.
	The last item pushed is the first item pulled (LIFO).
	After a pull the stack reverts to its state as it was before the last push.
Interface	new -> S
	push(item , S_1) -> S_2
	pop(S_2) -> S_1
	top(S_2) -> item
	empty(S) -> boolean
Axioms	top(push(item , S)) = item
	pop(push(item ,S)) = S
	empty(new) = TRUE
	empty(push(item , S)) = FALSE

9.2 LOTOS

LOTOS (Language of Temporal Ordering Specification) is a specification language which is an ISO standard, and is intended for use in specification of protocols. It has:

- process and type definitions
- Actions and events specified in *behaviour* expressions
  - ; $\Rightarrow$ action prefix (behaviour expression prefixed by an action)
  - [] $\Rightarrow$ choice
  - loop $\Rightarrow$ recursion
  - exit $\Rightarrow$ termination of process
  - stop $\Rightarrow$ inactivity
  - >> $\Rightarrow$ sequential composition
  - B1|[a,b...]|B2 $\Rightarrow$ parallel composition

LOTOS specifications may be animated, and there is tool support for direct conversion of LOTOS specifications to EFSMs and hence to code. LOTOS syntax and structure bears some relationship to Ada, and one of the LOTOS tools produces Ada code directly.

9.2.1 LOTOS example

We use the example given in Bustard, Norris and Orr [8], the ’pass-the-parcel’ game:

> Parcel, children, layers of paper, music, unwrapping, tea, circulation direction and so on...

Natural language descriptions may have many loopholes, and there are various ways we could formalize the description:

- State variables:
  - music on or off,
  - number of layers of paper,
  - child identity (and so on).
- Abstract data types
  - Functions given meaning through equations as we saw before.

LOTOS can support either method. It has:

- An abstract data type language, and
- A process algebra for describing system behaviour

We can pass the parcel like this:

```
process Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved, PresentFound]: exit :=
   ( ParcelPassed;
        Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved, PresentFound])
   []
   ( MusicStops;
        LayerRemoved;
           (( MusicStarts;
              ParcelPassed;
              Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved, PresentFound])
           [] ( PresentFound;
           exit)))
endproc (* Circulate *)
```

9.2.2 Alternating bit protocol

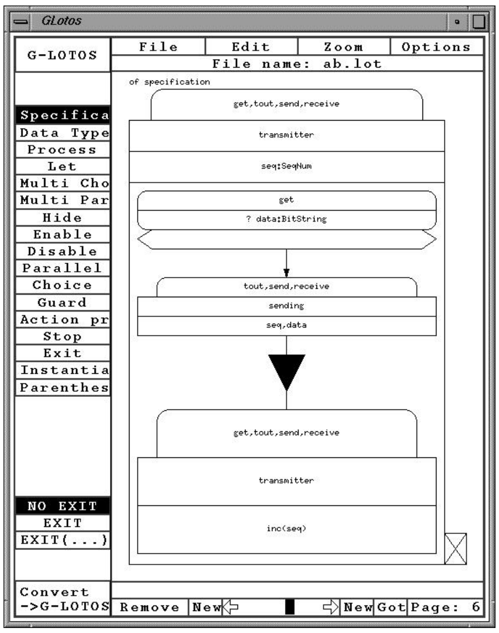
The LOTOS specification for the AB - alternating bit - protocol.

```
SPECIFICATION DataLink [get, give] : NOEXIT
TYPE Boolean IS
  SORTS
    bool
  OPNS
    true, false: -> bool
    not: bool -> bool
    _ and _, _ or _, _ xor _, _ im-
plies _, _ iff _: bool, bool -> bool
    _ equal _, _ ne _: bool, bool -> bool
  EQNS
    FORALL x, y: bool
    OFSORT bool
      not (true) = false ;
      not (not (x)) = x ;
      x and true = x ;
      x and false = false ;
      x or true = true ;
      x or false = x ;
      x xor y = (x and not (y)) or (y and not (x)) ;
      x implies y = y or not (x) ;
      x iff y = (x implies y) and (y im-
plies x) ;
      x equal y = x iff y ;
      x ne y = x xor y ;
  ENDTYPE

TYPE sequenceNumber IS
  Boolean
  SORTS
    SeqNum
  OPNS
    0: -> SeqNum
    inc: SeqNum -> SeqNum
    _ equal _: SeqNum, SeqNum -> bool
  EQNS
    OFSORT SeqNum
    FORALL x: SeqNum
      inc (inc (x)) = x ;

    OFSORT bool
    FORALL x, y: SeqNum
      x equal x = true ;
      x equal inc (x) = false ;
      inc (x) equal x = false ;
      inc (x) equal inc (y) = (x equal y) ;
  ENDTYPE

TYPE BitString IS
  sequenceNumber
  SORTS
    BitString
  OPNS
    empty: -> BitString
    add: SeqNum, BitString -> BitString
  ENDTYPE

TYPE FrameType IS
  Boolean
  SORTS
    FrameType
  OPNS
    info, ack: -> FrameType
    equal: FrameType, FrameType -> bool
  EQNS
    OFSORT bool
    FORALL x: FrameType
      equal (info, ack) = false ;
      equal (ack, info) = false ;
      equal (x, x) = true ;
  ENDTYPE
BEHAVIOUR
  HIDE tout, send, receive IN
    (   (   transmitter [get, tout, send, re-
ceive] (0)
          |||
            receiver [give, send, receive] (0)
        )
      |[tout, send, receive]|
        line [tout, send, receive]
    )
  WHERE
    PROCESS transmitter
            [get, tout, send, re-
ceive] (seq: SeqNum) : NOEXIT :=
        get ?data: BitString;
        sending [tout, send, re-
ceive] (seq, data)
      >>
        transmitter [get, tout, send, re-
ceive] (inc (seq))
      WHERE
        PROCESS sending [tout, send, receive]
                        (seq: Se-
qNum, data: BitString) : EXIT :=
          send !info !seq !data;
          (   receive !ack !inc (seq) !empty;
              EXIT
            []
              tout;
              sending [tout, send, re-
ceive] (seq, data)
          )
          ENDPROC (* sending *)
      ENDPROC (* transmitter *)

    PROCESS receiver [give, send, re-
ceive] (exp: SeqNum) : NOEXIT :=
      receive !info ?rec: SeqNum ?data: Bit-
String;
      (   [rec equal exp = true]->
            give !data;
            send !ack !inc (rec) !empty;
            receiver [give, send, re-
ceive] (inc (exp))
        []
          [inc (rec) equal exp = true]->
            send !ack !inc (rec) !empty;
            receiver [give, send, re-
ceive] (exp)
      )
      ENDPROC (* receiver *)

    PROCESS line [tout, send, re-
ceive] : NOEXIT :=
      send ?f: FrameType ?seq: Se-
qNum ?data: BitString;
      (   receive !f !seq !data;
          line [tout, send, receive]
        []
          I;
          tout;
          line [tout, send, receive]
      )
      ENDPROC (* line *)
ENDSPEC
```

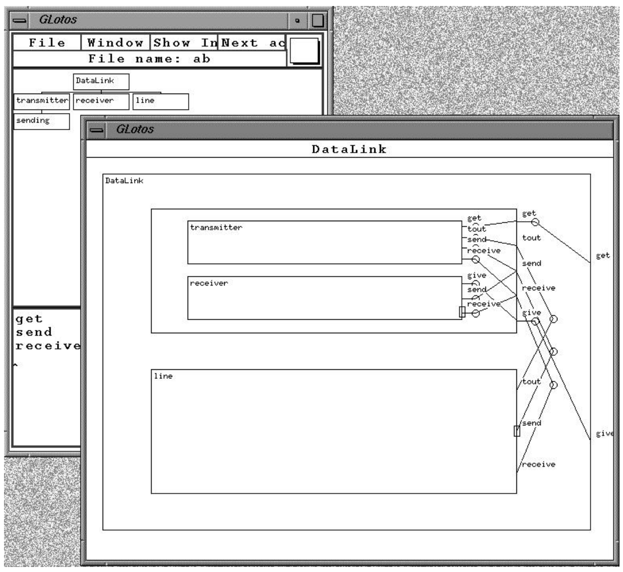
9.2.3 Tool support

The GLD tool can be used to view/edit/create a G-LOTOS (Graphical LOTOS) version of the specification. The G-LOTOS graphical specification can be directly converted to a textual version of the specification when needed.

The tool looks like this:



The GLA tool can be used to animate a LOTOS specification:



Bibliography

[1] Nancy Leveson, *“Safeware: System Safety and Computers,”* Addison-Wesley, 1995.

[2] Halls myths

[3] Bowens seven more

[4] Robinson JACM 12:23-41,1965

[5] BNF

[6] Backus “Can Programming be liberated...”

[7] Morgan - Prob proof article with Monty Hall

[8] Bustard, Norris and Orr, “An Exercise in Formalizing the Description of a Concurrent System,” S,P&E, Vol 22(12) 1069-1098, Dec 1992.