Software Blueprints Web Pages

These supply supplementary material for our book: *Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling*. We split the material into two media so that you can easily browse through the main text (a book is good for that purpose) but also obtain supplementary material which is necessary for completeness but makes a book slow to read (our Web pages deliver this).

The pages referenced from our book are listed below, with the numerical label matching the one in the book. The first of the two numbers on the label refers to the book chapter. Note that little of the material in the list below will make sense without first reading the appropriate chapter.

- 1.1 A quick introduction to Prolog programming.
- 2.1 Refinement of an operator model.
- 2.2 Formalisation of a decision procedure model.
- 2.3 Building an interpreter for the operator model.
- 2.4 Building an interpreter to test for inconsistency.
- 4.1 A detailed process model.
- 4.2 Further details of a basic scheduling model.
- 5.1 A more complex example of WWW site generation.
- 5.2 A more complex example of ecological model generation.
- 6.1 A sample of typical definitional skeletons.
- 6.2 A larger example of techniques editing.
- 6.3 An example of stripping structure from a predicate.
- 7.1 A more general view of endorsements.
- 7.2 More about justification of decisions.
- 10.1 Relating domain languages to logic.
- 10.2 Patterns of problem solving.

Quick Prolog

Our book isn't specific to Prolog but it chooses a style of formal description which is close to Prolog, so that you can start to implement some of the ideas directly in a commonly available programming language. The purpose of this page is to help make the book self-contained by giving you a brief guide to basic Prolog programming.

1 Introduction

There is little point, these days, in writing a large and introduction to Prolog programming. Most technical bookshops will sell a range of introductory Prolog texts, most of which repeat (in various different styles) the same basic material. So the first thing to do if you are serious about learning Prolog is stop reading these notes; go down to your nearest technical bookshop and find an introductory text which suits you. The next thing to do is to get access to one of the standard dialects of Prolog. There are umpteen different implementations of Prolog systems and each has its own idiosyncracies. Fortunately, there is a core set of notation which tends to be provided by most Prolog systems and there is now an ISO standard for the language. We try to stick to this core in order to avoid portability problems, but give no guarantees of portability.

2 Getting Started

This section is designed to give an initial, superficial introduction to basic Prolog programming. The idea is to get you working on the computer from the very first.

2.1 Some Anatomy

Prolog programs consist of **clauses**. For example, the following is a valid Prolog program containing 3 clauses (1 per line in this example):

```
parent(fred, greta).
parent(greta, henry).
grandparent(X, Z):- parent(X, Y), parent(Y, Z).
```

All Prolog clauses are **terms** finishing with a full stop symbol.
Prolog clauses are normally of three types:

- **facts** declare things that are always true.
- **rules** declare things that are true depending on a given condition.
- **questions** to find out if a particular goal is true.

Anatomy of a fact:

- `parent(fred, greta).` is a fact.
- `parent` is a **predicate name**.
- `fred` is the first **argument**.
- `greta` is the second argument.
- Arguments in a term are separated by commas.
- `parent`, `fred` and `greta` are **atoms**.

Anatomy of a rule:

- `grandparent(X, Z) :- parent(X, Y), parent(Y, Z).` is a rule.
- Its **head** is `grandparent(X, Z)`.
- Its **body** is `parent(X, Y), parent(Y, Z)`.
- `parent(X, Y)` is the first **subgoal** of the rule.
- `parent(Y, Z)` is the second **subgoal** of the rule.
- Subgoals in the body of a rule are separated by commas.
- `X`, `Y` and `Z` are **variables**.
- Its **declarative interpretation** can be stated in two different ways:
  - For all `X` and `Z`, `X` is a `grandparent` of `Z` if there exists some `Y` such that `X` is a `parent` of `Y` and `Y` is a `parent` of `Z`.
  - For all `X`, `Y` and `Z`, if `X` is a `parent` of `Y` and `Y` is a `parent` of `Z` then `X` is a `grandparent` of `Z`.
- Its **procedural interpretation** can be stated as:
  - The goal `grandparent(X, Z)` succeeds with binding `X1` for `X` and and binding `Z1` for `Z` if first, the goal `parent(X, Y)` succeeds with bindings `X1` and `Y1` and then the goal `parent(Y, Z)` succeeds with bindings `Y1` and `Z1`.

Running a Prolog program consists of supplying some **question**. This also is referred to as giving an **initial goal** or **query**. A goal **succeeds** if it is **satisfiable** from the set of clauses constituting ther Prolog program. A goal **fails** if it is **unsatisfiable** from the set of clauses constituting the Prolog program. For example:

- The query:

  ```
  ?- grandparent(fred, X).
  ```

is satisfiable from the program at the top of the page, with `X` **instantiated** to `henry`.

- On the other hand, the query:

  ```
  ?- grandparent(fred, bob).
  ```

  is not satisfiable from the program at the top of the page because `bob` doesn't appear in that set of clauses.

Prolog programs can be **recursive**. That is, a rule with some predicate as its head can contain a reference to the same predicate in its body. Consider the following program:

```
/* Clause1 */ ancestor(A, B):- parent(A, B).
/* Clause2 */ ancestor(A, B):- parent(P, B), ancestor(A, P).
/* Clause3 */ parent(fred, greta).
/* Clause4 */ parent(greta, henry).
```

Here the second clause of the predicate ancestor is recursive because it has `ancestor(A, B)` as its head and `ancestor(A, P)` as the second subgoal in its body.

The Prolog interpreter can **backtrack** to find alternative solutions to a given goal. Backtracking can be forced by typing a semi-colon after Prolog finds the first solution to the query. For example, we get the following behaviour when running the program given above:

What you see on the screen (annotations between `**` symbols) is:

```
| ?- ancestor(X, henry).  ** I give a query to Prolog **

     X=greta ;  ** It finds a solution and I type a semi-colon **

     X=fred    ** Another solution is found for the original query **
yes   ** Since I didn't type another semi-colon Prolog reports success **
| ?-  ** and prompts for another query **
```

An explanation of how Prolog obtains this observed behaviour is given by the following annotated sequence of events. Goal1 is the initial query.

```
Goal1:       ancestor(X, henry).
        ** Match head of Clause1 to Goal1 **
Clause1(a): ancestor(X, henry) if (Goal2) parent(X, henry).
        ** Attempt to satisfy Goal2 **
Goal2:       parent(X, henry).
        ** Match to Clause4 to Goal2 **
Clause4(a): parent(greta, henry).
        ** SUCCEED with X = greta **
        ** Asked to find another solution **
```

```
        ** so backtrack to look for alternative solution for Goal2 **
        ** No further matches for Goal2 **
        ** so backtrack to look for alternative solution for Goal1 **
        ** Match head of Clause2 to Goal1 **
Clause2(a): ancestor(X, henry) if (Goal3) parent(P, henry)
                              and (Goal4) ancestor(X, P).
        ** Attempt to satisfy Goal3 **
Goal3:      parent(P, henry).
        ** Match Clause4 to Goal3 **
Clause4(b): parent(greta, henry).
        ** so P is now bound to 'greta' **
        ** Attempt to satisfy Goal4 **
Goal4:      ancestor(X, greta).
        ** Match Clause 1 to Goal4 **
Clause1(b): ancestor(X, greta) if (Goal5) parent(X, greta).
        ** Attempt to satisfy Goal2 **
Goal5:      parent(X, greta).
        ** Match to Clause3 to Goal2 **
Clause3(a): parent(fred, greta).
        ** SUCCEED with X = fred **
```

3 Standard Programming Style

You should always document you Prolog programs in order that other people (or you, at some later date) can understand what they are meant to do. There aren't rigid guidelines for this but there is a reasonably well established set of conventions, which are as follows:

- First, put a skeletal description of your predicate, showing the predicate name and its arguments (in order), with the names of the arguments giving an idea of the type of structure expected and a **mode declaration** at the start of each argument. Mode declarations are either `+` (if the argument is instantiated whenever the predicate is called), `-` (if the argument is uninstantiated when the predicate is called) or `?` (if the argument can be either instantiated or uninstantiated when called).
- Below this put a description in English of the behaviour of the predicate. This often makes use of the argument names from the skeletal description.

Consider, for example, the following documentation for `memberchk/2`: checks if

```
%   memberchk(+Element, +Set)
%   means the same thing, but may only be used to test whether a known
%   Element occurs in a known Set.  In return for this limited use, it
%   is more efficient when it is applicable.
```

```
memberchk(Element, [Element|_]) :- !.
memberchk(Element, [_|Rest]) :-
        memberchk(Element, Rest).
```

Note the use of `%` symbols at the start of lines which are comments. When loading a program, Prolog ignores everything from this symbol up to the end of the same line. Larger comments can be enclosed between `\*` and `*\` symbols. In the `memberchk/2` example, `Element` and `Set` are both allocated mode `+` because both need to be instantiated for the predicate to work as intended. Compare this to the definition of `member/2` (below), where either of the two elements may be either instantiated or uninstantiated, as required, and are therefore given `?` as their mode declarations.

```
%   member(?Element, ?Set)
%   is true when Set is a list, and Element occurs in it.  It may be used
%   to test for an element or to enumerate all the elements by backtracking.
%   Indeed, it may be used to generate the Set!

member(Element, [Element|_]).
member(Element, [_|Rest]) :-
        member(Element, Rest).
```

One final point about these examples. Notice that an underscore to represent an **anonymous variable** is used wherever the name of a variable is unimportant. How do you know that it is unimportant? If you see a clause in which a named variable appears only once then its name doesn't matter since it isn't matching to anything in the clause. For instance the first clause of `member/2` could have been:

```
member(Element, [Element|Rest]).
```

but it isn't important to distinguish `Rest` with a special name, so the convention is to use an underscore. Many Prolog systems will generate warning messages when they find "singleton" variables, such as `Rest` in the example above. It is a good idea to get into the habit of using underscores to avoid these warning messages. That way, if you accidentally introduce a singleton variable (*e.g.* by mistyping the name of a variable) you will always be able to spot it when you load the file and see the singleton warning message.

4 Prolog Terms

All Prolog data structures are called **terms**. A term is either:

- A **constant**, which can be either an **atom** or a **number**.
- A **variable**.
- A **compound term**.

4.1 Atoms

An atom is:

- Anything enclosed in single quotes (e.g. `'Whatever you like'`).
- Any sequence of characters, numbers and/or the underscore character which is preceded by a lower case character (e.g. `this_IS_an_atom`).
- Any continuous sequence of the symbols: `+`, `-`, `*`, `/`, `\`, `^`, `>`, `<`, `=`, `'`, `:`, `.`, `?`, `@`, `#`, `$`, `&`. (e.g. `*******+@`).
- Any of the special atoms: `[]`, `{}`, `;`, `!`.

The Prolog built-in predicate `atom/1` will tell you if a given term is or isn't an atom. It gives the following behaviour:

```
| ?- atom(foo).
yes
| ?- atom(Foo).
no
| ?- atom('Foo').
yes
| ?- atom([]).
yes
```

4.2 Numbers

A number can be either:

- An integer (e.g. `99`).
- A floating point number (e.g. `99.91`).

The built-in predicate `integer/1` succeeds if its argument is an integer. The built-in predicate `real/1` succeeds if its argument is a real number. These give the following behaviour:

```
| ?- integer(9).
yes
| ?- integer(99.9).
no
| ?- real(99.9).
yes
| ?- real(9).
no
| ?- number(9).
yes
| ?- number(99.9).
yes
```

```
| ?- number(foo).
no
| ?- number(Foo).
```

4.3 Variables

A variable is:

- Any sequence of characters, numbers and/or the underscore character which is preceded by an upper case character (e.g. `This_IS_a_variable`).
- Any sequence of characters, numbers and/or the underscore character which is preceded by an underscore (e.g. `_this_IS_a_variable`).
- An underscore character by itself (e.g. `_`). The underscore character is referred to as the **anonymous variable** because it cannot be referenced by its name in other parts of the clause.

The Prolog built-in predicate `var/1` will tell you if a given term is or isn't a variable. It gives the following behaviour:

```
| ?- var(Foo).
     Foo=_125459
yes
| ?- var(_).
yes
| ?- var(foo).
no
```

4.4 Compound Terms

Compound terms allow the representation of data with substructure. A compound term consists of a **functor** followed by a sequence of one or more subterms called **arguments**. It sometimes helps to think of a compound term as a tree structure. For example, the term `sentence(nphrase(john),vbphrase(verb(likes),nphrase(mary)))` could be depicted as the structure:

```
       ______sentence______
      |                    |
   nphrase         _____vbphrase_____
      |           |                  |
    john         verb             nphrase
                  |                  |
                 likes              mary
```

where sentence is the **principal functor** of the term and its arguments are `nphrase(john)` and `vbphrase(verb(likes),nphrase(mary))`. Each argument in this example is also a term with functor name and arguments.

When we want to refer to a particular class of term without giving its precise definition, we give its **principal functor** and its **arity** (the number of arguments which it takes). The example above has principal functor sentence and arity 2. The shorthand notation for this is `sentence/2`.

The predicate `functor(Term, F, A)` succeeds if `Term` has functor `F` and arity `A`. Its behaviour is as follows:

```
| ?- functor(sentence(nphrase(john),vbphrase(verb(likes),nphrase(mary))),
|:            Functor, Arity).
     Functor=sentence
     Arity=2
yes
| ?- functor(foo, Functor, Arity).
     Functor=foo
     Arity=0
yes
```

4.5 Lists

Lists are a special class of compound term. They are represented in Prolog as either:

- The atom `[]`, representing the empty list.
- A compound term with functor `'.'` and two arguments representing respectively the head and tail of the list. The tail argument must, itself be a list (possibly empty).

For example the list containing the sequence of elements `a`, `b` and `c` is represented by the term `.(a,.(b,.(c,[])))`. Since this is hard to read, you can use an alternative notation. Simply write the list as a sequence of elements separated by commas and enclosed in square brackets. The previous example could we rewritten using this notation as `[a,b,c]`.

You can confirm this by giving the following goals to Prolog:

```
| ?- functor(.(a,.(b,.(c,[]))), Functor, Arity).
     Functor=(.)
     Arity=2
yes
| ?- functor([a,b,c], Functor, Arity).
     Functor=(.)
     Arity=2
yes
| ?- .(a,.(b,.(c,[]))) = [a,b,c].
yes
```

4.6 Operators

Operators in Prolog are simply a notational convenience. We could represent the English statement "2 plus 3" in Prolog as `+(2, 3)` but this isn't the way we are accustomed to seeing the expression. A more familiar notation involves making the `+` an INFIX operator so that we can write `2 + 3`. When an operator is defined it is also allocated two descriptive attributes:

- Its **precedence**. This is a number between 1 and 1200 which allows the Prolog system to decide the principal functor of any nested term. For example, the precedence of `+` is 500 while the precedence of `*` (the multiplication operator) is 400. The rule is that in a nested term the operator with highest precedence value is the principal functor. For instance, the term `2 + 3 * 5` has principal functor `+`. If we were to write it in standard Prolog syntax this term would look like this: `+(2,*(3,5))`. NOTE that terms representing numerical expressions are NOT automatically evaluated, like in a functional language. There is a special built–in predicate for doing this, which we shall meet later on.

- Its **type**. This defines whether the operator is:
  - **Prefix**, in which case it is allocated type `fx` or `fy`.
  - **Postfix**, in which case it is allocated type `xf` or `yf`.
  - **Infix**, in which case it is allocated type `xfx`, `xfy` or `yfx`.

  Some operators may have more than one type. For example the `+` operator is both prefix (e.g. `+ 2`) and infix (e.g. `2 + 3`).

You can check what the current operators in your Prolog system are by using the `current_op/3` built-in predicate. You can also define your own operators using the built–in `op/3` predicate but we won't worry about this yet. To display a term containing operators in standard Prolog syntax, use the built-in predicate `display/1`.

```
| ?- current_op(Precedence, Type, +).

     Precedence=500
     Type=fx ;

     Precedence=500
     Type=yfx
yes
| ?- current_op(Precedence, Type, *).

     Precedence=400
     Type=yfx
yes
| ?- display(2 + 3 + 4).
```

```
+(+(2,3),4)
yes
```

5 Unification

The way in which Prolog matches two terms is called **unification**. The idea is similar to that of unification in logic: we have two terms and we want to see if they can be made to represent the same structure. For example, we might have in our database the single Polog clause:

```
parent(alan, clive).
```

and give the query:

```
|?- parent(X,Y).
```

We would expect `X` to be **instantiated** to `alan` and `Y` to be instantiated to `clive` when the query succeeds. We would say that the term `parent(X,Y)` **unifies** with the term `parent(alan, clive)` with `X` bound to `alan` and `Y` bound to `clive`. The unification algorithm in Prolog is roughly this:

Definition 1 *Given two terms T_1 and T_2 which are to be unified:*

- *If T_1 and T_2 are constants (i.e. atoms or numbers) then if they are the same succeed. Otherwise fail.*
- *If T_1 is a variable then instantiate T_1 to T_2.*
- *Otherwise, If T_2 is a variable then instantiate T_2 to T_1.*
- *Otherwise, if T_1 and T_2 are complex terms with the same arity (number of arguments), find the principal functor F_1 of T_1 and principal functor F_2 of T_2. If these are the same, then take the ordered set of arguments of $\langle A_1, \cdots, A_N \rangle$ of T_1 and the ordered set of arguments $\langle B_1, ..., B_N \rangle$ of T_2. For each pair of arguments A_M and B_M from the same position in the term, A_M must unify with B_M.*
- *Otherwise fail.*

For example: applying this procedure to unify `foo(a,X)` with `foo(Y,b)` we get:

- `foo(a,X)` and `foo(Y,b)` are complex terms with the same arity (2).
- The principal functor of both terms is `foo`.
- The arguments (in order) of `foo(a,X)` are `a` and `X`.
- The arguments (in order) of `foo(Y,b)` are `Y` and `b`.
- So `a` and `Y` must unify , and `X` and `b` must unify.

– Y is a variable so we instantiate Y to a.

– X is a variable so we instantiate X to b.

- The resulting term, after unification is foo(a,b).

The built in Prolog operator '=' can be used to unify two terms. Below are some examples of its use. Annotations are between ** symbols.

```
| ?- a = a.        ** Two identical atoms unify **
yes
| ?- a = b.        ** Atoms don't unify if they aren't identical **
no
| ?- X = a.        ** Unification instantiates a variable to an atom **
     X=a
yes
| ?- X = Y.        ** Unification binds two differently named variables **
     X=_125451     ** to a single, unique variable name **
     Y=_125451
yes
| ?- foo(a,b) = foo(a,b).       ** Two identical complex terms unify **
yes
| ?- foo(a,b) = foo(X,Y).       ** Two complex terms unify if they are **
     X=a                        ** of the same arity, have the same principal**
     Y=b                        ** functor and their arguments unify **
yes
| ?- foo(a,Y) = foo(X,b).       ** Instantiation of variables may occur **
     Y=b                        ** in either of the terms to be unified **
     X=a
yes
| ?- foo(a,b) = foo(X,X).       ** In this case there is no unification **
no                              ** because foo(X,X)  must have the same **
                                ** 1st and 2nd arguments **
| ?- 2*3+4 = X+Y.       ** The term 2*3+4 has principal functor + **
     X=2*3              ** and therefore unifies with X+Y with X instantiated**
     Y=4                ** to 2*3 and Y instantiated to 4 **
yes
| ?- [a,b,c] = [X,Y,Z]. ** Lists unify just like other terms **
     X=a
     Y=b
     Z=c
yes
| ?- [a,b,c] = [X|Y].   ** Unification using the '|' symbol  can be used **
     X=a                ** to find the head element, X, and tail list, Y, **
     Y=[b,c]            ** of a list **
yes
| ?- [a,b,c] = [X,Y|Z]. ** Unification on lists doesn't have to be **
     X=a                ** restricted to finding the first head element **
```

```
     Y=b                ** In this case we find the 1st and 2nd elements **
     Z=[c]              ** (X and Y) and then the tail list (Z) **
yes
| ?- [a,b,c] = [X,Y,Z|T].       ** This is a similar example but here **
     X=a                        ** the first 3 elements are unified with **
     Y=b                        ** variables X, Y and Z, leaving the **
     Z=c                        ** tail, T, as an empty list [] **
     T=[]
yes
| ?- [a,b,c] = [a|[b|[c|[]]]].  ** Prolog is quite happy to unify these **
yes                             ** because they are just notational **
                                ** variants of the same Prolog term **
```

6 Expressing disjunctive subgoals

You have already seen how to represent alternative ways of satisfying a particular goal by giving more than one clause which could match that goal. For example, we could write a program to decide whether or not any number was between two other numbers using the following two clauses:

```
between(N, N1, N2):-
    N > N1,
    N < N2.
between(N, N1, N2):-
    N < N1,
    N > N2.
```

This program states that `N` is between two numbers `N1` and `N2` if **either** `N` is greater than `N1` and `N` is less than `N2` **or** `N` is less than `N1` or `N` is greater than `N2`. This could be stated more succinctly using the special disjunction operator, `‘;’`, which is often interpreted in English as ”or”.

```
between(N, N1, N2):-
    (N > N1,
     N < N2) ;
    (N < N1,
     N > N2).
```

The brackets round the conjunction of inequalities on each side of the disjunction operator aren't really necessary. We have included them to emphasise the nested structure of the term. This new definition of `between/3` using the `‘;’` operator is identical in meaning to the original definition using two separate clauses.

7 Evaluating numerical expressions

To Prolog a numerical expression is just a structure which can be manipulated just like any other term. So the expression `2 + 3 * 4` can be "unpacked" like any other term. For example, the goal:

```
|?- 2+3*4 = X+Y.
```

Unpacks `2+3*4` into subterms `X` (instantiated to `2`) and `Y` (instantiated to `3*4`). If you want to **evaluate** a numerical expression, such as the one given above, then you must use the built in operator: `is`. This is an infix operator which takes on its right side a numerical expression to be evaluated and the result of evaluating this expression is the number on its left side. For example, to find the result of evaluating the expression `2+3*4` we give the goal:

```
|?- X is 2+3*4.
    X=14
yes
```

We don't have to put a variable on the left of the `is` operator. We could use it to test that a particular number is the result of evaluating a given numerical expression:

```
|?- 14 is 2+3*4.
yes
```

Only certain special operators are allowed in evaluable expressions (i.e. terms which appear on the right side of `is` goals). These include the normal arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). Some dialects of Prolog also allow operators such as `sin/1`, `cos/1`, *etc.*

Just to emphasise that numerical expressions can be picked up from the database and passed around like any other expression, only being evaluated when you want it to, consider the following program: We represent data expressions using the following two facts:

```
data(2).
data(3*4).
```

We then give a rule which will calculate the sum of any pair of data expressions contained in the database:

```
calc(N):-
    data(N1),
    data(N2),
    N is N1 + N2.
```

We can then find the sums of all combinations of data expressions by giving the goal `calc(N)` and forcing backtracking for alternative solutions using the `';'` operator:

```
| ?- calc(N).
     N=4 ;
     N=14 ;
     N=14 ;
     N=24
yes
```

8 Negation

You may want to make the success of a particular goal contingent on the failure of some subgoal. This can be done using the built in `\+` operator. For example, in code Section 13.1 file there is a predicate `possible_to_marry(A,B)` which states that "It is possible for A to marry B if A is male and B is female and A is not related to B". The code for this is:

```
    possible_to_marry(A, B):-
        male(A),
        female(B),
        \+ related(A, B).
```

Note that `\+` is a prefix operator which, in this case, takes `related(A,B)` as its argument. An important point to note about negation in Prolog is that it doesn't mean that the goal you have negated is proved false in the logical sense. It does mean that the goal can not be established from the current set of axioms in the database. The implicit assumption here is that the set of axioms in the current database are a complete description of every aspect of the problem. This is often referred to as the **closed world assumption**. This rather liberal interpretation of negation can be used to express defaults. For example we might have a problem in which we could identify all females by name, using the predicate `female/1` to denote that someone is female:

```
female(alice).
female(betty).
    ...etc.
```

and, given that we have recorded the names of all females, we might want to assume that anyone who wasn't a known female was male. We could do this using the rule:

```
male(X):-
    \+ female(X).
```

Now the goal `|?- male(alice).` will fail because `female(alice)` is true and so `\+ female(alice)` must fail. Conversely, the goal `|?- male(fred).` will succeed because there is no way to establish `female(fred)` and so `\+female(fred)` succeeds. However, you have to be careful when exploiting the closed world assumption in this way because someone may subsequently run your program without being aware of this limitation. For example, I happen to know that 'minnie mouse' is female but the program in our example will happily allow the goal `|?- male('minnie mouse').` to succeed.

9 Cuts

Prolog is good at backtracking but uncontrolled backtracking can be a nuisance. The cut symbol (written as `!`) is a special subgoal used to provide a limited form of control over backtracking, by committing the Prolog interpreter to a particular portion of the potential search space. An example will help start to explain how this works. The code for this example can be found in Section 13.2 and further discussion appears in Sterling & Shapiro "The Arto fo Prolog".

We start with the following definition of `minimum/3`:

```
% minimum(+Number1, +Number2, ?Minimum).
% Succeeds if Minimum is the minimum value of Number1 and Number2.

minimum(X, Y, X):-
    X =< Y.
minimum(X, Y, Y):-
    X > Y.
```

The potential search tree for the goal `minimum(1,2,X)` is:

```
                minimum(1, 2, X)
                /             \
               /               \
        minimum(1, 2, 1)    minimum(1, 2, 2)
            /                     \
           /                       \
          /                         \
        1 =< 2                      1 > 2
```

In other words, the Prolog interpreter, in trying to satisfy `minimum(1,2,X)` would first test whether `1 =< 2`, which succeeds giving the result `minimum(1,2,1)`. However, if forced to backtrack it would also try the second clause and try to establish `1 > 2`, which will fail. It would be useful to have some way of telling the Prolog interpreter, once it had established `1 =< 2` that there is no point in looking for any other ways of finding a solution. This information can be given by putting a cut after the first subgoal of the first clause, as shown in `gc_minimum/3`.

```
%  gc_minimum(+Number1, +Number2, ?Minimum).
%Identical to minimum/3 but with a cut in the first clause.

gc_minimum(X, Y, X):-
```

```
    X =< Y, !.
gc_minimum(X, Y, Y):-
    X > Y.
```

The potential search tree for the goal `gc_minimum(1,2,X)` is:

```
                 gc_minimum(1, 2, X)
                  /           \
                 /             *<--- This branch is pruned from the tree
                /               \    once the ! has been reached.
   gc_minimum(1, 2, 1)   gc_minimum(1, 2, 2)
              /                   \
             /                     \
            /                       \
        1 =< 2, !                 1 > 2
         /
        /
       !
```

Notice that this doesn't alter the set of results which can be obtained from running the program. All it does is pre–empt search which would be fruitless. Having put the cut into `gc_minimum/3` we might go one step further and remove the `X > Y` test in the second subgoal. The argument for doing this might be that if we know that the cut in the first clause will prevent the second clause ever being tried if `X =< Y` then the only condition under which the second clause is reached is when `X > Y`, so we don't need to make this test explicit. By making this change we get `rc_minimum/3` as shown below:

```
% rc_minimum(+Number1, +Number2, ?Minimum).
% Like to gc_minimum/3 but with the check in the second clause
% left out, because the programmer thinks it isn't needed (given
% that Y should be the chosen minimum if X isn't).

rc_minimum(X, Y, X):-
    X =< Y, !.
rc_minimum(_, Y, Y).
```

The potential search tree for the goal `rc_minimum(1,2,X)` is:

```
                rc_minimum(1, 2, X)
                 /          \
                /            *<--- This branch is pruned from the tree
               /              \    once the ! has been reached.
rc_ minimum(1, 2, 1)   rc_minimum(1, 2, 2)
             /
            /
           /
      1 =< 2, !
       /
      /
     !
```

But there is a major flaw in this reasoning, as shown by calling the goal `minimum(1, 2, 2)`, which will (erroneously) succeed, since the first clause doesn't match and the cut is therefore never reached. Thus the intended meaning of `rc_minimum/3` is different from that of `minimum/3`. Some people refer to cuts which don't alter the meaning of the program as "green cuts" and cuts which do alter the meaning of the program as "red cuts".

9.1 Using Cuts to Specify Defaults

Cuts are often used to give a range of behaviours of the program under special circumstances, with a "catch–all" default definition at the end. Consider, for example, the following program, which is discussed in Sterling & Shapiro "The Arto of Prolog". We want to define a predicate which will be able to determine a pension for some person (`X`), given certain characteristics of `X`. The default is `nothing` but, before that we put a series of tests to see if `X` qualifies for any pension. The code we use is:

```
pension(X, invalid_pension):-
    invalid(X), !.
pension(X, old_age_pension):-
    over_65(X),
    paid_up(X), !.
pension(X, supplementary_benefit):-
    over_65(X), !.
pension(_, nothing).    % This is the default if none of the above succeed.
```

Let's also define some characteristics for a sample of people.

```
invalid(fred).

over_65(fred).
```

```
over_65(joe).
over_65(jim).

paid_up(fred).
paid_up(joe).
```

This works fine for queries such as `pension(fred,P)`, which would instantiate `P` to `invalid` and return no further results. However, there are ways of forcing the program to give incorrect recommendations. In particular, the query `pension(fred,nothing)` would succeed. How can this be fixed? One way is to make it explicit that the default of `nothing` only applies when there is no possible pension. In the code below, this is achieved by inventing a new predicate, `entitlement/2` which makes this distinction.

```
entitlement(X, Y):-
    possible_pension(X, Y).
entitlement(X, nothing):-
    \+ possible_pension(X, _).

possible_pension(X, invalid_pension):-
    invalid(X).
possible_pension(X, old_age_pension):-
    over_65(X),
    paid_up(X).
possible_pension(X, supplementary_benefit):-
    over_65(X).
```

9.2 Just One More Thing About Cuts

At the start of the section on cuts, we said that they pruned the search space by telling the interpreter, once it has hit a cut, to forget about any other definitions for the predicate in which the cut appears. The cut also has another effect: it tells the interpreter not to look for any new solutions to any subgoals appearing in the body of the clause before the cut. The best way to explain this is by using an example, which you can find in Section 13.3.

Let's start with some test data:

```
baz(1, 2).
baz(2, 3).
baz(3, 4).

bar(3).
bar(4).

eek(3, a).
eek(3, b).
eek(4, c).
```

Now we define a predicate, `foo/2`, as follows:

```
foo(A, B):-
        baz(A, B),
        bar(B), !.
```

The cut at the end of this clause will commit to a single solution for `baz(A, B)` and for `bar(B)`, so the behaviour of this program is:

```
| ?- foo(A, B).
     A=2
     B=3 ;
no
```

Let's now invent a new predicate, `foo1/2`:

```
foo1(A, B, C):-
        baz(A, B),
        bar(B), !,
        eek(B, C).
```

This has an extra subgoal after the cut, so the subgoals before the cut will be "frozen", as before, but the interpreter is free to generate new solutions for the subgoal after the cut. Thus the behaviour is:

```
| ?- foo1(A, B, C).
     A=2
     B=3
     C=a ;
     A=2
     B=3
     C=b ;
no
```

10 Some Common Techniques

To become a skilled Prolog programmer requires, above all else, **practice**. However, there are a few standard ways of constructing a program which give some overall structure to the task. Some of these are described below:

Technique 1 *Recursion, decomposing a structure. The key idea is to separate base cases from recursive cases, then make sure that the recursive subgoals all tend towards one or more of the base cases. The general form of this technique is:*

$P(BaseCase)$.
$P(Structure)$:–
        $P(SmallerStructure)$.

For example:

```
member(X, [X|_]).
member(X, [_|T]):-
    member(X, T).
```

Technique 2 *Recursion, building up a structure in recursive subgoals. Typically two arguments are used: the first storing the accumulated result and the second being used to "pass back" the final state of the accumulator.*

$P(Final, Final).$
$P(Sofar, Final) : -$
$\quad Update(Sofar, NewSofar),$
$\quad P(NewSofar, Final).$

For example:

```
reverse([], Final, Final).
reverse([H|T], Sofar, Final):-
     reverse(T, [H|Sofar], Final).
```

Technique 3 *Recursion, building up a structure in the head of each clause. This normally involves just one extra argument which contains one or more variables to be instantiated from successful recursive subgoals.*

$P(BaseCase).$
$P(Final) :-$
$\quad P(Partial),$
$\quad Update(Partial, Final).$

For example:

```
numbers_in_term(A, [A]):- number(A).
numbers_in_term(A + B, Final):-
    numbers_in_term(A, PA),
    numbers_in_term(A, PB),
    append(PA, PB, Final).
```

Technique 4 *Committment by cases. If the problem is one which involves a number of mutually exclusive cases then these can be tackled using a sequence of clauses with cuts placed at the points of committment.*

$P(Term) :-$
$\quad Test1(Term), !,$
$\quad P1(Term).$

$\vdots$

$P(Term) :-$
$\quad TestN(Term), !,$
$\quad PN(Term).$

For example:

```
has_type(X, variable):-
    var(X), !.
has_type(X, atom):-
    atom(X), !.
has_type(X, number):-
    number(X), !.
has_type(X, structure):-
    nonvar(X),
    functor(X, _, N), N > 0.
```

Technique 5 *Failure driven loops to force all solutions. This is often used in combination with printing routines.*

P :–
    Goal,
    fail.
P.

For example:

```
all_solutions(Goal):-
    Goal,
    write(Goal), nl,
    fail.
all_solutions(_).
```

Technique 6 *Repeat loops to perform some task until a terminating condition is obtained. These are often used as replacements for recursive predicates in cases where a simple cycle of repeating and testing is required.*

P :–
    repeat,
    Goal,
    Test, !.

For example:

```
read_until(Prompt, Test):-
    repeat,
    write(Prompt), read(Answer),
    ((Test, !) ; fail).
```

Technique 7 *Cut and fail to ensure no solutions on particular cases.*

P :–
    UndesirableCase,
    !, *fail*.
⟨ *OTHER DEFINITIONS OF P.* ⟩

For example:

```
no_numbers([H|_]):-
    number(H),
    !, fail.
no_numbers([_|T]):-
    no_numbers(T).
no_numbers([]).
```

Technique 8 *Double negation to apply a test without binding any variables in the terms used during the test.*

$P(Term)$:–
        \+ \+ $P1(Term)$.

For example:

```
unifiable(Term1, Term2):-
    \+ \+ Term1 = Term2.
```

11 Efficiency Considerations

The quality of a program depends on 3 main metrics:

1. How easy it is to read and understand.
2. The speed with which it performs a given task.
3. The amount of space it needs to perform the task.

Metric 1 is often the most important for beginners but 2 and 3 are always hanging around in the background and need to be considered explicitly. It isn't easy to lay down precise guidelines for 1, 2 or 3 because they often interact – a program which is easy to read may not be maximally fast to run – so it is often necessary to strike a balance. Where this balance lies will depend on the demands of the application for which the program is required. The following are a few useful heuristics:

Useful Tip 1

Be careful how you order your clauses. The order of clauses and/or subgoals in a Prolog program with a declarative interpretation doesn't affect its declarative meaning buy often affects the way the program is executed. For example, the ancestor relation from the program Section 13.1 was defined as follows:

```
ancestor(A, B):-
    parent(A, B).
ancestor(A, B):-
    parent(P, B),
    ancestor(A, P).
```

By switching the ordering of the clauses and/or subgoals in this definition we can obtain 3 different ways of defining the ancestor relation:

- `ancestor1` has the two clauses switched around.
- `ancestor2` has the order of subgoals switched around.
- `ancestor3` has the order of both clauses and subgoals switched around.

This provides the definitions shown below:

```
ancestor1(A, B):-
    parent(P, B),
    ancestor1(A, P).
ancestor1(A, B):-
    parent(A, B).

ancestor2(A, B):-
    parent(A, B).
ancestor2(A, B):-
    ancestor2(A, P),
    parent(P, B).

ancestor3(A, B):-
    ancestor3(A, P),
    parent(P, B).
ancestor3(A, B):-
    parent(A, B).
```

All of these variants of `ancestor/2` have the same declarative meaning; the order of clauses or subgoals doesn't affect this. However, the way they are executed by the Prolog interpreter is very different, giving a different external behaviour of the program. Consider the following transcript of a Prolog session, in which we have first loaded the definitions in Section 13.1 and then the variants of `ancestor` given above. We then give the goal to find some ancestor of `dave` using each of the definitions in turn :

```
yes
| ?- ancestor(X, dave).

     X=clive ;

     X=clarissa ;

     X=alan ;

     X=andrea ;
```

```
     X=bruce ;

     X=betty
yes
| ?- ancestor1(X, dave).

     X=alan ;

     X=andrea ;

     X=bruce ;

     X=betty ;

     X=clive ;

     X=clarissa
yes
| ?- ancestor2(X, dave).

     X=clive ;

     X=clarissa ;

     X=alan ;

     X=andrea ;

     X=bruce ;

     X=betty
yes
| ?- ancestor3(X, dave).

%%% Local stack overflow - forced to abort %%%
```

Although `ancestor1` and `ancestor2` return the same values for `X`, these values are returned in a different order. This is because in `ancestor1` the recursive clause appears before the base case, so the interpreter always tries to find a solution "the long way" before considering the shorter alternative. In `ancestor3`, the combination of putting the recursive case first and putting the recursive call to `ancestor3` before the step to parent causes the interpreter to descend through an endless chain of subgoals until it eventually runs out of memory and drops dead.

The moral of the story is : Your Prolog program may have a nice declarative interpretation but if you want it to execute you must often consider its procedural aspects.

Useful Tip 2

Be careful how you call your predicates. In the `colour_countries/1` example in Section 13.4 the order in which countries are coloured makes a big difference to the speed of solution. You can test this by changing the order of clauses for `ngb/2`, which gives the neighbouring countries. The general principle is to select the (potentially) most difficult countries and colour them first. The ones likely to be most difficult are the ones with most neighbours. (See Bratko, "Prolog Programming for AI" for a detailed discussion of efficiency for the map colouring problem).

Useful Tip 3

Take advantage of tail recursion optimisation. This is a technique built into many Prolog systems which allows the space allocated to the head of a clause to be used for the last subgoal, provided that there is no chance of further results on backtracking by the time the last subgoal gets called. Consider, for example, the `append/3` predicate:

```
append([H|T], L1, [H|L2]):-
    append(T, L1, L2).
append([], L, L).
```

For this predicate, if we are trying to satisfy a goal such as:

```
| ?- append([a,b], [c,d], X).
```

and have managed to match on the head of the first clause and are at the point of calling the last (and only) subgoal of that clause then we know that there would be no other ways of finding a solution (since there are no other subgoals within the clause and the 2nd clause wouldn't match to the goal). If the Prolog interpreter is smart enough to spot this, it can get rid of any choice points for the current goal (`append([a,b], [c,d], X)`) and replace it in memory with the outstanding subgoal (`append([b], [c,d], X)`). This means that a solution to the `append/3` goal can be found using constant memory space, rather than requiring space which increases linearly with the depth of recursion. Of course, this all depends on the interpreter being able to recognise when tail recursion optimisation can be applied - a task which isn't always easy. The `append/3` example, above, is comparatively simple but even it isn't straightforward. Suppose that the goal had been:

```
| ?- append(X, Y, [a,b]).
```

Then tail recursion optimisation wouldn't apply because the program isn't deterministic at the point where the subgoal of the 1st clause gets called. Some Prolog systems have fancy indexing systems which allow them to detect determinism quite effectively. However, there is one way in which you can make it clear to the Prolog interpreter that no choice points remain at the point where the last subgoal of a clause is about to be called....add a cut. For example, suppose we have the following program which is designed to gobble up space on the machine:

```
gobble_space:-
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    statistics(local_stack, L), write(L), nl,
    gobble_space.
gobble_space.

gobble.
gobble:-
    gobble_space.
```

The behaviour of this predicate is as follows:

```
| ?-  gobble_space.
[380,16000]
[652,15728]
......SOME TIME LATER.....
[4190812,2468]
[4191084,2196]
{ERROR: Memory allocation failed}
{ Execution aborted }
```

i.e. It eats up a lot of space and eventually exceeds the limits of the system. Now let's try the same program but with a cut in the first clause of `gobble_space/0` to permit tail recursion optimisation.

```
gobble_space:-
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    gobble,
    statistics(local_stack, L), write(L), nl, !,
    gobble_space.
gobble_space.
```

The behaviour of this predicate is:

```
| ?- gobble_space.
[380,16000]
[380,16000]
......SOME TIME LATER.....
[380,16000]
[380,16000]
```

i.e. It doesn't eat up space and will continue to run indefinitely (or at least for a very long time).

Useful Tip 4

Don't use cuts indiscriminately. Here is an example taken from O'Keefe, "The Craft of Prolog". The plan is to define a predicate, `match1/2`, which will take as its first argument a pattern for a sequence (with some "holes" in it) and take as its second argument a complete sequence of symbols. The predicate should succeed if the pattern can be matched to the sequence. For example:

```
| ?- match1([i,X,my,Y], [i,like,my,lecturing,job]).
X = [like],
Y = [lecturing,job]
```

One way to define it is as follows:

```
match1([], []).
match1([H|T], [F|R]):-
    nonvar(H), !,
    H = F,
    match1(T, R).
match1([V|T], [F|R]):-
    match1(T, R),
    V = [F].
match1([V|T], [F|R]):-
    match1([New|T], R),
    V = [F|New].
```

This works, but is not as fast as the following definition, named `match2/2`, which has no cuts but distinguishes the type of symbol in the pattern list using `w(X)`, if `X` is a word, and `s(X)`, if `X` is a segment of the sequence.

```
match2([], []).
match2([H|T], [F|R]):-
    match2(H, T, F, R).

match2(w(Word), Items, Word, Words):-
    match2(Items, Words).
match2(s([Word|Seg]), Items, Word, Words0):-
```

```
    append(Seg, Words1, Words0),
    match2(Items, Words1).
```

The behaviour of this predicate is as follows:

```
| ?- match2([w(i),s(X),w(my),s(Y)], [i,like,my,lecturing,job]).
X = [like],
Y = [lecturing,job]
```

Useful Tip 5

For some tasks you can save space by using failure driven loops. For example, suppose we want to write a predicate, `read_file/1`, which will read each clause, `C`, from a given file and assert them into the database as a series of facts of the form `cl(C)`. The first part of this predicate takes care of opening the file; calling the subgoal, `read_all_clauses`, which does the reading and asserting; and closing the file at the end.

```
read_file(File):-
    open(File, read, Stream),
    set_input(Stream),
    read_all_clauses,
    close(Stream).
```

We could write a recursive definition of `read_all_clauses/0`:

```
read_all_clauses:-
    read(Clause),
    ((Clause = end_of_file, !) ;
     (assert(cl(Clause)), read_all_clauses)).
```

But a failure driven version will be more efficient:

```
read_all_clauses:-
    repeat,
    read(Clause),
    ((Clause = end_of_file, !) ;
     (assert(cl(Clause)), fail)).
```

Useful Tip 6

Don't duplicate effort. If you are going to need to do the same, involved calculation a number of times then it is probably worth storing the result of the first calculation for later use.

12 Definite Clause Grammars

DCGs are a special notation provided in most Prolog systems which provide you with a convenient way of defining grammar rules. The general form of each DCG clause is as follows:

```
Head --> Body
```

meaning "A possible form for Head is Body". `Head` and `Body` are Prolog terms and '`-->`' is an infix operator.

The most common use of DCGs is to parse (or generate) some list of symbols according to valid "rules of grammar" defined by the DCG clauses. DCGs can be used for purposes other than parsing but for the purposes of this explanation we will stick with the standard parsing application.

Agree to call the symbols (usually words) in the parsed sentence "terminal symbols". Also agree to call symbols which represent higher level language constructs "non-terminal symbols".

- A non-terminal symbol may be any Prolog term, other than a variable or number.
- A terminal symbol may be any Prolog term.
- DCG heads consist of a non-terminal symbol, optionally followed by a sequence of terminal symbols.
- DCG bodies may contain any sequence of terminal or non-terminal symbols, separated by the '`,`' (conjunction) and/or '`;`' (disjunction) operators.
- In the body of a DCG clause, terminal symbols are distinguished from non-terminal symbols by being enclosed within a Prolog list: `[T1,T2,...TN]` An empty sequence of terminal symbols is represented by the empty list `[]`.
- Extra conditions, in the form of Prolog procedure calls, may be included in the body of DCG clause. These are distinguished by being enclosed in curly brackets: `{P1,P2,...PN}`
- The cut symbol can be included in the body of a DCG clause and doesn't have to be enclosed within curly brackets.

12.1 How do DCGs relate to standard Prolog clauses ?

DCGs are really just a convenient abbreviation for ordinary Prolog clauses. Each DCG clause is translated into a standard Prolog clause as it is loaded. The procedural interpretation of a grammar rule is that it takes an input list of symbols, analyses some initial portion of that list and produces the remaining portion (possibly enlarged) as output for further analysis. The arguments required for the input and output lists are not written explicitly in a grammar rule but are added when the rule is translated into an ordinary Prolog clause. For example, the DCG rule:

```
p(X, Y) --> q(X), r(X, Y), s(Y).
```

translates into:

```
p(X, Y, Input, Output):-
    q(X, Input, Out1),
    r(X, Y, Out1, Out2),
    s(Y, Out2, Output).
```

Terminal symbols are translated using the built in predicate :

```
'C'(Input, Symbol, Remainder)
```

which nips the given Symbol from the Input list, leaving the Remainder of the list of terminal symbols. It is defined by the single clause:

```
'C'([X|Remainder], X, Remainder).
```

This predicate isn't normally accessed by the programmer, hence the obscure name. This gives us the following translation for a DCG rule containing both terminal and non-terminal symbols:

```
p(X) --> [go,to], q(X), [stop].
```

is translated into:

```
p(X, Input, Output):-
    'C'(Input, go, Out1),
    'C'(Out1, to, Out2),
    q(X, Out2, Out3),
    'C'(Out3, stop, Output).
```

Extra conditions expressed as procedure calls just appear as themselves, without any extra translation, so:

```
p(X) --> [X], {integer(X), X > 0}, q(X).
```

is translated into:

```
p(X, Input, Output):-
    'C'(Onput, X, Out1),
    integer(X),
    X > 0,
    q(X, Out1, Output).
```

12.2 Finding out for yourself

If you want to experiment with DCGs here are a few things you should know:
There are two ways to call a procedure which you have defined using DCGs:

- Using the built in predicate `phrase/2`. This takes as its first argument the highest level DCG rule head to apply (think of this as being like a top level Prolog goal) and as its second argument the list of terminal symbols which can be parsed using the group of DCG clauses referenced by the term in the first argument.
- By just calling the translated clauses directly (see details of translation above).

If you want to see how the Prolog system will translate your DCGs into ordinary clauses there are two ways in which you can do this:

- Call the built in predicate `expand_term/2` with the first argument containing a DCG clause. It will return the expanded translation as its second argument. For example:

```
| ?- expand_term(a --> b, X).
     X = a(_124,_133):-b(_124,_133)
```

- Load a DCG from a file and then call the built in predicate `listing/0` which will list out all the loaded clauses in translated form.

13 Code Examples

This section contains example programs used by other parts of the tutorial.

13.1 Ancestry Program

```
/* Suppose we have a family tree like this :

alan andrea   bruce betty      eddie elsie   fred  freda
 |     |        |     |          |     |       |     |
 |_____|        |_____|          |_____|       |_____|
    |              |                |             |
  clive        clarissa            greg         greta
   |  |__________|___|              |             |
   |__________|__|                  |_____________|
          |   |                            |
        dave doris                        henry

which is defined in Prolog by the following 3 sets of predicates:
```

```
*/

%   parent(Parent, Child).
%   Parent is the parent of Child.

parent(alan, clive).
parent(andrea, clive).
parent(bruce, clarissa).
parent(betty, clarissa).
parent(clive, dave).
parent(clarissa, dave).
parent(clive, doris).
parent(clarissa, doris).
parent(eddie, greg).
parent(elsie, greg).
parent(fred, greta).
parent(freda, greta).
parent(greg, henry).
parent(greta, henry).


%   male(Person).
%   This Person is male.

male(alan).
male(bruce).
male(clive).
male(dave).
male(eddie).
male(fred).
male(greg).
male(henry).


%   female(Person).
%   This Person is female.

female(andrea).
female(betty).
female(clarissa).
female(doris).
female(elsie).
female(freda).
female(greta).
```

```
%   married(Person1, Person2).
%   Person1 is married to Person2.

married(alan, andrea).
married(bruce, betty).
married(clive, clarissa).
married(eddie, elsie).
married(fred, freda).
married(greg, greta).

%   PROBLEM 1
%   How do you find out if someone is the ancestor of someone else ?

ancestor(A, B):-        % A is B's ancestor if
    parent(A, B).       % A is B's parent.

ancestor(A, B):-        % A is B's ancestor if
    parent(P, B),       % some person P is B's parent and
    ancestor(A, P).     % A is P's ancestor.


%   PROBLEM 2
%   How do you find out if someone is the descendant of someone else ?

descendant(A, B):-      % A is B's descendant if
    parent(B, A).       % B is A's parent.

descendant(A, B):-      % A is B's descendant if
    parent(B, P),       % B is a parent of some person P and
    descendant(A, P).   % A is P's descendant.


%   PROBLEM 3
%   How do you know if someone is related to someone else ?

related(A, B):-         % A is related to B if
    ancestor(A, B).     % A is B's ancestor.

related(A, B):-         % A is related to B if
    descendant(A, B).   % A is B's descendant.

related(A, B):-         % A is related to B if
    ancestor(X, A),     % some person X is A's ancestor and
    ancestor(X, B),     % X is also B's ancestor and
    A \== B.            % A is not the same as B.
```

```
%   There are lots of other different ways of solving this problem.


%   PROBLEM 4
%   How can you decide whether two people could possibly marry, given that
%   only an unrelated male and female are allowed to do this ?

possible_to_marry(A, B):-       % It is possible for A to marry B if
    male(A),                    % A is male and
    female(B),                  % B is female and
    \+ related(A, B).           % A is not related to B.


%   PROBLEM 5
%   How can you decide whether two people can marry, given the current
%   matrimonial ties ?

can_marry(A, B):-               % A can marry B if
    possible_to_marry(A, B),    % it is possible for A to marry B and
    \+ matrimonial_tie(A),      % there is no matrimonial tie for A and
    \+ matrimonial_tie(B).      % there is no matrimonial tie for B.

matrimonial_tie(X):-    % X is matrimonially tied if
    married(X, _).      % X is married to somebody.

matrimonial_tie(X):-    % X is matrimonially tied if
    married(_, X).      % somebody is married to X.


%   FOOTNOTE
%   The symbol \== means "not the same as" (e.g. foo \== baz is true).
%   The symbol \+ means "not provable" (e.g. \+ false is always true).
%   The symbol _ represents an "anonymous variable" (i.e. a variable
%              for which there is no need for a particular name).
%   Comments are on lines beginning with %
%            or in sections bounded by /* and */
```

13.2 Cut Demonstration

```
% Experimenting with the use of the ''cut'' predicate
% (after Sterling and Shapiro ''The Art of Prolog'')

%       GREEN CUTS
%       ----------
```

```
%       minimum(+Number1, +Number2, ?Minimum).
%       Succeeds if Minimum is the minimum value of Number1 and Number2.

minimum(X, Y, X):-
        X =< Y.
minimum(X, Y, Y):-
        X > Y.

/*
        The potential search tree for the goal minimum(1,2,X) is:

                 minimum(1, 2, X)
                 /             \
                /               \
        minimum(1, 2, 1)    minimum(1, 2, 2)
             /                     \
            /                       \
           /                         \
        1 =< 2                     1 > 2
*/

%       gc_minimum(+Number1, +Number2, ?Minimum).
%       Identical to minimum/3 but with a cut in the first clause.

gc_minimum(X, Y, X):-
        X =< Y, !.
gc_minimum(X, Y, Y):-
        X > Y.

/*
        The potential search tree for the goal minimum(1,2,X) is:

                 minimum(1, 2, X)
                  /           \
                 /             *<--- This branch is pruned from the tree
                /               \    once the ! has been reached.
         minimum(1, 2, 1)   minimum(1, 2, 2)
              /                    \
             /                      \
            /                        \
         1 =< 2, !                 1 > 2
          /
         /
        !
*/
```

```
%       Another "green cut" example:

%       delete(+List, +Item, ?Result).
%       Removes all instances of Item from List, returning the Result.

delete([H|T], H, Result):- !,
        delete(T, H, Result).
delete([H|T], X, [H|Rest]):-
        \+ H = X, !,
        delete(T, X, Rest).
delete([], _, []).


%       RED CUTS
%       --------

%       rc_minimum(+Number1, +Number2, ?Minimum).
%       Like to gc_minimum/3 but with the check in the second clause
%       left out, because the programmer thinks it isn't needed (given
%       that Y should be the chosen minimum if X isn't).

rc_minimum(X, Y, X):-
        X =< Y, !.
rc_minimum(_, Y, Y).

/*
        The potential search tree for the goal minimum(1,2,X) is:

                 minimum(1, 2, X)
                  /          \
                 /            *<--- This branch is pruned from the tree
                /              \    once the ! has been reached.
         minimum(1, 2, 1)   minimum(1, 2, 2)
              /
             /
            /
         1 =< 2, !
          /
         /
        !


        **BUT** There is a major flaw in this reasoning, as shown by
        calling the goal minimum(1, 2, 2), which will (erroneously) succeed.
```

```
        i.e. The intended meaning of minimum has been lost.

*/


%       Another "red cut" example:

%       rc_delete(+List, +Item, ?Result).
%       Like delete/3, but omits the non-unification check in the second
%       clause.  This makes the meaning of the code more obscure.

rc_delete([H|T], H, Result):- !,
        rc_delete(T, H, Result).
rc_delete([H|T], X, [H|Rest]):- !,
        rc_delete(T, X, Rest).
rc_delete([], _, []).



%       USING CUTS TO SPECIFY DEFAULTS
%       ------------------------------

pension(X, invalid_pension):-
        invalid(X), !.
pension(X, old_age_pension):-
        over_65(X),
        paid_up(X), !.
pension(X, supplementary_benefit):-
        over_65(X), !.
pension(_, nothing).    % This is the default if none of the above succeed.

invalid(fred).

over_65(fred).
over_65(joe).
over_65(jim).

paid_up(fred).
paid_up(joe).

/*
        **BUT** This doesn't work in certain cases.  For example:
        the query pension(fred,nothing) would succeed.
*/

entitlement(X, Y):-
```

```
        possible_pension(X, Y).
entitlement(X, nothing):-
        \+ possible_pension(X, _).

possible_pension(X, invalid_pension):-
        invalid(X).
possible_pension(X, old_age_pension):-
        over_65(X),
        paid_up(X).
possible_pension(X, supplementary_benefit):-
        over_65(X).

/*
        This is a bit better, and the query entitlement(fred, nothing)
        will now succeed.
*/


%       CUT AND FAIL
%       ------------

%       not(+X).
%       Succeeds if X fails.

not(X):-
        X, !, fail.
not(_).
```

13.3 More on Cuts

```
% Some test data on which we shall test various procedures which contain cuts.

baz(1, 2).
baz(2, 3).
baz(3, 4).

bar(3).
bar(4).

eek(3, a).
eek(3, b).
eek(4, c).


% EXAMPLE 1 : Cut as final subgoal returns only one solution, although
%             several would be possible if the cut were removed.
```

```
%
%       | ?- foo(A, B).
%
%            A=2
%            B=3 ;
%       no

foo(A, B):-
        baz(A, B),
        bar(B), !.

% EXAMPLE 2 : A third subgoal is added after the cut in the code from EXAMPLE 1
%             We now get two solutions because eek/2 is not frozen by the cut.
%
%       | ?- foo1(A, B, C).
%
%            A=2
%            B=3
%            C=a ;
%
%            A=2
%            B=3
%            C=b ;
%       no

foo1(A, B, C):-
        baz(A, B),
        bar(B), !,
        eek(B, C).


% EXAMPLE 3 : Produces the same results as EXAMPLE 2 but using foo/2 as
%             a subgoal.
%
%       | ?- foo2(A, B, C).
%
%            A=2
%            B=3
%            C=a ;
%
%            A=2
%            B=3
%            C=b ;
%       no

foo2(A, B, C):-
```

```
        foo(A, B),
        eek(B, C).
```

13.4 Map Colouring

```
% A map colouring program (after Bratko, ''Prolog Programming for AI'' p190)

% THE PROBLEM : To find a way of colouring a map using only 4 colours in
%               such a way that no pair of neighbouring countries are
%               the same colour.
%*******************************************************************

% Predicate : colour_countries(-List).
% Returns a List of terms of the form:
%       Country/Colour where - Country is the name of some country.
%                            - Colour is the map colour for that country.
% This program gives the following behaviour:
%
% | ?- colour_countries(X).
%
%     X=[austria/red,belgium/green,denmark/blue,france/red,
%       italy/yellow,netherlands/blue,portugal/blue,
%       spain/yellow,switzerland/blue,w_germany/yellow]
%yes

% colour_countries/1 works by first using setof/3 to construct a list of terms
% of the form : Country/Var where - Country is the name of a country
%                                   Var is a variable
% It then uses colours/2 to bind each Var in this list to an appropriate colour

colour_countries(Colours):-
        setof(Country/_, X^ngb(Country,X), Colours),
        colours(Colours).

% Predicate : colours(+List)
% Succeeds when, given a List of elements of form Country/Colour, a value
% for each Colour can be found such that the Country to which it is attached
% has no neighbour of the same colour.

        % If there are no elements left in the list then just succeed.
colours([]).
        % For a list with head Country/Colour and tail Rest, colour all
        % the Rest then select a value for Colour from the list of candidates
        % then check that there is no country in Rest which neighbours
        % the Country just coloured and has the same Colour.,
colours([Country/Colour|Rest]):-
```

```
        colours(Rest),
        member(Colour, [yellow,blue,red,green]),
        \+ (member(Country1/Colour, Rest), neighbour(Country, Country1)).


% Predicate : neighbour(Country, Neighbour)
% Succeeds if Neighbour is a neighbour of Country

neighbour(Country, Country1):-
        ngb(Country, Neighbours),
        member(Country1, Neighbours).


% Predicate : ngb(Country, Neighbours).
% Determines the list of Neighbours of a Country.

ngb(portugal, [spain]).
ngb(spain, [portugal,france]).
ngb(france, [spain,belgium,switzerland,w_germany,italy]).
ngb(belgium, [france,w_germany,netherlands]).
ngb(netherlands, [belgium,w_germany]).
ngb(w_germany, [netherlands,belgium,france,switzerland,austria,denmark]).
ngb(switzerland, [france,w_germany,austria,italy]).
ngb(austria, [w_germany,switzerland,italy]).
ngb(italy, [france,switzerland,austria]).
ngb(denmark, [w_germany]).


% Predicate : member(Element, List)
% Standard membership utility

member(X, [X|_]).
member(X, [_|T]):-
        member(X, T).
```

Refinement of the Operator Model

This page describes formally how we refine Figure 2.5 into a more detailed model so that we can connect it later to a decision procedure. To do this, we must make explicit the output of each operator and the conditions which must be satisfied to produce it. The translation for each link on the diagram is as follows:

- Find the label, S, of the source operator of the link. Construct the type restriction $V_s \in S$, where V_s is a new variable.
- Find the label, T, of the target operator of the link. Construct the type restriction $V_t \in T$, where V_t is a new variable.
- Find the label of the link, which will be of the form $P(A_1, \cdots, A_n)$.
- Construct the term $P(V_1, \cdots, V_n)$, where each V_i is a new variable, and the set $\{V_1 \in A_1, \cdots, V_n \in A_n\}$ of type restrictions on the variables.
- Merge $\{V_1 \in A_1, \cdots, V_n \in A_n\}$ with $\{V_s \in S, V_t \in T\}$ to give $\{V_1 \in A_1, \cdots, V_m \in A_m\}$.
- The final expression is $V_1 \in A_1. \cdots .V_n \in A_m.stream\_link(P(V_1, \cdots, V_n), V_s, V_t)$

Applying this procedure to each link in Figure 2.5 we obtain:

$$P \in person.B \in branch.L \in loan\_dept.\ stream\_link(loan\_request(P, B), B, L) \quad (1)$$
$$P \in person.B \in branch.C \in central\_admin.\ stream\_link(defaulter(P), B, C) \quad (2)$$
$$P \in person.R \in answer.L \in loan\_dept.B \in branch.\ stream\_link(outcome(P, R), L, B) \quad (3)$$
$$P \in person.L \in loan\_dept.C \in central\_admin.\ stream\_link(info\_request(P), L, C) \quad (4)$$
$$D \in datastore.C \in central\_admin.\ stream\_link(database(D), C, C) \quad (5)$$
$$P \in person.I \in advice.C \in central\_admin.L \in loan\_dept.\ stream\_link(loan\_info(P, I), C, L) (6)$$

This definition of *stream_link* generalises the structure of the diagram by the use of typed variables, which is an intermediate step in defining the operators. These are described by the predicate *output* which has three arguments: the operator name; the data output; and the conditions which must be satisfied in order for the output to be produced. These conditions are of two kinds: inputs to the operator which are shown as links in the operator diagram; and conditions internal to the operator which will be specified in the next model. The refinement from *stream_link* definitions (1 to 6) to operator definitions is as follows:

1. Rewrite $Q_1 \cdots Q_i.stream\_link(O, F, T)$ to $Q_1 \cdots Q_i.output(F, O, true)$, where $Q_1 \cdots Q_i$ are the quantifiers for the link and $true$ denotes a condition which is always true.

2. Replace each of the quantifiers, $Q_1 \cdots Q_i$, in one of the following ways (assuming the quantifier we select at each stage is of the form $X \in S$):

   (a) Replace one or more of the quantifiers, each of the form $X \in S$, by a condition of the form $S'(X)$ (where S' is a subsort of S).

   (b) Remove $X \in S$ and instantiate X to S'.

   (c) Replace $X \in S$ by an operator condition of the form $input(I)$. The new input I must refer to X and must appear in another stream link of the form $stream\_link(I, F'', F)$ (in other words, we must have already stipulated that an input stream exists for it). It is the designer's responsibility to ensure that this condition maintains the sort condition imposed by S.

   (d) Replace $X \in S$ by an operator condition involving X. These conditions relate the operator's communication channels and its internal decisions. It is the designer's responsibility to ensure that each condition maintains the sort condition imposed by S.

3. The designer can choose to further constrain the operator by adding more input or operator conditions, with the proviso that input conditions must have *stream_link* connections as stipulated above.

Applying this to expression 3 allows us to make the following refinement steps:

$$
\begin{array}{l}
P \in person.R \in answer.L \in loan\_dept.B \in branch.\ stream\_link(outcome(P,R), L, B) \\
\qquad\qquad\qquad \Downarrow 1 \\
P \in person.R \in answer.L \in loan\_dept.B \in branch.\ output(L, outcome(P,R), true) \\
\qquad\qquad\qquad \Downarrow 2\text{b} \\
P \in person.R \in answer.B \in branch.\ output(loan\_dept, outcome(P,R), true) \\
\qquad\qquad\qquad \Downarrow 2\text{a} \\
R \in answer.B \in branch.\ output(loan\_dept, outcome(P,R), true) \leftarrow person(P) \\
\qquad\qquad\qquad \Downarrow 2\text{c} \\
R \in answer.\ output(loan\_dept, outcome(P,R), input(loan\_request(P,B))) \leftarrow person(P) \\
\qquad\qquad\qquad \Downarrow 3 \\
R \in answer.\ output(loan\_dept, outcome(P,R), \left(\begin{array}{c} input(loan\_request(P,B)) \wedge \\ input(loan\_info(P,I)) \end{array} \right)) \leftarrow person(P) \\
\qquad\qquad\qquad \Downarrow 2\text{d} \\
output(loan\_dept, outcome(P,R), \left(\begin{array}{c} input(loan\_request(P,B)) \wedge \\ input(loan\_info(P,I)) \wedge \\ loan\_decision(P,B,I,R) \end{array} \right)) \leftarrow person(P)
\end{array}
$$

Formalisation of the Decision Procedure Model

In this page we translate the information in Figure 2.6 into a more conventional logical notation, ready for further refinement. As in the translation of our operator diagram, the algorithm looks complex but can be automated. Our method for this is as follows:

- Construct the top-level expression:
  - Look up the top-level procedure on the diagram, which is of the form $P(S_1, \cdots, S_n)$. Construct an expression: $(V_1 \in S_1. \cdots .V_n \in S_n.P(V_1, \cdots, V_n)$ where each V_i is a new variable restricted to type S_i. For example the top-level expression in Figure 2.6 is: $P \in person.D \in datastore.R \in info.risk\_assessment(P, D, R)$.
  - For each function oval, labelled F, on the diagram, find the set, D, consisting of the data types $\{T_1, \cdots, T_{m-1}\}$ on which it depends (found by following backwards along the arrows on the diagram to its data sources) and the type T_m of its result. Now construct a term $F(W_1, \cdots, W_m)$ where each W_j is either a variable V_k, if T_j is the same type as S_k, or, if not, is a new variable. Any new variables are existentially quantified. For example the *select_model* function on Figure 2.6 yields the expression: $\exists M \in model.select\_model(P, D, M)$.
  - Construct the expression: $(V_1 \in S_1. \cdots .V_n \in S_n.(P(V_1, \cdots, V_n) \leftarrow C)$, the left side of which comes from the first step and where C is the conjunction of all the terms from the previous step. For our example, this gives us expression 7 shown below.
- For each function oval, labelled F, find the set of data types $\{T_1, \cdots, T_p\}$ on which it depends (by following backwards along the arrows on the diagram to its data sources). Find the set of data types $\{T_{p+1}, \cdots, T_q\}$ which the function produces (by following the arrows leading out of its oval). Construct an expression $(V_1 \in T_1, \cdots, V_q \in T_q.P(V_1, \cdots, V_q)$ where each V_i is a new variable restricted to type T_i. All the variables $T_{p+1}, \cdots, T_q$ are existentially quantified. For example the *select_model* function oval in Figure 2.6 yields expression 8 shown below.

We have a risk assessment for a person and some data source if there is some model which we can select for that person and data, and we can assess the loan risk based on that model.

$$P \in person.D \in datastore.R \in info. \\ risk\_assessment(P, D, R) \leftarrow \exists M \in model. \left(\begin{array}{c} select\_model(P, D, M) \wedge \\ assess\_risk(M, R) \end{array} \right) \tag{7}$$

For every person and data store we can select some model.

$$P \in person.D \in datastore.\exists M \in model.select\_model(P, D, M) \tag{8}$$

Building a Formal Interpreter for the Operator Model

This page defines a mechanism for analysing this structure of our operator model to see what behaviours it will allow. The basic support tree is constructed by nesting the following terms:

- $s(Op, X)$ records that the output, X, from operator, Op, was available from the pool of data streams.
- $out(Op, X, \mathcal{T})$ records that operator, Op, produced output, X, given the support $\mathcal{T}$.
- $and(\mathcal{T}_1, \mathcal{T}_2)$ records that support trees $\mathcal{T}_1$ and $\mathcal{T}_2$ were both needed.
- $imp(C, \mathcal{T})$ records that support tree $\mathcal{T}$ was needed to conclude C.
- Any unit goal can appear as leaf of our support tree.

We now need a mechanism for generating this support tree for any operator's output - which should be possible whenever the conditions for the corresponding operator are satisfied. We use the predicate $support\_tree(Op, X, \mathcal{T})$ to determine whether a support tree, $\mathcal{T}$, exists for the output, X, of operator, Op. It acts as a bridge to the predicate which does the real work: $p\_output(Op, X, D, S_i, S_f, \mathcal{T})$, where D is the maximum number of steps we are allowed to take between operators; S_i is the set of initial operator outputs appearing in the pool of output streams; and S_f is the final pool of output streams when X has been obtained. The use of D is to limit the length of each chain of operator applications - otherwise we might find that our mechanism gets bogged down in repeating similar operator sequences *ad infinitum*. There are a variety of other ways of limiting search but it would be a digression to discuss these here. For our example, a convenient number for D is 4. We also stipulate S_i to be [], denoting that there are initially no outputs in the pool.

We can generate a support tree, T, for the output, X, of operator, Op, if X is a possible output, given no initial outputs in the pool of streams and no sequence of operators longer than 4.

$$support\_tree(Op, X, \mathcal{T}) \quad \leftarrow \quad p\_output(Op, X, 4, [\,], S_f, \mathcal{T}) \tag{9}$$

X is a possible output for operator Op, given the current output streams, S_c, if either of the following hold.

- *It appears in S_c - in which case the support tree is $s(Op, X)$.*
- *We haven't exceeded the maximum length of operator sequence, D, and it doesn't appear in S_c and and there is an operator definition for Op which*

supplies X given the condition, C, and this condition can be satisfied by generating support sub-tree $\mathcal{T}$ - in which case the final support tree is $out(Op, X, \mathcal{T})$.

$$
\begin{aligned}
p\_output(Op, X, D, S_c, S_c, s(Op, X)) &\leftarrow s(Op, X) \in S_c & (10)\\
p\_output(Op, X, D, S_c, S_f, out(Op, X, \mathcal{T})) &\leftarrow D > 0 \wedge & (11)\\
&\quad not(s(Op, X) \in S_c) \wedge\\
&\quad output(Op, X, C) \wedge\\
&\quad D_n \; is \; D - 1 \wedge\\
&\quad satisfy(C, D_n, S_c, S_f, \mathcal{T})
\end{aligned}
$$

We can satisfy a condition, C, with current output streams S_c, giving final output streams S_f and support tree $\mathcal{T}$ if any of the following hold:

- *C is of the form $input(X)$ and X is a possible output for some operator.*
- *C is of the form $(A \wedge B)$ and A can be satisfied (yielding new output streams S_n and support sub-tree $\mathcal{T}_a$) and B can be satisfied (yielding final output streams S_f and support sub-tree $\mathcal{T}_b$). The final support tree is $and(\mathcal{T}_a, \mathcal{T}_b)$.*
- *C is a system defined predicate and it can be satisfied directly by the Prolog system. The final support tree is C itself.*
- *C is neither a system predicate, nor a conjunction, nor an input. There is a clause which concludes C given precondition P and P can be satisfied, yielding support tree $\mathcal{T}_p$. The final support tree is $imp(C, \mathcal{T}_p)$*

$$
\begin{aligned}
satisfy(input(X), D, S_c, [s(Op, X)|S_r], \mathcal{T}) &\leftarrow p\_output(Op, X, D, S_c, S_r, \mathcal{T}) & (12)\\
satisfy((A \wedge B), D, S_c, S_f, and(\mathcal{T}_a, \mathcal{T}_b)) &\leftarrow satisfy(A, D, S_c, S_n, \mathcal{T}_a) \wedge\\
&\quad satisfy(B, D, S_n, S_f, \mathcal{T}_b) & (13)\\
satisfy(C, D, S_c, S_c, C) &\leftarrow not(C = (A \wedge B)) \wedge\\
&\quad system\_defined(C) \wedge\\
&\quad C & (14)\\
satisfy(C, D, S_c, S_f, imp(C, \mathcal{T}_p)) &\leftarrow not(system\_defined(C)) \wedge\\
&\quad not(C = (A \wedge B)) \wedge\\
&\quad not(C = input(X)) \wedge\\
&\quad clause(C, P) \wedge\\
&\quad satisfy(P, D, S_c, S_f, \mathcal{T}_p) & (15)
\end{aligned}
$$

We can now use definitions 9 to 15 to determine which outputs can be generated by the operators in our system. The most interesting output is the final loan decision from the loan department, which we can test by finding solutions to the goal:

$$support\_tree(loan\_dept, outcome(P, R), \mathcal{T})$$

where P will be instantiated to the name of a person, R will be instantiated to a loan decision for that person (which will either be $agree$ or $disagree$) and $\mathcal{T}$ will be the corresponding support tree - allowing us to analyse how the result was obtained. A diagrammatic version of one legitimate support tree appears in Figure 2.7.

Building a Formal Interpreter to Test Inconsistency in the Model

This page describes one way in which the model interpreter used earlier in the chaper to generate support trees can be extended to highlight inconsistencies in model behaviour. Our first step is to add a different definition for satisfaction of conditions from that given by the predicate $satisfy$ in the earlier interpreter. For this we define the new predicate $satisfy1$ which succeeds if a condition can be both confirmed and denied (in which case the support tree records the support for and against the condition) or if the condition can be confirmed but not denied (in which case the support tree its support).

$$
\begin{aligned}
satisfy1(C, D, S_c, S_f, inconsistent(\mathcal{T}_c, \mathcal{T}_d)) &\leftarrow satisfy(C, D, S_c, S_f, \mathcal{T}_c) \wedge \qquad (16) \\
&\quad\ \exists S_d.deny(C, D, S_c, S_d, \mathcal{T}_d) \\
satisfy1(C, D, S_c, S_f, consistent(\mathcal{T}_c)) &\leftarrow satisfy(C, D, S_c, S_f, \mathcal{T}_c) \wedge \qquad (17) \\
&\quad\ not(\exists S_d.deny(C, D, S_c, S_d, \mathcal{T}_d))
\end{aligned}
$$

Whenever we are attempting to find some input for an operator (expression 18) we check to see if we know of another input which might conflict with it and, if so, attempt to confirm the conflicting input using our existing definition of $satisfy$. If the conflicting input is confirmed then the support tree contains the pair of inputs which are conflicting and the support for the conflicting one. Since inputs often appear in conjunctions of conditions we need to stipulate what constitutes a denial of the whole conjunction. For this, it is sufficient to deny any part of the conjunction which is allowed by expressions 19 and 20. These “unpack” conjunctions, searching for input expressions which we might deny, so for denial of any conjunction, in our sense, it is both necessary and sufficient to find within it one deniable input expression.

$$
\begin{aligned}
deny(input(I), D, S_c, S_f, co(I, I1, \mathcal{T})) &\leftarrow conflicting\_outputs(I, I1) \wedge \qquad (18) \\
&\quad\ output(Op, I1, Conditions) \wedge \\
&\quad\ satisfy(Conditions, D, S_c, S_f, \mathcal{T}) \\
deny((A \wedge B), D, S_c, S_f, dc((A, B), A, \mathcal{T})) &\leftarrow deny(A, D, S_c, S_f, \mathcal{T}) \qquad (19) \\
deny((A \wedge B), D, S_c, S_f, dc((A, B), B, \mathcal{T})) &\leftarrow deny(B, D, S_c, S_f, \mathcal{T}) \qquad (20)
\end{aligned}
$$

All that is left is to define which outputs can conflict. Since this relationship is symmetrical (if output A conflicts with output B then B also conflicts with A) we first define the symmetrical property of $conflicting\_outputs$:

$$
\begin{aligned}
conflicting\_outputs(A, B) &\leftarrow conflicting\_output(A, B) \qquad (21) \\
conflicting\_outputs(A, B) &\leftarrow conflicting\_output(B, A) \qquad (22)
\end{aligned}
$$

We then say, for our example, which outputs could conflict. In our example, only the information on the loan risk of the applicant can be conflicting.

$$conflicting\_output(loan\_info(P, good\_risk), loan\_info(P, bad\_risk))$$

Using our new support tree generator ($satisfy1$) instead of the old one ($satisfy$) in our output generation mechanism we can produce support trees in which potential conflicts are identified. A graphical representation of one of these for loan disagreement for $p1$ is shown in Figure 2.8.

Detailed Process Model

This page supplements the analysis of BSDM in Chapter 4 by giving an example of the mechanism for determining which states are accessible in the detailed process model.

A final state, X_f, is accessible from itself, without any process applications or from the current state, X, if there is an applicable process, P; and we make the changes for P which are given in Ch to yield intermediate state X_c; and we remove all the deletions, De, stipulated by P and add all the new information, Or, originated by P to give new state X_n; and X_f is accessible from X_n.

$$
\begin{array}{rcl}
accessible(X_f, X_f, []) & & \\
accessible(X, X_f, [p(P, Or, Re, Ch, De)|H_r]) & \leftarrow & process(P, Or, Re, Ch, De, X) \wedge \\
 & & change(Ch, X, X_c) \wedge \\
 & & X_n = Or \cup (X_c - De) \wedge \\
 & & accessible(X_n, X_f, H_r)
\end{array}
$$

We need to define the processes which will be responsible for moving between states. We shall do this using a standard format in which we give the name of the process goal (always of the form $process(P, Or, Re, Ch, De, X)$ with P being instantiated to one of the process names from the table in Figure 4.3). For each we define the trigger which would allow us to apply the process; the operations which the process performs when triggered; and the changes to each of the components of the process scope (Or, Re, Ch, De). Our three example processes appear in Figures 1, 2 and 3. Notice that the $trigger$, $operation$ and $effects$ conditions for each process could be parcelled together in a single expression defining the whole process:

$$process(P, Or, Re, Ch, De, X) \leftarrow trigger \wedge operation \wedge effects$$

To complete our basic model all we need is a definition of the conditions for assigning grades. These are given formally below.

We give an 'a' grade if both exam score (E_a) and practical score (P_a) are 75 *or over. A 'b' grade is given if both marks are at least* 65 *but at least one is under* 75. *A 'c' grade is given if both marks are at least* 50 *but at least one is under* 65. *A 'd' grade is given if either mark is below* 50.

$$
\begin{array}{rcl}
grade\_assignment(E_a, P_a, a) & \leftarrow & E_a \geq 75 \wedge P_a \geq 75 \\
grade\_assignment(E_a, P_a, b) & \leftarrow & (E_a < 75 \vee P_a < 75) \wedge E_a \geq 65 \wedge P_a \geq 65 \\
grade\_assignment(E_a, P_a, c) & \leftarrow & (E_a < 65 \vee P_a < 65) \wedge E_a \geq 50 \wedge P_a \geq 50 \\
grade\_assignment(E_a, P_a, d) & \leftarrow & E_a < 50 \vee P_a < 50
\end{array}
$$

The process:

$$process(award\_grade(S,C,G), Or, Re, Ch, De, X)$$

denotes that student S has been awarded a grade, G, for course, C. It must satisfy the following trigger conditions and apply the given operations, which will cause the stipulated changes to the process scope:

- Trigger: *We must not have any existing final mark for S in the current state, X; a registration for S for course C must have taken place; and we must know the exam and practical marks for S.*

$$not \left(\begin{array}{l} \exists G', E' P'. \\ attribute(grade, final\_assessment(\begin{array}{l} exam\_mark(S, exam\_assessment(C,E')) \\ practical\_mark(S, practical\_assessment(C,P')) \end{array}), G') \in X \end{array} \right)$$
$$\wedge$$
$$register(S,C) \in X$$
$$\wedge$$
$$attribute(score, exam\_mark(S, exam\_assessment(C,E)), M_e) \in X$$
$$\wedge$$
$$attribute(score, practical\_mark(S, practical\_assessment(C,P)), M_p) \in X$$

- Operation: *We assign a grade G based on the exam mark (M_e) and practical mark (M_p).*

$$grade\_assignment(M_e, M_p, G)$$

- Effects on state: *We originate a final mark for S, having required information about his or her exam and practical marks.*

$$Or = \{attribute(grade, final\_assessment(\begin{array}{l} exam\_mark(S, exam\_assessment(C,P')) \\ practical\_mark(S, practical\_assessment(C,E')) \end{array}), G)\}$$
$$\wedge$$
$$Re = \left\{ \begin{array}{l} register(S,C), \\ attribute(score, exam\_mark(S, exam\_assessment(C,P)), M_e), \\ attribute(score, practical\_mark(S, practical\_assessment(C,E)), M_p) \end{array} \right\}$$
$$\wedge$$
$$Ch = \{\}$$
$$\wedge$$
$$De = \{\}$$

Figure 1: Definition of grade award process

The process:

$$process(revise\_grade(S, C, G, R), Or, Re, Ch, De, X)$$

denotes that the grade for student S has been revised from G to R for course, C. It must satisfy the following trigger conditions and apply the given operations, which will cause the stipulated changes to the process scope:

- Trigger: *We must know the current final grade (G), exam mark (M_e) and practical mark (M_p) in current state, X.*

$$\begin{gathered} attribute(grade, final\_assessment(\begin{array}{l} exam\_mark(S, exam\_assessment(C, E)) \\ practical\_mark(S, practical\_assessment(C, P)) \end{array}), G) \in X \\ \wedge \\ attribute(score, exam\_mark(S, exam\_assessment(C, E)), M_e) \in X \\ \wedge \\ attribute(score, practical\_mark(S, practical\_assessment(C, P)), M_p) \in X \end{gathered}$$

- Operation: *We add 5 to the exam and practical marks and re-calculate the grade, R. It must be different from the current grade, G.*

$$\begin{gathered} E_r = M_e + 5 \\ \wedge \\ P_r = M_p + 5 \\ \wedge \\ grade\_assignment(E_r, P_r, R) \\ \wedge \\ not(R = G) \end{gathered}$$

- Effects on state: *We require information about the exam mark and practical mark for S. We change the grade of the final mark from G in the current state to R in the new state.*

$$\begin{gathered} Or = \{\} \\ \wedge \\ Re = \left\{ \begin{array}{l} attribute(score, exam\_mark(S, exam\_assessment(C, P)), M_e), \\ attribute(score, practical\_mark(S, practical\_assessment(C, E)), M_p) \end{array} \right\} \\ \wedge \\ Ch = \left\{ \begin{array}{l} (attribute(grade, final\_assessment(\begin{array}{l} exam\_mark(S, exam\_assessment(C, E)) \\ practical\_mark(S, practical\_assessment(C, P)) \end{array}), G) \\ attribute(grade, final\_assessment(\begin{array}{l} exam\_mark(S, exam\_assessment(C, E)) \\ practical\_mark(S, practical\_assessment(C, P)) \end{array}), R) \end{array} \right\} \\ \wedge \\ De = \{\} \end{gathered}$$

Figure 2: Definition of grade revision process

The process:

$$process(withdraw(S, C), Or, Re, Ch, De, X)$$

denotes that student S has withdrawn course, C. It must satisfy the following trigger conditions and apply the given operations, which will cause the stipulated changes to the process scope:

- Trigger: *S must have registered for C in the current state, X.*

$$register(S, C) \in X$$

- Operation: *No additional operation information is needed for this process. We assume that all withdrawals are simply accepted.*
- Effects on state: *We require information about the registration of S in the current state and delete that information from the new state.*

$$\begin{aligned} Or &= \{\} \\ Re &= \{register(S, C)\} \\ Ch &= \{\} \\ De &= \{register(S, C)\} \end{aligned}$$

Figure 3: Definition of registration withdrawal process

Further Details of a Basic Scheduling Model

This page supplies the further definitions needed to complete the basic scheduling example in Chapter 4.

Given a number of activities (i.e. a_1, a_2, a_3 and a_4), a number of time periods (i.e. t_1, t_2 and t_3) chronologically ordered ($t_1 \leq t_2 \leq t_3$) and a set of requirements stating the desired ordering and simultaneity of activities, then a schedule is a set of activity-time pairs assigning to each activity exactly one time period, where the assignements must satisfy all the requirements.

We have four types of requirements on the schedule:

- Activities X and Y are *simultaneous.*
- Activities X and Y are *not simultaneous.*
- Activity X must *precede* activity Y.
- Activity X must *not precede* activity Y.

Furthermore, we prefer schedules that assign to each activity a time period that occurs as early as possible. The schedulling process is stated as follows.

We have activities $a1$, $a2$, $a3$ and $a4$.

$$\begin{gathered} activity(a1) \\ activity(a2) \\ activity(a3) \\ activity(a4) \end{gathered}$$

We have time points $t1$, $t2$ and $t3$.

$$\begin{gathered} time\_point(t1) \\ time\_point(t2) \\ time\_point(t3) \end{gathered}$$

The time points are ordered with $t1$ forst, followed by $t2$ and then $t3$.

$$\begin{gathered} before(t1, t2) \\ before(t1, t3) \\ before(t2, t3) \end{gathered}$$

We can refine a schedule, S, to a new schedule formed from S and an activity, A, started at time, T, if A is an activity; T is a time point; and A has not already been assigned in schedule S.

$$refine(S, \{assign(A, T)\} \cup S) \leftarrow activity(A) \wedge time\_point(T) \wedge not(\exists T_1.assign(A, T_1) \in S)$$

A schedule, S, is complete if every activity, A, is assigned a time, T in S.

$$complete(S) \leftarrow \forall A.(activity(A) \rightarrow \exists T.assign(A, T) \in S)$$

Requirements on the schedule are represented as pairs where the first element is a schedule and the second is a predicate which must hold for that schedule. For our example, these are as follows:

For any schedule, S: $a3$ must precede $a2$ in S; $a3$ must precede $a4$ in S; and $a2$ must not occur at the same time as $a4$ in S.

$$
\begin{array}{l}
requirement(S, precedes(a3, a2, S)) \\
requirement(S, precedes(a3, a4, S)) \\
requirement(S, not(simultaneous(a2, a4, S)))
\end{array}
$$

We next define the tests required above.

Activity, A_1, precedes activity, A_2, in schedule, S, if every time point at which A_1 is assigned in S is earlier than any to which A_2 is assigned.

$$precedes(A_1, A_2, S) \leftarrow \forall T_1, T_2((assign(A_1, T_1) \in S \wedge assign(A_2, T_2) \in S) \rightarrow earlier(T_1, T_2))$$

Activity, A_1, is simultaneous with activity, A_2, in schedule, S, if both are assigned to the same time point.

$$simultaneous(A1, A2, S) \leftarrow \forall T_1, T_2((assign(A_1, T_1) \in S \wedge assign(A_2, T_2) \in S) \rightarrow T_1 = T_2)$$

Time point, T_1, is earlier than time point, T_2, if T_1 immediately precedes T_2 or if T_1 immediately precedes some other time point, T_n, which is earlier than T_2.

$$
\begin{array}{lcl}
earlier(T_1, T_2) & \leftarrow & before(T_1, T_2) \\
earlier(T_1, T_2) & \leftarrow & before(T_1, T_n) \wedge earlier(T_n, T_2)
\end{array}
$$

Finally we define what we mean by a correct schedule.

A schedule S is correct *if all requirements R hold for it.*

$$correct(S) \leftarrow \forall R.(requirement(S, R) \rightarrow R)$$

This completes our definition of the scheduling model. That it is a model, rather than a specification of a real scheduler, is clear because it omits many features of practical schedulers (such as an efficient schedule selection algorithm). Nevertheless, it is executable and can generate alternative schedules for our test example by finding solutions for the goal $design([], X)$. Instances generated for X by satisfing this goal from our definitions include:

$$[assign(a4, t3), assign(a3, t1), assign(a2, t2), assign(a1, t1)]$$

A More Complex WWW Site Generation Example

Section 5.1 of the book explains, using a toy example, how parameterisable components can be used to generate a WWW site from a conceptual model of our research group. This page gives a more realistic description of what the problem description language and parameterisable components must represent in order to generate our actual WWW site. For completeness, this example repeats a little of the explanation from the book.

14 Problem Description Language (Research Group)

The conceptual model which we use at time of writing is constructed using fifteen types of assertion. We describe a representative sample of five of these below:

Expression	Interpretation
$group\_name(G)$	G is the name of the group.
$cluster(P, I, T)$	We want an information cluster of type, P, referenced by image, I, and title T.
$person(X, R, S)$	P is the name of a person with WWW personal page reference R and S indicating the degree of attachment to the group (either a *current* member, *previous* member or *friend*).
$group\_photo(I)$	I gives to the location of a photo of the group.
$project(P, T, D, M)$	There is a project, P, with title, T; textual description, D; and associated group members, M.

To use these to generate our WWW site we have defined a collection of expressions relating the expressions in our conceptual model to the structure of our WWW site. Satisfying these expressions generates an instance of our WWW site. Although the structure of the site itself consists of a collection of pages written in the HTML language, we use the translators provided by the Pillow system[1] to permit us a higher level vocabulary, using structured terms, which the Pillow translators automatically convert to HTML. A sample of the structured terms used to represent the form of WWW pages is given below:

[1] These were extended slightly to give us convenient "macros" for some commonly occurring HTML structures

Expression	Interpretation as HTML
$page(P, W_p)$	An HTML page named P with contents W_p.
$start$	The beginning of an HTML page.
end	The end of an HTML page
$head(H)$	A page heading with contents, H.
$title(X)$	A page title with text, X.
$heading(N, X)$	A heading within HTML text at level N with text, X.
$ref(A, S)$	A hypertext link to URL address A via the symbol S, which may be a string of text or an image.
$image(A)$	An image found at URL address, A.
$table(T)$	A table with contents, T.
$itemize(L)$	An itemised list, L, of entries.
nl	A line break.
$center(H)$	The HTML for structure H is centred on the page.

15 Parameterisable Components (Site Pages)

We now show how we defined the connection between conceptual model and HTML structures[2]. The basic idea is to define a "top-level" predicate which generates the entire WWW site description and use its preconditions to construct the smaller site components. An example of a top-level site generator and some smaller components is given below.

We have a site consisting of pages $\mathcal{W}_m \cup \mathcal{W}_p$ if $\mathcal{W}_m$ is the set of main pages, given the set S_m of topic clusters, and $\mathcal{W}_p$ is the set of project pages, given the set S_p of project identifiers. To construct these pages we also need $table(N)$ which contains the hypertext references to each of the topic clusters and is embedded within each page as a convenient means of navigation.

$$\begin{array}{rl} group\_site(\mathcal{W}_m \cup \mathcal{W}_p) \leftarrow & S_m = \{P \mid \exists I', T'.cluster(P, I, T)\} \wedge \\ & S_i = \{ref(P, image(I))) \mid \exists T'.cluster(P, I, T')\} \wedge \\ & S_n = \{ref(P, T)) \mid \exists I'.cluster(P, I, T')\} \wedge \\ & S_p = \{P_p \mid \exists D, T_p, M.project(P, T, D, M)\} \wedge \\ & table([S_i, S_t], N) \wedge \\ & main\_pages(S_m, table(N), \mathcal{W}_m) \wedge \\ & project\_pages(S_p, table(N), \mathcal{W}_p) \end{array} \tag{23}$$

The main pages are the set S_m of pages named P with contents, W_p, for which we can satisfy the relation $P(\mathcal{N}, W_p)$ where $\mathcal{N}$ is the navigator for the page. Thus, if 'people' is one of the elements of S_m and we can satisfy $people(\mathcal{N}, W_p')$ then W_p' is an element of $\mathcal{W}_m$.

$$main\_pages(S_m, \mathcal{N}, \mathcal{W}_m) \leftarrow \mathcal{W}_m = \{page(P, W_p) | P \in S_m \wedge P(\mathcal{N}, W_p)\} \tag{24}$$

[2]In reality we used Prolog Definite Clause Grammars for this task but this is a notational detail which need not concern us here.

This takes us down to the level of generating individual HTML pages. Rather than describing all of these for the site (there are six page templates in total) we describe a characteristic example, which deals with the people in the group. The expression which generates this page from our conceptual model is given below.

A page describing the people in the group is represented by the sequence of HTML structures corresponding to: the page title, H_p; standard heading, H, for group pages; our group photo, I; the navigator, N; another separator; a list of current members, M_c, with heading; a list of past members, M_p with heading; a list of friends of the group, M_f, with heading; a final separator; and a second instance of the navigator.

$$\begin{array}{ll} people(N, W_p) \leftarrow & S_m = \{ref(R, X) \mid \exists R.person(X, R, current)\} \wedge \\ & S_p = \{ref(R, X) \mid \exists R.person(X, R, previous)\} \wedge \\ & S_f = \{ref(R, X) \mid \exists R.person(X, R, friend)\} \wedge \\ & standard\_heading('\texttt{People}', H) \wedge \\ & separator(S) \wedge \\ & group\_photo(I) \wedge \\ & H_p = head(title(`\texttt{SSP People}')) \wedge \\ & H_c = heading(2, `\texttt{Current Members}') \wedge \\ & M_c = itemize(S_m) \wedge \\ & H_p = heading(2, `\texttt{Previous Members}') \wedge \\ & M_p = itemize(S_p) \wedge \\ & H_f = heading(2, `\texttt{Past and current co} - \texttt{workers}') \wedge \\ & M_f = itemize(S_f) \wedge \\ & W_p = [start, H_p, H, I, N, S, H_c, M_c, H_p, M_p, H_f, M_f, S, N, end] \end{array} \tag{25}$$

Our standard heading for pages consists of the group name, G, followed by a separator.

$$\begin{array}{ll} standard\_heading(T, [center(heading(1, G)), center(heading(1, T)), S]) \leftarrow & group\_name(G) \wedge \\ & separator(S) \end{array} \tag{26}$$

A separator consists of an appropriately aligned coloured bar.

$$separator([nl, center(image('\texttt{bar.gif}', [\texttt{align} = \texttt{middle}])), nl]) \tag{27}$$

16 Parameterisation System (Simple Instantiation)

The mechanism for instantiating our definitions attempts to find a solution for the goal *group_site*(X) and, if successful, this instantiates X to a sequence of terms in the Pillow notation. This sequence can then be translated directly to the pages comprising our WWW site.

A More Complex Ecological Model Generation Example

Section 5.2 of the book explains, using a toy example, how parameterisable components can be used to control the generation of a toy "ecological" model of. This page gives an example for a more realistic model, in a similar descriptive style to that of the toy example.

17 Design Generated (Animal Population model)

Our example is an animal population model based on a rectangular spatial grid. The grid has a fixed number of rows and columns and we know the initial population densities for each grid square. We will refer to the sub-populations in each square as population elements. Movement of animals can take place to neighbouring squares (those above, below, left or right of a given square) and this is observed as changes in the density of animals in the squares. The density of animals in a square may also change as a result of births and deaths. We are not interested in which individuals migrate, are born or die - our only interest is in the changes in the numbers of animals in each square (the population density) over time. In general, we are interested in the change of total biomass of the population over time, which is proportional to the total population density (assuming that we know the biomass of an "average" animal and that this multiplied by the density of animals gives a good approximation of the biomass in a unit area). The passage of time is represented using a simple "ticking clock", with the population densities at any given time being calculated from those of the previous "tick". The specification of our model is as follows, with expressions 28 to 40 defining the model and the expressions immediately following 40 defining its initialisation data.

The biomass, N, of the population at time T is calculated by taking the sum of the biomasses, N_s, of all the elements, X, of the population.

$$\begin{aligned} pop\_biomass(T, N) \quad \leftarrow \quad & S = [N_s | \exists X.(in\_pop(X) \wedge biomass(X, T, N_s))] \wedge \\ & sum(S, N) \end{aligned} \tag{28}$$

An element of the population, represented by the term $pop(X_i, Y_j)$ exists for $i \in \{1, \cdots, M_x\}$ and $j \in \{1, \cdots, M_y\}$, where M_x and M_y are the maximum numbered "X" and "Y" coordinate squares on the spatial grid.

$$\begin{aligned} in\_pop(pop((X_i, Y_j))) \quad \leftarrow \quad & max\_x(M_x) \wedge \\ & max\_y(M_y) \wedge \\ & between(1, M_x, X_i) \wedge \\ & between(1, M_y, Y_j) \end{aligned} \tag{29}$$

The biomass, N, of any element, X, of the population at time, T, is calculated from that element's density, D multiplied by a fixed proportion of the average weight, W, of an animal in the population (the proportion being that of dry weight to total weight).

$$\begin{array}{lcl} biomass(X,T,N) & \leftarrow & density(X,T,D) \wedge \\ & & av\_weight(W) \wedge \\ & & N \ is \ D \times 0.1 \times W \end{array} \tag{30}$$

The density, D, of any element, X, of the population at time, T, is either its initial density, if T is the initial time (expression 31), or is calculated from its density, D_p, at the previous time, T_p, given the positive/negative effects of migration, N_m, from nearby elements; the addition of births, N_b; and subtraction of deaths, N_d (expression 32).

$$\begin{array}{lcl} density(X,T,D) & \leftarrow & initial\_time(T) \wedge \\ & & initial\_density(X,D) \end{array} \tag{31}$$

$$\begin{array}{lcl} density(X,T,D) & \leftarrow & not(initial\_time(T)) \wedge \\ & & previous(T,T_p) \wedge \\ & & density(X,T_p,D_p) \wedge \\ & & S = [D_n | \exists X_n . (nearby(X,X_n) \wedge density(X_n,T_p,D_n))] \wedge \\ & & migration(S,D_p,N_m) \wedge \\ & & births(X,D_p,N_b) \wedge \\ & & deaths(X,D_p,N_d) \wedge \\ & & D_1 \ is \ int(D_p + N_m + N_b - N_d) \wedge \\ & & ((D_1 \geq 0 \wedge D = D_1) \vee (D_1 < 0 \wedge D = 0)) \end{array} \tag{32}$$

The number of animals migrating into a population element with density D_p, given a list of the densities of its surrounding elements, is obtained as follows. If the list of surrounding densities is empty (expression 33) then migration is zero. Otherwise, calculate the contribution to migration from the first element, D_n, of the surrounding list as a proportion of the difference between D_n and D_p (a positive value representing an influx into the element and a negative value representing an exodus), and summing this with the total migration to/from the rest of the list.

$$migration([], D_p, 0) \tag{33}$$

$$\begin{array}{lcl} migration([D_n|T], D_p, N_m) & \leftarrow & N \ is \ (D_n - D_p) \times 0.1 \wedge \\ & & migration(T, D_p, N_r) \wedge \\ & & N_m \ is \ N_r + N \end{array} \tag{34}$$

The number of animals born, N_b, in a population element, X, with density, D, is a function of D and the resource limit, R, for the population. This function *will give low numbers of births at low densities, increasing to a maximum at moderate densities, then decreasing to zero at the resource limit.*

$$\begin{aligned} births(X, D, N_b) \quad \leftarrow \quad & resource\_limit(X, R) \wedge \\ & N_1 \; is \; int(D \times (1 - D/R)) \wedge \\ & ((N_1 \leq 0 \wedge N_b = 0) \vee (N_1 > 0 \wedge N_b = N_1)) \end{aligned} \tag{35}$$

The number of deaths, N_d, in a population element, X, with density, D, is a function of D; the resource limit, R, for the population; and a mortality coefficient, K. This function will give low numbers of deaths at low densities, increasing progressively as density increases.

$$\begin{aligned} deaths(X, D, N_d) \quad \leftarrow \quad & resource\_limit(X, R) \wedge \\ & mortality\_coefficient(K) \wedge \\ & N_1 \; is \; int(K \times D \times (D/R)) \wedge \\ & ((N_1 \leq D \wedge N_d = N_1) \vee (N_1 > D \wedge N_d = D)) \end{aligned} \tag{36}$$

We are modelling time as a sequence of integer "ticks" so the previous time point, T_p, from any time, T, is simply the previous integer.

$$previous(T, T_p) \leftarrow not(initial\_time(T)) \wedge T_p \; is \; T - 1 \tag{37}$$

Given a population element of the form $pop((X, Y))$, where X and Y are the coordinates of the grid square for the element, we can find a nearby element of the form $pop((X_n, Y_n))$, where X_n is near to X and Y_n is near to Y.

$$\begin{aligned} nearby(pop((X, Y)), pop((X_n, Y_n))) \quad \leftarrow \quad & max\_x(M_x) \wedge \\ & max\_y(M_y) \wedge \\ & nearby\_coordinate(X, M_x, X_n) \wedge \\ & nearby\_coordinate(Y, M_y, Y_n) \end{aligned} \tag{38}$$

Grid square coordinate C_n is near to coordinate C, given that the maximum coordinate value is M_c if C_n is one above or below the value of C without going outside the coordinate range $1, \cdots, M_c$.

$$nearby\_coordinate(C, M_c, C_n) \quad \leftarrow \quad C < M_c \wedge C_n \; is \; C + 1 \tag{39}$$

$$nearby\_coordinate(C, M_c, C_n) \quad \leftarrow \quad C > 1 \wedge C_n \; is \; C - 1 \tag{40}$$

The initial time is zero.

$$initial\_time(0)$$

The average weight of animals is 10.

$$av\_weight(10)$$

The coefficient for mortality is 0.2.

$$mortality\_coefficient(0.2)$$

The resource limit for all populations is 100.

$$resource\_limit(P, 100)$$

The spatial grid has 5 *rows and* 5 *columns.*

$$\begin{array}{c} max\_x(5) \\ max\_y(5) \end{array}$$

The initial density, D, of population, X, is either stipulated using i_density or it is not stipulated, in which case it is assumed to be zero.

$$\begin{array}{rcl} initial\_density(X, D) & \leftarrow & i\_density(X, D) \\ initial\_density(X, 0) & \leftarrow & not(\exists D.i\_density(X, D)) \end{array}$$

The stipulated initial densities for population elements at particular grid locations are as follows:

$$\begin{array}{c} i\_density(pop((1,1)), 7) \\ i\_density(pop((2,1)), 18) \\ i\_density(pop((1,2)), 25) \\ i\_density(pop((2,2)), 74) \\ i\_density(pop((3,2)), 19) \\ i\_density(pop((2,3)), 31) \\ i\_density(pop((3,3)), 12) \\ i\_density(pop((4,4)), 15) \\ i\_density(pop((5,4)), 9) \\ i\_density(pop((4,5)), 28) \\ i\_density(pop((5,5)), 6) \end{array}$$

We can execute this specification directly using a Prolog interpreter. In particular, we can find out what will be the total biomass, N, of the population at any given time (say time 10) by satisfying the goal:

$$pop\_biomass(10, N)$$

We can also find out the population density, D, for any element X at any time (say time 10) by satisfying the goal:

$$density(X, 10, D)$$

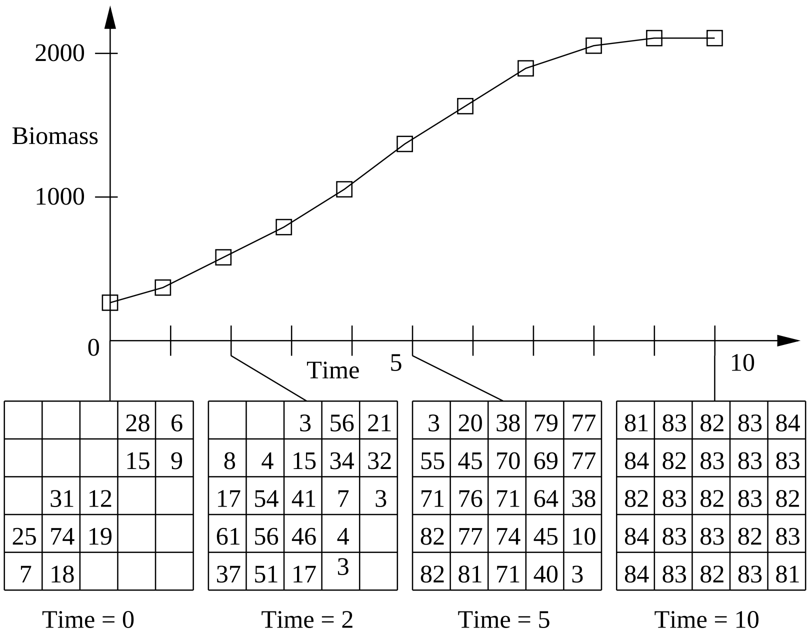


Figure 4: Results of running the example model

Figure 4 shows a plot of the biomass values generated by the model for times 0 to 10. These follow a curve upwards, levelling off to an asymptote, after which the total biomass fluctuates by only small amounts around a constant value. By looking at the changes of densities in grid squares over time - shown by the grids in Figure 4 - we can see why this happens. The first grid shows the initial distribution of animals, with two small clusters of higher density and the majority of the squares empty. By time 2 more of the squares are inhabited by migrations from the more heavily populated ones but the numbers of animals in each square are still low. By time 5 all of the squares are inhabited but the distribution of animals remains uneven. These differences have smoothed out by time 10 and migrations, deaths and births in each square are therefore approximately constant.

18 Parameterisable Components (Model Fragments)

How could we help an ecologist to build a model like this? Not by giving them yet another general-purpose modelling language because these require the modeller to understand exactly which bits of the language to fit together to construct the model. Ecologists do not normally possess this sort of specialist knowledge and few seem highly motivated to acquire it. What often happens in practice is that a specialist in a sub-domain of ecology (animal population dynamics in our example) collaborates with a specialist in particular ecological modelling methods. A skilled modeller is often able to suggest a likely architecture from just a few ecological cues and to follow this up with directed questioning aimed at filling out the details. One way of providing support for this activity is to discern the major architectural features of programs such as the one above and express them in such a way that they can be instantiated to full specifications of models. These are the components which we use in model design. Each has three elements:

- A set of expressions defining the contribution of the component to the specification. These may not be fully defined - they may contain variables which are instantiated to constants, structured terms, predicates or even sets of expressions when the component is applied (see next item).
- A conjunction of constraints determining whether a component is appropriate in a given context and how any variables referring to gaps in the set of expressions (above) will be instantiated. These constraints must be satisfied in order to parameterise the component in constructing the specification.
- A set of the names of predicates which are not defined in the current component but which, if not already defined, need to be supplied by the application of other components in order to complete the specification.

Choosing how to construct components requires the component designer to understand both an appropriate formal language (of which there are many) and the problem domain, so there is a training overhead at this stage in development. In general, designers need to have a feel for those parts of the specification which "belong together" and must then be able to generalise some of the structures in those packages by substituting them with variables. Those variables are then constrained by conditions which guide their instantiation by those in the domain of application.

We now define a set of components sufficient to construct our example model and then describe how these are parameterised to construct our example model.

Component 1 (Population aggregation) *We can define an aggregate attribute of a population in terms of the sum of the attribute which it is aggregating, assuming that we are dealing with a population which is subdivided in some way.*

$$\{P(T,N) \leftarrow S = [N_s | \exists X.(in\_pop(X) \wedge Q(X,T,N_s))] \wedge sum(S,N)\}$$

$$constraint(attribute(Q) \wedge aggregate\_attribute(Q,P) \wedge subdivided\_population)$$

$$needs(\{Q, in\_pop\})$$

Component 2 (Population elements) *We can generate elements of a population according to their X and Y coordinates if the population is spatially subdivided.*

$$\left\{ \begin{array}{lll} in\_pop(pop((X_i, Y_j))) & \leftarrow & max\_x(M_x) \wedge \\ & & max\_y(M_y) \wedge \\ & & between(1, M_x, X_i) \wedge \\ & & between(1, M_y, Y_j) \end{array} \right\}$$

$$constraint(spatially\_subdivided\_population)$$

$$needs(\{max\_x, max\_y\})$$

Component 3 (Biomass from density) *The biomass of a population at any time can be some function of its density at that time, provided we know that that function is.*

$$\{biomass(X, T, N) \leftarrow density(X, T, D) \wedge av\_weight(W) \wedge \mathcal{F}\}$$

$$constraint(function(\{D, W\}, N, \mathcal{F}))$$

$$needs(\{density, av\_weight\})$$

Component 4 (Population density) *The density of a population may be determined as a function of the migrations into or out of it; the births taking place and the number of deaths. This assumes a population spatially distributed on a grid, and that it makes sense to calculate births and deaths at each time step. It also relies on us being able to supply the function determining densities and to stipulate the initial population densities for appropriate grid squares.*

$$
\left\{
\begin{array}{lll}
density(X,T,D) & \leftarrow & initial\_time(T)\wedge \\
 & & initial\_density(X,D), \\
density(X,T,D) & \leftarrow & not(initial\_time(T))\wedge \\
 & & previous(T,T_p)\wedge \\
 & & density(X,T_p,D_p)\wedge \\
 & & S = [D_n | \exists X_n.(nearby(X,X_n) \wedge density(X_n,T_p,D_n))]\wedge \\
 & & migration(S,D_p,N_m)\wedge \\
 & & births(X,D_p,N_b)\wedge \\
 & & deaths(X,D_p,N_d)\wedge \\
 & & \mathcal{F}, \\
nearby(pop((X,Y)),pop((X_n,Y_n))) & \leftarrow & max\_x(M_x)\wedge \\
 & & max\_y(M_y)\wedge \\
 & & nearby\_coordinate(X,M_x,X_n)\wedge \\
 & & nearby\_coordinate(Y,M_y,Y_n), \\
nearby\_coordinate(C,M_c,C_n) & \leftarrow & C < M_c \wedge C_n \text{ is } C+1, \\
nearby\_coordinate(C,M_c,C_n) & \leftarrow & C > 1 \wedge C_n \text{ is } C-1, \\
initial\_density(X,D) & \leftarrow & i\_density(X,D), \\
initial\_density(X,0) & \leftarrow & not(\exists D.i\_density(X,D)), \\
 & & \mathcal{DATA}
\end{array}
\right\}
$$

$$
constraints \left(
\begin{array}{c}
grid\_spatial\_distribution \wedge movement\_between\_squares\wedge \\
births\_each\_time \wedge deaths\_each\_time\wedge \\
function(\{D_p,N_m,N_b,N_d\},D,\mathcal{F})\wedge \\
supply\_densities(M_x,M_y,\mathcal{DATA})
\end{array}
\right)
$$

$$
needs(\{initial\_time, previous, migration, births, deaths, max\_x, max\_y\})
$$

Component 5 (Migration) *We can calculate the total numbers migrating to/from a grid square with a particular population density, given a list of the densities of neighbouring squares, if we can give the function for the migration between each neighbouring square and the central one. This assumes that we have a grid square representation and that migration rates can be determined from the cumulative effect of migrations between each pair of squares.*

$$
\left\{
\begin{array}{lll}
migration([],D_p,0) & & \\
migration([D_n|T],D_p,N_m) & \leftarrow & \mathcal{F} \wedge migration(T,D_p,N_r)\wedge \\
 & & N_m \text{ is } N_r + N
\end{array}
\right\}
$$

$$
constraints \left(
\begin{array}{c}
grid\_spatial\_distribution\wedge \\
independent\_movement\_between\_squares\wedge \\
function(\{D_n,D_p\},N,\mathcal{F})
\end{array}
\right)
$$

$$
needs(\{\})
$$

Component 6 (Births) *The number of births in a population can be determined from its density if we know the resource limit on the size of the population and there births depend on some function of the current density and that limit. This assumes that births are simply density dependent, with no other influences involved.*

$$\{births(X, D, N_b) \leftarrow resource\_limit(X, R) \wedge \mathcal{F}\}$$

$$constraint(simple\_density\_dependent\_births \wedge function(\{D, R\}, N_b, \mathcal{F}))$$

$$needs(\{resource\_limit\})$$

Component 7 (Deaths) *The number of deaths in a population can be determined from its density if we know the resource limit on the size of the population and the coefficient determining the proportion of animals who die each time. Deaths depend on some function of the current density; the resource limit and the coefficient. This assumes that births are simply density dependent, with no other influences involved.*

$$\{deaths(X, D, N_d) \leftarrow resource\_limit(X, R) \wedge mortality\_coefficient(K) \wedge \mathcal{F}\}$$

$$constraint(simple\_density\_dependent\_deaths \wedge function(\{D, R, K\}, N_d, \mathcal{F}))$$

$$needs(\{resource\_limit, mortality\_coefficient\})$$

Component 8 (Time step) *The previous time point is calculated as a function of the current time, assuming a "ticking clock" model of time.*

$$\{previous(T, T_p) \leftarrow not(initial\_time(T)) \wedge \mathcal{F}\}$$

$$constraint(ticking\_clock\_time\_model \wedge function(\{T\}, T_p, \mathcal{F}))$$

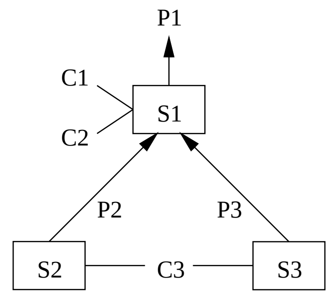
$$needs(\{\})$$

19 Problem Description Language (Ecological Conditions)

In this case, the problem description consists of exactly those predicates required by the *constraint* part of each component. For instance, if we know that the problem requires simple density dependent births and we also know the function relating population density and the resource limit to births then we can select and parameterise component 6.

20 Parameterisation System (Constrained Generation)

Figure 5 illustrates how the parameterisation process works. The diagram on the left gives a typical example of the relationships between the data structures



System engineer supplies:

Model components <S1, {C1,C2}, {P1, P2}>
<S2, {C3}, {}>
<S3, {C3}, {}>

Component selection conditions (C1, C2, C3).

Component needs (P2,P3)

Ecologist Controls:

Top level predicate name (P1).

Instantiation of conditions (C1, C2, C3).

System controls:

Choice of other predicate names (P2, P3).

Instantiation of components (S1, S2, S3).

Figure 5: Illustration of component parameterisation for model generation

involved, whilst the items on the right show where these come from and the division between automated and human responsibilities when using the method. Each component is designed to supply the definitions needed to satisfy a particular predicate (*e.g.* to calculate biomass) and there may be numerous alternative components for any given predicate (*e.g.* there are many standard ways to calculate biomass in a population dynamics model). Model construction is driven by the introduction of a "top level" predicate, P1 in the illustration, which is to be the main output of the model. The parameterisation system finds those components which can construct a definition for P1 and chooses which is appropriate by prompting the ecologist for the conditions associated with it - in the illustration conditions C1 and C2 have been satisfied (and instantiated by the ecologist) in order to select and instantiate component S1. The use of S1 introduces a need to define predicates P2 and P3, which are used in the definition of S1 but not defined directly within that component. These are defined by S2 and S3 respectively, with the ecologist confirming and instantiating condition C3 which is a common prerequisite for both components. The process terminates when we have components providing definitions for every predicate that we have introduced. In doing this most of the complexity of the components is hidden from the ecologist, who controls only the choice of top level predicate and the instantiation of conditions used in selecting components.

To demonstrate how this works for our example (although it omits the details of the dialogue between ecologist and model generator) we use the diagrammatic representation which we introduced in Figure 5, extended to the example model. This is shown in Figure 6. Our top level predicate is $pop\_biomass$ and this is defined using Component 1, with the ecologist having supplied the information that the attribute we are concerned with is biomass (condition $attribute(biomass)$ for Component 1) and that this is aggregated into an attribute describing the population biomass (condition $aggregate\_attribute(biomass, pop\_biomass)$

for Component 1). This introduces the need to define two new predicates: *biomass* and *in_pop*. We leave to the reader the task of following down through the chain of parameterisations used in Figure 6 to define these and the other predicates we introduced during model generation. Notice that to save space we haven't shown all the details of the functions used to instantiate conditions of the form $function(I, O, \mathcal{F})$, where I is the set of input variables; O is the output variable; and $\mathcal{F}$ is the arithmetic function relating inputs to outputs. In the diagram we show all instances of $\mathcal{F}$ as $(\cdots)$ to denote that the actual instantiation has been omitted. You can reconstruct the actual instantiation by comparing the appropriate component to the predicate definition which it generates. For example, Component 3 is used to define *biomass* given the conditions that the average weight is 10 (written as $av\_weight(10)$) and that there is a function relating variables $\{D, W\}$ to variable N. Although not shown on the diagram, the arithmetic function is: $(N\ is\ D \times 01 \times W)$, as we can see by looking at the generated definition.

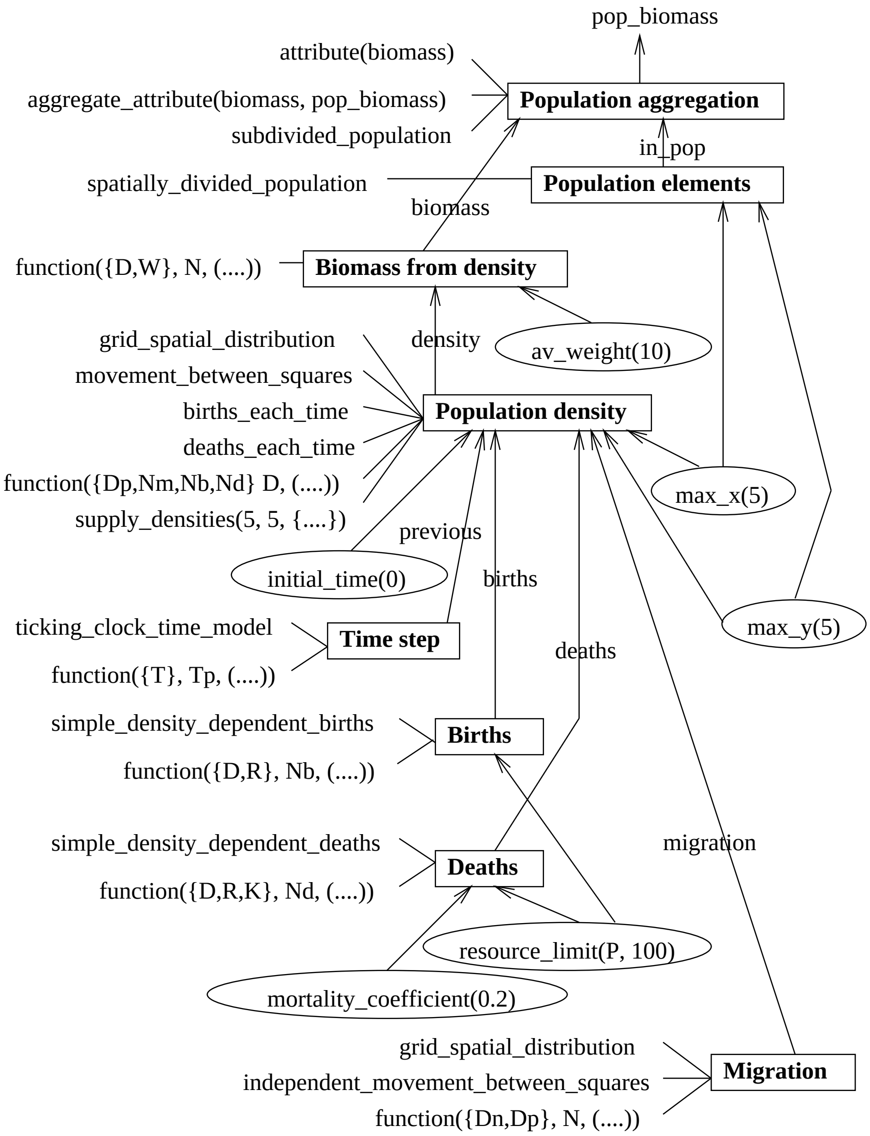


Figure 6: Map of component parameterisation for our example

A Sample of Typical Skeletons

This page gives additional definitions of skeletons which complement those of Chapter 6 of the book. It also supplies some definitions of predicates used in the skeletons and additions of Chapter 6.

Traversing a list recurses through each of its elements, H, an applies some test, $\mathcal{T}$, to it. The base case is an empty list, [], allowing our search to end when we have run out of elements.

$$list\_traversal(P) \triangleright \left\{ \begin{array}{l} P([]), \\ P([\mathtt{H}|\mathtt{T}]) \leftarrow \mathcal{T}(\{\mathtt{H}\}) \wedge P(\mathtt{T}) \end{array} \right\} \tag{41}$$

Decrementing a counter recurses through a sequence of positive integer values from X down to 0.

$$decrementing\_counter(P) \triangleright \left\{ \begin{array}{l} P(\mathtt{0}), \\ P(\mathtt{X}) \leftarrow \mathtt{Y\ is\ X} - \mathtt{1} \wedge P(\mathtt{Y}) \end{array} \right\} \tag{42}$$

A recursive deconstruction of a binary term either succeeds by applying some test $\mathcal{T}_1$ to the term C; or (if the term is composed from functor, F, with arguments A and B) it recurses on A and B; or it obtains some new term, T, from C and recurses on T.

$$recursive\_deconstruction(P) \triangleright \left\{ \begin{array}{l} P(C) \leftarrow \mathcal{T}_1(C), \\ P(F(A,B)) \leftarrow P(A) \wedge P(B), \\ P(C) \leftarrow \mathcal{U}(\{C\},T) \wedge P(T) \end{array} \right\} \tag{43}$$

$carrier(P,C,C',\mathtt{V})$ succeeds if C' is the term constructed by replacing in C all instances of $P(A)$ with $P(A,\mathtt{V})$.

$$\begin{array}{rcll} carrier(P,(A \wedge B),(A' \wedge B'),\mathtt{V}) & \leftarrow & carrier(P,A,A',\mathtt{V}) \wedge & \\ & & carrier(P,B,B',\mathtt{V}) & (44) \\ carrier(P,P(A),P(A,\mathtt{V}),\mathtt{V}) & & & (45) \\ carrier(P,G,G,\mathtt{V}) & \leftarrow & not(G = (A,B) or G = P(A)) & (46) \end{array}$$

$partials(P,B,B',A_s)$ constructs the conjunction of terms, B', from B by replacing each term of the form $P(A)$ in B with one of the form $P(A,V)$, where V is a new variable. A_s is the set of all such variables.

$$\begin{array}{rcll} partials(P,(A \wedge B),(A' \wedge B'),A_s) & \leftarrow & partials(P,A,A',A_1) \wedge & \\ & & partials(P,B,B',A_2) \wedge & \\ & & A_s = A_1 \cup A_2 & (47) \\ partials(P,P(A),P(A,\mathtt{V}),\{\mathtt{V}\}) & & & (48) \\ partials(P,G,G,\{\}) & \leftarrow & not(G = (A \wedge B) \vee G = P(A)) & (49) \end{array}$$

A Larger Example of Techniques Editing

This page gives a larger example of the form of predicate construction used in Chapter 6. We have chosen the $satisfy$ predicate which we used at WWW page 2.3 to help us explore the behaviour of an operator model. To do the example we need an extra form of addition, as described below.

accumulator_pair(P) *adds a pair of arguments,* S *and* F*, to a predicate named* P*. The purpose of* S *is to accumulate information on the way down through the recursion, with* F *being used to "pass back" the final instantiation of* S*. The first mapping is the recursive case which - via the* $pairs$ *condition - ensures that the accumulator is passed through each of the recursive goals (in order) in the condition,* B*. For example, if* B *contains the sequence* $\{P(A_1), P(A_2)\}$ *then these are replaced by* $\{\mathcal{U}_1(\{S\}, S_1) \wedge P(A_1, S_1, S_2), \mathcal{U}_1(\{S_2\}, S_3) \wedge P(A_2, S_3, F)\}$*. The second mapping takes a non-recursive expression and adds the two arguments* S *and* F*, while noting that* F *should be obtained from* S*.*

$$accumulator\_pair(P) \triangleleft \left\{ \begin{array}{ccl} (P(A) \leftarrow B & \mapsto & P(A, \mathtt{S}, \mathtt{F}) \leftarrow B', \mathcal{U}(\{\mathtt{N}\}, \mathtt{F}))\ if\ pairs(P, B, B', \mathtt{S}, \mathtt{N}) \\ P(A) & \mapsto & P(A, \mathtt{S}, \mathtt{F}) \leftarrow \mathcal{U}(\{\mathtt{S}\}, \mathtt{F}) \end{array} \right\} \tag{50}$$

$pairs(P, B, B', V_1, V_f)$ *constructs the conjunction of terms,* B'*, from* B *by replacing the sequence of recursive subgoals,* $\{P(A_1), \ldots, P(A_n)\}$ *in* B *by the new sequence* $\{\mathcal{U}(\{V_1\}, V_2) \wedge P(A_1, V_2, V_3), \ldots, \{\mathcal{U}(\{V_m\}, V_{m+1}) \wedge P(A_n, V_{m+1}, V_f)\}$ *in corresponding positions in* B'*.*

$$\begin{array}{rcl} pairs(P, (A \wedge B), (A' \wedge B'), \mathtt{S}, \mathtt{F}) & \leftarrow & pairs(P, A, A', \mathtt{S}, \mathtt{N}) \wedge \\ & & pairs(P, B, B', \mathtt{N}, \mathtt{F}) \qquad (51) \\ pairs(P, P(A), (\mathcal{U}(\{\mathtt{S}\}, \mathtt{N}), P(A, \mathtt{N}, \mathtt{F})), \mathtt{S}, \mathtt{F}) & & \qquad (52) \\ pairs(P, G, G, \mathtt{S}, \mathtt{S}) & \leftarrow & not(G = (A \wedge B) or G = P(A)) \qquad (53) \end{array}$$

We start by instantiating the skeleton $recursive\_deconstruction(satisfy)$ (defined at WWW page 6.1) to give:

$$\begin{array}{rcl} satisfy(C) & \leftarrow & \mathcal{T}_1(\{C\}) \\ satisfy(F(A, B)) & \leftarrow & satisfy(A, D) \wedge \\ & & satisfy(B) \\ satisfy(C) & \leftarrow & \mathcal{U}(\{C\}, P) \wedge \\ & & satisfy(P) \end{array}$$

We know that $satisfy$ has to unpack conjunctions of terms so we instantiate F to the conjunction symbol, '$\wedge$'.

$$
\begin{array}{rcl}
satisfy(C) & \leftarrow & \mathcal{T}_1(\{C\}) \\
satisfy((A \wedge B)) & \leftarrow & satisfy(A, D) \wedge \\
 & & satisfy(B) \\
satisfy(C) & \leftarrow & \mathcal{U}(\{C\}, P) \wedge \\
 & & satisfy(P)
\end{array}
$$

There are two different base cases for $satisfy$: when C is an input or when C is a system defined goal. Our skeleton currently has only one base case so we replicate the expression $satisfy(C) \leftarrow \mathcal{T}_1(\{C\})$ to give $satisfy(C) \leftarrow \mathcal{T}_2(\{C\})$.

$$
\begin{array}{rcl}
satisfy(C) & \leftarrow & \mathcal{T}_1(\{C\}) \\
satisfy((A \wedge B)) & \leftarrow & satisfy(A) \wedge \\
 & & satisfy(B) \\
\underline{satisfy(C)} & \underline{\leftarrow} & \underline{\mathcal{T}_2(\{C\})} \\
satisfy(C) & \leftarrow & \mathcal{U}(\{C\}, P) \wedge \\
 & & satisfy(P)
\end{array}
$$

We then make the following instantiations:

- The first base case succeeds when C is an input for which we can find a potential output so $\mathcal{T}_1(\{C\})$ is instantiated to:
  $C = input(X) \wedge p\_output(Op, X, D, S_c, S_r, T)$

- The second base case succeeds when C is a non-conjunctive term which is system defined so $\mathcal{T}_2(\{C\})$ is instantiated to:
  $not(C = (A \wedge B)) \wedge system\_defined(C)$

- The means of obtaining P from C in the final recursive case is to test that C is not system defined, a conjunction or an input and that there is a clause determining C from P. This is expressed by instantiating $\mathcal{U}(\{C\}, P)$ to:
  $(not(system\_defined(C)) \wedge not(C = (A \wedge B))) \wedge not(C = input(X)) \wedge clause(C, P))$

$$
\begin{array}{rcl}
satisfy(\underline{input(X)}) & \leftarrow & \underline{p\_output(Op, X, D, S_c, S_r, T)} \\
satisfy((A \wedge B)) & \leftarrow & satisfy(A) \wedge \\
 & & satisfy(B) \\
satisfy(C) & \leftarrow & \underline{not(C = (A \wedge B))} \wedge \\
 & & \underline{system\_defined(C)} \wedge \\
 & & \underline{C} \\
satisfy(C) & \leftarrow & \underline{not(system\_defined(C))} \wedge \\
 & & \underline{not(C = (A \wedge B))} \wedge \\
 & & \underline{not(C = input(X))} \wedge \\
 & & \underline{clause(C, P)} \wedge \\
 & & satisfy(P)
\end{array}
$$

This completes the flow of control for $satisfy$, which is determined solely by the first argument and associated conditions. Turning to the second argument, this is simply a number giving the depth limit, which needs to be carried through the search for solutions. We apply the addition $carry\_through(satisfy)$ (defined in Chapter 6 of the book):

$$
\begin{array}{rcl}
satisfy(input(X), \underline{D}) & \leftarrow & p\_output(Op, X, D, S_c, S_r, T) \\
satisfy((A \wedge B), \underline{D}) & \leftarrow & satisfy(A, \underline{D}) \wedge \\
 & & satisfy(B, \underline{D}) \\
satisfy(C, \underline{D}) & \leftarrow & not(C = (A \wedge B)) \wedge \\
 & & system\_defined(C) \wedge \\
 & & C \\
satisfy(C, \underline{D}) & \leftarrow & not(system\_defined(C)) \wedge \\
 & & not(C = (A \wedge B)) \wedge \\
 & & not(C = input(X)) \wedge \\
 & & clause(C, P) \wedge \\
 & & satisfy(P, \underline{D})
\end{array}
$$

The third and fourth arguments of $satisfy$ are used to carry in the current input streams (via the third argument) and pass back the revised set of streams (via the fourth argument) after processing has occurred. This can be viewed as a form of paired accumulator so we apply the addition $accumulator\_pair(satisfy)$ (defined above):

$$
\begin{array}{rcl}
satisfy(input(X), D, \underline{S_c, S_f}) & \leftarrow & p\_output(Op, X, D, S_c, S_r, T) \\
 & & \underline{\mathcal{U}_1(\{S_c\}, S_f)} \\
satisfy((A \wedge B), D, \underline{S_c, S_f}) & \leftarrow & \underline{\mathcal{U}_2(\{S_c\}, S_1)} \wedge \\
 & & satisfy(A, D, \underline{S_1, S_2}) \wedge \\
 & & \underline{\mathcal{U}_3(\{S_2\}, S_3)} \wedge \\
 & & satisfy(B, D, \underline{S_3, S_4}) \\
 & & \underline{\mathcal{U}_4(\{S_4\}, S_f)} \\
satisfy(C, D, \underline{S_c, S_f}) & \leftarrow & not(C = (A \wedge B)) \wedge \\
 & & system\_defined(C) \wedge \\
 & & C \wedge \\
 & & \underline{\mathcal{U}_5(\{S_c\}, S_f)} \\
satisfy(C, D, \underline{S_c, S_f}) & \leftarrow & not(system\_defined(C)) \wedge \\
 & & not(C = (A \wedge B)) \wedge \\
 & & not(C = input(X)) \wedge \\
 & & clause(C, P) \wedge \\
 & & \underline{\mathcal{U}_6(\{S_c\}, S_1)} \wedge \\
 & & satisfy(P, D, \underline{S_1, S_2}) \\
 & & \underline{\mathcal{U}_7(\{S_2\}, S_f)}
\end{array}
$$

We now have a lot of loose ends to tie, thanks to the updates introduced by the paired accumulator. All of these, except for the first, simply require the variable we need to be identified with the variable from which the update says it should be derived. The exception to this is $\mathcal{U}_1$ because this applies to the case where we have applied an operator and need to add it to the set of those which we are accumulating. The instantiations are:

- $\mathcal{U}_1(\{S_c\}, S_f)$ is instantiated to $(S_f = [s(Op, X)|S_r])$
- $\mathcal{U}_2(\{S_c\}, S_1)$ is instantiated to $(S_1 = S_c)$
- $\mathcal{U}_3(\{S_2\}, S_3)$ is instantiated to $(S_3 = S_2)$
- $\mathcal{U}_4(\{S_4\}, S_f)$ is instantiated to $(S_f = S_4)$
- $\mathcal{U}_5(\{S_c\}, S_f)$ is instantiated to $(S_f = S_c)$
- $\mathcal{U}_6(\{S_c\}, S_1)$ is instantiated to $(S_1 = S_c)$
- $\mathcal{U}_7(\{S_2\}, S_f)$ is instantiated to $(S_f = S_2)$

$$
\begin{array}{rcl}
satisfy(input(X), D, S_c, \underline{[s(Op, X)|S_r]}) & \leftarrow & p\_output(Op, X, D, S_c, S_r, T) \\
satisfy((A \wedge B), D, S_c, S_f) & \leftarrow & satisfy(A, D, \underline{S_c, S_n}) \wedge \\
 & & satisfy(B, D, \underline{S_n, S_f}) \\
satisfy(C, D, S_c, \underline{S_c}) & \leftarrow & not(C = (A \wedge B)) \wedge \\
 & & system\_defined(C) \wedge \\
 & & C \\
satisfy(C, D, S_c, S_f) & \leftarrow & not(system\_defined(C)) \wedge \\
 & & not(C = (A \wedge B)) \wedge \\
 & & not(C = input(X)) \wedge \\
 & & clause(C, P) \wedge \\
 & & satisfy(P, D, \underline{S_c, S_f})
\end{array}
$$

The fifth, and last, argument contains the support tree which is built to record of how the conjunction in the first argument was satisfied. For this we use a single argument accumulator by applying the addition $accumulator(satisfy)$ (defined in Chapter 6 of the book):

$$
\begin{array}{rcl}
satisfy(input(X), D, S_c, [s(Op, X)|S_r], \underline{T}) & \leftarrow & p\_output(Op, X, D, S_c, S_r, T) \wedge \\
 & & \underline{\mathcal{U}_1(\{\}, T)} \\
satisfy((A \wedge B), D, S_c, S_f, \underline{T}) & \leftarrow & satisfy(A, D, S_c, S_n, \underline{T_a}) \wedge \\
 & & satisfy(B, D, S_n, S_f, \underline{T_b}) \wedge \\
 & & \underline{\mathcal{U}_2(\{T_a, T_b\}, T)} \\
satisfy(C, D, S_c, S_c, \underline{T}) & \leftarrow & not(C = (A \wedge B)) \wedge \\
 & & system\_defined(C) \wedge \\
 & & C \wedge \\
 & & \underline{\mathcal{U}_3(\{\}, T)} \\
satisfy(C, D, S_c, S_f, \underline{T}) & \leftarrow & not(system\_defined(C)) \wedge \\
 & & not(C = (A \wedge B)) \wedge \\
 & & not(C = input(X)) \wedge \\
 & & clause(C, P) \wedge \\
 & & satisfy(P, D, S_c, S_f, \underline{T_p}) \wedge \\
 & & \underline{\mathcal{U}_4(\{T_p\}, T)}
\end{array}
$$

Finally, we make the instantiations:

- $\mathcal{U}_1(\{\}, T) = true$
- $\mathcal{U}_2(\{T_a, T_b\}, T) = (T = and(T_a, T_b))$
- $\mathcal{U}_3(\{\}, T) = true$
- $\mathcal{U}_4(\{T_p\}, T) = (T = imp(C, T_p))$

$$
\begin{array}{rcl}
satisfy(input(X), D, S_c, [s(Op, X)|S_r], T) & \leftarrow & p\_output(Op, X, D, S_c, S_r, T) \\
satisfy((A \wedge B), D, S_c, S_f, \underline{and(T_a, T_b)}) & \leftarrow & satisfy(A, D, S_c, S_n, T_a) \wedge \\
 & & satisfy(B, D, S_n, S_f, T_b) \\
satisfy(C, D, S_c, S_c, \underline{C}) & \leftarrow & not(C = (A \wedge B)) \wedge \\
 & & system\_defined(C) \wedge \\
 & & C \\
satisfy(C, D, S_c, S_f, \underline{imp(C, T_p)}) & \leftarrow & not(system\_defined(C)) \wedge \\
 & & not(C = (A \wedge B)) \wedge \\
 & & not(C = input(X)) \wedge \\
 & & clause(C, P) \wedge \\
 & & satisfy(P, D, S_c, S_f, T_p)
\end{array}
$$

An example of Stripping Structure from a Predicate

This page describes a basic (and sometimes inaccurate) way of stripping structures from predicates. We can strip argument N from the definition of a target predicate named P by performing, in the order given, the following transformations to each of its clauses:

- Rewrite every unit term in the clause of the form $P(A_1, \ldots, A_N, \ldots, A_F)$ to $P(A_N)$.
- Rewrite every structured term inside a unit term, of the form $F(A_1, \ldots, A_M)$ to $V(A_1, \ldots, A_M)$, where V is a new variable not appearing in the clause.
- Remove any unit goals from the clause which refer only to variables which do not appear elsewhere in that clause (these are referred to as singleton variables).

We can match a skeletal definition with the definition generated by stripping an argument from a target predicate if for every clause in the skeletal definition there is a corresponding clause in the stripped definition such that the clauses match, and no two clauses in the skeletal clause are matched to the same clause in the stripped definition. Two clauses match if:

- Their heads are unifiable and
- For every unit goal in the skeletal clause we remove a subsequence of unit goals from the stripped clause such that:
  - If the skeletal goal is a generic test or update goal (the $\mathcal{T}$ and $\mathcal{U}$ predicates of book Chapter 6) then we remove a sequence of one or more unit goals from the stripped clause.
  - If the skeletal goal is not a generic test or update goal then we remove a single unit goal from the stripped clause.
- Having removed unit goals from the body of the stripped clause in this way, we should be left with no unmatched unit goals in the body of that clause.
- None of the matches should have instantiated variables in the in the target clause to non-variable terms in the skeletal clause, although it is allowed for variables in the skeletal clause to be instantiated to terms in the target clause. This means, for example, that a skeletal term $p(f(A))$ is not allowed to match with a target term $p(X)$, since this would instantiate X to $f(A)$. However, the match is permitted is the skeletal term is $p(X)$ and the target term is $p(f(A))$.

Let us apply this method to the example we gave in Chapter 6 of the book. This time we assume that our "case library" is simply a file of predicate definitions, of which one is the *satisfy* predicate defined at WWW page 6.2. Suppose that, in our techniques editor, we have selected the recursive deconstruction skeleton and have instantiated it in a similar way to an early stage of the example at WWW page 6.2 (although without committing to the predicate name, which we leave as the variable, P). Our skeleton is:

$$\begin{array}{rcl} P(C) & \leftarrow & \mathcal{T}_1(\{C\}) \\ P((A \wedge B)) & \leftarrow & P(A) \wedge P(B) \\ P(C) & \leftarrow & \mathcal{T}_2(\{C\}) \\ P(C) & \leftarrow & \mathcal{U}(\{C\}, P) \wedge P(P) \end{array}$$

We now want to find whether there is an existing predicate definition which might be relevant to this skeleton so we apply our method for stripping arguments to our definition of *satisfy*. This yields five sets of clauses (one for each argument in *satisfy*), which we give below, in the order of the argument position from which they were stripped.

$$\begin{array}{rcl} satisfy(input(X)) & \leftarrow & p\_output(Op, X, D, S_c, S_r, \mathcal{T}) \\ satisfy((A \wedge B)) & \leftarrow & satisfy(A) \wedge satisfy(B) \\ satisfy(C) & \leftarrow & not(C = (A \wedge B)) \wedge system\_defined(C) \wedge C \\ satisfy(C) & \leftarrow & \quad not(system\_defined(C)) \wedge not(C = (A \wedge B)) \wedge \\ & & \quad not(C = input(X)) \wedge clause(C, P) \wedge satisfy(P) \end{array} \tag{54}$$

$$\begin{array}{rcl} satisfy(D) & \leftarrow & p\_output(Op, X, D, S_c, S_r, \mathcal{T}) \\ satisfy(D) & \leftarrow & satisfy(D) \wedge satisfy(D) \\ satisfy(D) & & \\ satisfy(D) & \leftarrow & satisfy(D) \end{array} \tag{55}$$

$$\begin{array}{rcl} satisfy(S_c) & \leftarrow & p\_output(Op, X, D, S_c, S_r, \mathcal{T}) \\ satisfy(S_c) & \leftarrow & satisfy(S_n) \\ satisfy(S_c) & & \\ satisfy(S_c) & \leftarrow & satisfy(S_c) \end{array} \tag{56}$$

$$\begin{array}{rcl} satisfy([s(Op, X)|S_r]) & \leftarrow & p\_output(Op, X, D, S_c, S_r, \mathcal{T}) \\ satisfy(S_f) & \leftarrow & satisfy(S_n) \wedge satisfy(S_f) \\ satisfy(S_c) & & \\ satisfy(S_f) & \leftarrow & satisfy(S_f) \end{array} \tag{57}$$

$$\begin{array}{rcl} satisfy(\mathcal{T}) & \leftarrow & p\_output(Op, X, D, S_c, S_r, \mathcal{T}) \\ satisfy(and(\mathcal{T}_a, \mathcal{T}_b)) & \leftarrow & satisfy(\mathcal{T}_a) \wedge satisfy(\mathcal{T}_b) \\ satisfy(C) & \leftarrow & not(C = (A \wedge B)) \wedge system\_defined(C) \wedge C \\ satisfy(imp(C, \mathcal{T}_p)) & \leftarrow & \quad not(system\_defined(C)) \wedge not(C = (A \wedge B)) \wedge \\ & & \quad not(C = input(X)) \wedge clause(C, P) \wedge satisfy(P) \end{array} \tag{58}$$

If we now apply our matching method to these five candidates we obtain only one match, which is between the skeleton and candidate 54. The correspondence of unit goals in this match is shown in the table below, where the first column gives the number of the clauses which match[3], the second column shows the matching term in the skeleton and the third column shows the matching term in the target.

Clause	Skeleton	Target
1	$P(C)$	$satisfy(input(X))$
	$\mathcal{T}_1(\{C\})$	$p\_output(Op, X, D, S_c, S_r, \mathcal{T})$
2	$P((A \wedge B))$	$satisfy((A \wedge B))$
	$P(A)$	$satisfy(A)$
	$P(B)$	$satisfy(B)$
3	$P(C)$	$satisfy(C)$
	$\mathcal{T}_2(\{C\})$	$not(system\_defined(C)) \wedge not(C = (A \wedge B)) \wedge$ $not(C = input(X)) \wedge clause(C, P) \wedge satisfy(P)$
4	$P(C)$	$satisfy(C)$
	$\mathcal{U}(\{C\}, P)$	$not(system\_defined(C)) \wedge not(C = (A \wedge B)) \wedge$ $not(C = input(X)) \wedge clause(C, P)$
	$P(P)$	$satisfy(P)$

None of the other candidates match. There are various reasons why each could be rejected but the most immediate one is that none of the heads of clauses in the other candidates can be unified with the head of the second clause of the skeleton without binding a variable in the candidate to a term from the skeleton. For instance, the head of every clause in candidate 55 is $satisfy(D)$ and if we unify this with $P((A \wedge B))$ then the variable D is bound to the term $(A \wedge B)$, which contravenes the restriction on unification placed on our matching method.

You may have noticed that the single match we obtained (candidate 54) is exactly the predicate which we obtained in Chapter 6 of the book by undoing some of the additions made when constructing the original definition of $satisfy$ by techniques editing. Therefore, we could complete our current example by taking candidate 55 and adapting it as shown in the book.

[3]It happens in this case that the same clause numbers match in skeleton and target although this isn't necessary in general.

A More General View of Endorsements

The mechanism for propagating uncertainty which we described in Chapter 8 of the book propagates the intersections and unions of endorsement sets, with it being left to modellers to judge how this information influences their assessment of the model. In some cases we may wish to provide more sophisticated propagation methods which reduce the volume of information presented to those assessing the model. In this section we discuss briefly how this may be done, using an example adapted from Sullivan & Cohen's paper at IJCAI'85.

In user modelling systems it is common to have some representation (hidden from the user) of standard plans of action. By comparing user input to these stored plans, the user modelling system attempts to guess which particular plan the user is most likely to be following. This allows the system to provide the user with advice about how best to achieve the hypothesised plan. Clearly this matching process involves considerable uncertainty, since initial user input is unlikely to be able to determine precisely what the correct plan should be. Plan recognition systems for complex real–world tasks need to use large numbers of sophisticated plans but, to keep our example simple, imagine that we have a rudimentary planning system in which only two linear plans are allowed to be used. Each of these is described using the predicate $plan(P, S)$, where P is the name of the plan and S is the sequence of names of the tasks which it involves.

Plan 1 involves the sequence of tasks 'a' then 'b' then 'c'. Plan 2 involves the sequence of tasks 'b' then 'd' then 'e'.

$$plan(plan1, [a, b, c]) \tag{59}$$
$$plan(plan2, [b, d, e]) \tag{60}$$
$$\tag{61}$$

Input from the user will consist of selecting a particular sequence of tasks and we want to guess, at each point in the input sequence, which plan the user has in mind. Our classification of the plan at any given stage in the sequence may be uncertain for any of a number of reasons. To encode these reasons we use endorsements which we define as rules of the form:

$$endorsement(P, A, S_p, E, W) \leftarrow C$$

where P is the identifier of the plan to which the endorsement applies; A is the action to which it applies; S_p is the sequence of actions which have taken place prior to action A; E is a term describing the endorsement; W is the weight of the endorsement, which can be either positive or negative; and C is the condition of the endorsement rule. In our example, we use five of these endorsement rules (labelled 62 to 66 below).

An observed task, A, could be a mistake if it appears in the sequence, S, of any plan, P.

$$endorsement(P, A, S_p, could\_be\_a\_mistake(A), neg) \leftarrow plan(P, S) \wedge A \in S \tag{62}$$

Plan P_1 is the only grammatical possibility given observed action, A, if there is no other plan containing A.

$$endorsement(P_1, A, S_p, only\_grammatical\_possibility(A), pos) \leftarrow plan(P_1, S_1) \wedge A \in S_1 \wedge not(plan(P_2, S_2) \wedge not(P_1 = P_2) \wedge A \in S_2) \tag{63}$$

There is an alternative plan, P_2, to plan P_1 for task A if both plans involve A. This weakens our certainty that plan P_1 is the one we are observing.

$$endorsement(P_1, A, S_p, other\_grammatical\_possibility(A), neg) \leftarrow plan(P_1, S_1) \wedge plan(P_2, S_2) \wedge not(P_1 = P_2) \wedge A \in S_1 \wedge A \in S_2 \tag{64}$$

If the last task of the observed sequence was A and we have just observed task B and there is a plan, P, which has a subsequence containing A followed by B then our confidence that we are dealing with plan P is increased because this maintains continuity between tasks.

$$endorsement(P, B, [A|S_r], continuity\_is\_desirable(A), pos) \leftarrow plan(P, S) \wedge subsequence([A, B], S) \tag{65}$$

If the last task of the observed sequence was A and we have just observed task B and there is a plan, P_1, which contains B but there is another plan, P_2, which contains A followed by B then we are more sceptical that we are dealing with plan P_1 because this disturbs the continuity between tasks for P_2.

$$endorsement(P_1, B, [A|S_r], discontinuity\_is\_undesirable(A), neg) \leftarrow plan(P_1, S_1) \wedge B \in S_1 \wedge plan(P_2, S_2) \wedge not(P_1 = P_2) \wedge subsequence([A, B], S_2) \tag{66}$$

Suppose that the user attempts task 'a', followed by task 'b', followed by task 'd'. Using endorsement rules 62 to 66, we can generate at each step sets of endorsements for each of the two potential plans. This is shown in Figure 7. Each endorsement has been numbered for ease of reference.

Looking at the sequence of tasks in Figure 7, we can see that the set of endorsements for each plan will tend to grow larger with each successive task. This is undesirable, since the number of active endorsements could become very large for long sequences of tasks. To reduce this problem, we introduce rules for removing from the pool of endorsements those which we consider to have become obsolete due to the arrival of new informaton. In our current, simple example we use only two such rules:

If for plan, P, we have the negative endorsement that task, A, could be a mistake and we also have the positive endorsement that continuity of the plan via some task, B, is desirable then we retract the negative endorsement on A.

Task		Endorsement
a	(1)	$endorsement(plan1, a, [], could\_be\_a\_mistake(a), neg)$
	(2)	$endorsement(plan1, a, [], only\_grammatical\_possibility(a), pos)$
b	(3)	$endorsement(plan1, b, [a], could\_be\_a\_mistake(b), neg)$
	(4)	$endorsement(plan1, b, [a], continuity\_is\_desirable(a, b), pos)$
	(5)	$endorsement(plan1, b, [a], other\_grammatical\_possibility(b), neg)$
	(6)	$endorsement(plan2, b, [a], could\_be\_a\_mistake(b), neg)$
	(7)	$endorsement(plan2, b, [a], discontinuity\_is\_undesirable(a, b), neg)$
	(8)	$endorsement(plan2, b, [a], other\_grammatical\_possibility(b), neg)$
d	(9)	$endorsement(plan2, d, [b, a], could\_be\_a\_mistake(d), neg)$
	(10)	$endorsement(plan2, d, [b, a], continuity\_is\_desirable(b, d), pos)$
	(11)	$endorsement(plan2, d, [b, a], only\_grammatical\_possibility(d), pos)$

Figure 7: Endorsements for the sequence a, b, d.

$$\begin{array}{l} endorsement(P, A, S_p, could\_be\_a\_mistake(A), neg) \wedge \\ endorsement(P, B, S'_p, continuity\_is\_desirable(A, B), pos) \rightarrow \\ \quad retract(endorsement(P, A, S_p, could\_be\_a\_mistake(A), neg)) \end{array}$$

If task B allows us to switch to plan P, and thus introduces an undesirable discontinuity with in our choice of plan given task A, but B also allows a desirable continuity in plan P to task C then we retract the negative undesirable discontinuity endorsement on P.

$$\begin{array}{l} endorsement(P, A, S_p, discontinuity\_is\_undesirable(A, B), neg) \wedge \\ endorsement(P, C, S'_p, continuity\_is\_desirable(B, C), pos) \rightarrow \\ \quad retract(endorsement(P, A, S_p, discontinuity\_is\_undesirable(A, B), neg)) \end{array}$$

By applying the above rules at each step in the sequence of tasks from Figure 7 we can prune out obsolete endorsements from each plan, thus leaving fewer to be carried forward to later tasks. Figure 8 illustrates this process diagrammatically. The numbers on the diagram refer to endorsements from Figure 7. The horizontal layers denote steps in the task sequence. The introduction of an endorsement is shown by the appearance of its number in the appropriate task layer and vertical lines show the duration of the endorsement. Lines ending in a cross denote pruning of an endorsement. Those endorsements which remain in the final task are terminated by an arrow head.

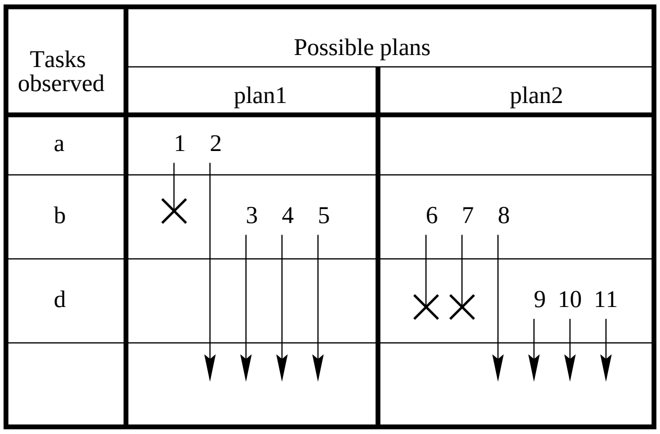


Figure 8: Example endorsement sequence

More About Justification of Decisions

21 Data Describing Reserve Sites

Cocks & Baird's reserve selections are based on a number of sets of initial data, describing important characteristics of the available reserve sites and the plant communities within them. Six types of plant community are considered, which we shall name c_1 to c_6. We use the predicate $community(C)$ to denote that C is a community of interest in the reserve selection process. Thus, to state that c_1 is such a community, we write $community(c_1)$. We do the same for the other five communities.

There are 101 potential sites which we might use as reserves and, for simplicity, we shall use the integers from 1 to 101 to refer to these sites. For every site Cocks & Baird provide a simple numerical measure of abundance for each community. We use the predicate $appearance(S, C, P)$ to denote that the index of abundance of community C at site S has the value P. This can vary from 0 (no chance of appearing in some standard sample) to 1 (always appearing in some standard sample). The clauses corresponding to this information for all communities in site 1 are:

$$
\begin{gathered}
appearance(1, c_1, 0.019) \\
appearance(1, c_2, 0.571) \\
appearance(1, c_3, 0.79) \\
appearance(1, c_4, 0) \\
appearance(1, c_5, 0) \\
appearance(1, c_6, 0.053)
\end{gathered}
$$

Similarly, we create *appearance* clauses for each community on each of the remaining 100 sites. Each site also has a known area which we record using the predicate $area(S, A)$, where S is a site and A is its area (in km^2). Thus, for site 1 the area is defined as $area(1, 27.5)$.

It is important to know whether sites possess inland and/or coastal habitat. We use the predicates $inland(S)$ and $coastal(S)$ to denote these properties, respectively, for any site, S, to which they apply. For example, we know that site 10 is inland ($inland(10)$) and site 1 is coastal ($coastal(1)$).

Finally, a few of the sites are already assigned as reserves. We use the predicate $existing\_reserve(S)$ to denote that a site, S, has this property. For instance, site 99 is an existing reserve so we define $existing\_reserve(99)$.

This completes our description of the site data given in Cocks & Baird's paper. To save space, we have not given the complete set of definitions for each of these predicates but it is straightforward to reconstruct these from the original paper, following the procedures given above.

22 Formal Requirements Guidelines

Cocks & Baird supply a number of guidelines for making appropriate reserve selection choices. We provide formal definitions of each of these below, following our normal format but with the addition of quotation marks around the italicised informal descriptions since these are taken directly from Cocks & Baird's paper. Explanatory comments appear immediately after each definition. The guidelines are of three kinds:

- $test(R, T) \leftarrow \mathcal{P}$ defines a requirement which is necessary to be satisfied by the set of selected reserves, R, in order for them to be complete in a given context. The term, T, summarises the test and the precondition, $\mathcal{P}$, gives the conditions which satisfy it (thus parameterising the test).
- $select(R, S, E) \leftarrow \mathcal{P}$ defines a way in which an individual reserve, S, may be chosen to be added to the set of selected reserves, R. The term, E, summarises the selection criterion and the precondition, $\mathcal{P}$, gives the conditions which satisfy it (thus parameterising the selection).
- $constraint(C, R) \leftarrow test(R, T_1), \ldots, test(R, T_n)$ defines an example collection of tests (T_1 to T_n) which we might require the final set of selected reserves, R, to satisfy. For convenience of reference, each of these constraints is labelled with the identifier, C.

"As far as possible ensure at least two expected appearances of each community appear in the reserve system."

$$test(R, min\_appearance(M)) \leftarrow not(\exists C. \left(\begin{array}{c} community(C) \wedge \\ sum\_appearance(C, R, P) \wedge \\ P < M \end{array} \right)) \quad (67)$$

Expression 67 acts across all communities by requiring that there should not be a community, C, such that the sum of appearances for C across the selected reserves, R, is less than M. Notice that by leaving the minimum threshold of appearance, M, as a variable we allow for the possibility of different threshold values, thus generalising from the value of 2 given in the original guideline. In all the definitions we give we shall follow this standard principle of being as general as possible without losing the option of stipulating the original guideline as a special case.

"As far as possible give preference to including large reserves in the reserve system."

$$select(R, S, largest\_area(S)) \quad \leftarrow \quad \exists A. \left(\begin{array}{c} area(S, A) \wedge \\ not(S \in R) \wedge \\ not(\exists S_1, A_1. \left(\begin{array}{c} area(S_1, A_1) \wedge \\ not(S_1 \in R) \wedge \\ A_1 > A \end{array} \right) \end{array} \right) \quad (68)$$

This is a selection condition because it provides a guideline for choosing a new site, S, to add to the list, R, of reserves already selected. The precondition first finds the area, A, of S; then tests that S does not already appear in list R; then tests that there is not some other site, S_1, with area A_1 such that S_1 doesn't appear in R and A_1 is greater than A.

"As far as possible ensure that near-coastal reserve systems are included in the reserve system."

$$select(R, S, coastal\_reserve(S)) \leftarrow coastal(S) \wedge not(S \in R) \tag{69}$$

$$\begin{aligned} test(R, min\_num\_coastal(M)) \quad \leftarrow \quad & S = \{X | (X \in R \wedge coastal(X))\} \wedge \\ & length(S, N) \wedge N \geq M \end{aligned} \tag{70}$$

We have included two conditions for this guideline. The first is a selection condition which allows the addition of a coastal reserve to the list of selected reserves, provided that it does not already appear in the list. The second is a necessary condition which stipulates that the cardinality of the set, S, of coastal reserves appearing in the list of selected reserves should be at least equal to a given threshold value, M.

"As far as possible ensure that inland reserves are included in the reserve system."

$$select(R, S, inland\_reserve(S)) \leftarrow inland(S) \wedge not(S \in R) \tag{71}$$

$$\begin{aligned} test(R, min\_num\_inland(M)) \quad \leftarrow \quad & S = \{X | (X \in R \wedge inland(X))\} \wedge \\ & length(S, N) \wedge N \geq M \end{aligned} \tag{72}$$

The above conditions are constructed in a similar way as in the case of coastal reserves.

"As far as possible favour the inclusion of sites having a high expectation of containing one or both of the two rarest communities, i.e. communities with the lowest total number of expected appearances over all sites."

$$\begin{aligned} select(R, S, max\_appearance(C, S)) \leftarrow \quad & community(C) \wedge \\ & appearance(S, C, P) \wedge \\ & not(S \in R) \wedge \\ & not(\exists S_1, P_1. \left(\begin{array}{c} appearance(S1, C, P1) \wedge \\ not(S_1 \in R) \wedge \\ P_1 > P \end{array} \right)) \end{aligned} \tag{73}$$

Expression 73 selects a new site, S, with maximum appearance value for a given community, C. The test which ensures maximality is the final line of the precondition, which states that there should be no site, S_1, such that S_1 doesn't already appear in the set of selected reserves and for which the appearance value of S_1 is greater than the appearance value of S. This condition is a little different from the textual requirement since it allows the resource manager to specify the community, C, rather than specifying that it should be "as far as possible ...one or both of the two rarest". Our interpretation of this vague part of the guideline is that there is scope for subjective interpretation and, accordingly, we have not stipulated this formally - the resource manager must decide which of the communities C will be.

"Include in the reserve system all sites which are already declared reserves."

$$select(R, S, existing\_reserve(S)) \leftarrow existing\_reserve(S) \wedge not(S \in R) \tag{74}$$

$$test(R, contains\_all\_existing\_reserves) \leftarrow not(\exists S. \left(\begin{array}{c} existing\_reserve(S) \wedge \\ not(S \in R) \end{array} \right)) \tag{75}$$

This condition states that there should not exist a site which already has reserve status and which does not appear in the list of selected reserves. We have now completed our definition of reserve selection guidelines.

Cocks & Baird give four examples of constraints on the final lists of sites chosen as reserves, which are as follows:

"...specify a reserve system of no more than 12 sites, collectively containing at least two expected appearances of each of [the] six communities...the reserve system was required to have an area of not much more, and preferably less, than 25% of the total area over all sites. Finally, there should be some spread of sites between coastal and inland locations. Solution 1...specifies a reserve system satisfying these requirements."

$$\begin{aligned} constraint(1, R) \quad \leftarrow \quad & test(R, max\_reserve\_number(12)) \wedge \\ & test(R, min\_appearance(2)) \wedge \\ & test(R, max\_proportion\_reserve\_area(0.4)) \wedge \\ & test(R, min\_ratio\_coastal(0.2)) \wedge \\ & test(R, min\_ratio\_inland(0.2)) \end{aligned} \tag{76}$$

Cocks & Baird's general guideline is not explicit about the precise values allocated to the maximum permitted proportion of area given over to reserves or to the ratio of inland to coastal reserves. We have chosen here to set the maximum proportion of area as 0.4 and the minimum proportion of inland and coastal reserves to be 0.2.

"Solution 2 retains the constraint that each community must be expected to appear in the system at least twice, but otherwise is concerned with only one other goal - to minimise the total area of all sites in the reserve system."

$$\begin{aligned} constraint(2, R) \quad \leftarrow \quad & test(R, min\_appearance(2)) \wedge \\ & test(R, min\_total\_area) \end{aligned} \tag{77}$$

This constraint introduces a particularly strong requirement: that there should not exist any other valid allocation of reserves which produces a list of sites with a smaller total reserve area than that of the current list. This is interesting from a logical point of view because it requires the definition of a constraint in terms of the reserve selection process itself, rather than simply in terms of properties of the reserves. We show how to deal with this in expression 84.

"Solution 3 also retains the constraint that each community must be expected at least twice but, in this instance, is concerned only with the goal of minimising the number of sites in the reserve system."

$$\begin{aligned} constraint(3, R) \quad \leftarrow \quad & test(R, min\_appearance(2)) \wedge \\ & test(R, min\_reserves) \end{aligned} \tag{78}$$

The representation here is similar to 77 and similar comments apply.

"Solution 4 reflects the impact of re-running the initial problem (Solution 1) with the additional design requirement...that the system must contain the three existing declared reserves..." (The solution given by Cocks & Baird also sets the maximum reserve number to 14).

$$\begin{aligned} constraint(4, R) \quad \leftarrow \quad & test(R, max\_reserve\_number(14)) \wedge \\ & test(R, min\_appearance(2)) \wedge \\ & test(R, max\_proportion\_reserve\_area(0.4)) \wedge \\ & test(R, min\_ratio\_coastal(0.2)) \wedge \\ & test(R, min\_ratio\_inland(0.2)) \wedge \\ & test(R, contains\_all\_existing\_reserves) \end{aligned} \tag{79}$$

This is similar to 76, with the addition of one extra test: that all existing reserves should appear in the list of selected sites. This completes our description of the test sets. All that remains is to define the constituent tests of constraints 1 to 4. For these we do not have quotable English definitions from Cocks & Baird's paper so we supply our own definitions (italicised as usual but not quoted).

We have a maximum number of reserves within the limit, M, if the cardinality of the set of reserves, R, is less than M.

$$test(R, max\_reserve\_number(M)) \leftarrow length(R, N) \wedge N \leq M \tag{80}$$

We are within the maximum proportion of reserves to total area if the sum, S, of all the areas of selected reserves, R, when divided by the total area of all sites is less than the limit, M.

$$\begin{aligned} test(R, max\_proportion\_reserve\_area(M)) \quad \leftarrow \quad & total\_site\_area(T) \wedge \\ & sum\_of\_areas(R, S) \wedge \\ & S/T \leq M \end{aligned} \tag{81}$$

The proportion of selected reserves which are coastal exceeds a given minimum, M if the ratio of the cardinality, L_s, of the list, S, of selected coastal reserves to the cardinality, L_r, of all selected is no lower than the minimum level.

$$
\begin{aligned}
test(R, min\_ratio\_coastal(M)) \quad \leftarrow \quad & S = \{X | (X \in R \wedge coastal(X))\} \wedge \\
& length(S, L_s) \wedge \\
& length(R, L_r) \wedge \\
& L_s / L_r > M
\end{aligned} \tag{82}
$$

The proportion of selected reserves which are inland exceeds a given minimum, M if the ratio of the cardinality, L_s, of the list, S, of selected coastal reserves to the cardinality, L_r, of all selected is no lower than the minimum level.

$$
\begin{aligned}
test(R, min\_ratio\_inland(M)) \quad \leftarrow \quad & S = \{X | (X \in R \wedge inland(X))\} \wedge \\
& length(S, L_s) \wedge \\
& length(R, L_r) \wedge \\
& L_s / L_r > M
\end{aligned} \tag{83}
$$

The total reserve area is minimised if there is no other allocation of reserves, R_1, which yields a smaller sum of areas than for the currently selected set, R.

$$
\begin{aligned}
test(R, min\_total\_area) \quad \leftarrow \quad & sum\_of\_areas(R, S) \wedge \\
& not(\exists R_1, S_1. \left(\begin{array}{c} allocate\_reserves(R_1, E) \wedge \\ sum\_of\_areas(R_1, S_1) \wedge \\ S_1 < S \end{array} \right))
\end{aligned} \tag{84}
$$

The *allocate_reserves* condition of this test requires a definition of the algorithm for reserve selection, which we supply below. However it should be noted that for large and complex reserve selection tasks there is a prohibitive computational overhead in demonstrating that no other reserve selection could ever yield a better set of selected reserves, since this requires (in the worst case) performing all reserve selections. Various forms of optimisation can be performed to approximate to this exhaustive computation but, since this opens up the broad area of constraint satisfaction methods, we shall not discuss these.

The total number of reserves is minimised if there is no other allocation of reserves, R_1, which is of smaller cardinality than for the currently selected set, R.

$$
\begin{aligned}
test(R, min\_reserves) \leftarrow \quad & length(R, L) \wedge \\
& not(\exists R_1, S_1, E.T. \left(\begin{array}{c} allocate\_reserves(T, R_1, E) \wedge \\ length(R_1, N_1) \wedge \\ N_1 < N \end{array} \right))
\end{aligned} \tag{85}
$$

23 Relating Choices to Justifications

As we explain in the book, we can test the selection definitions by satisfying the expression: $allocate\_reserves(T, R, E)$, using the index number T to select an appropriate set of tests. If, for example we use the tests in 79 (*i.e.* $N = 4$) then a valid result for R (corresponding to Cocks & Baird's solution) is:

$$R = \{99, 100, 101, 1, 87, 35, 15, 56, 57, 93, 54, 78, 79, 70\}$$

The structured term to which E is instantiated is complex to read directly but we provide a summarised version in Figure 9.

Selection methods applied (in order) in making the reserve selections, with references to the identification number of the general selection rule applied:

- $existing\_reserve(99)$ By 74
- $existing\_reserve(100)$ By 74
- $existing\_reserve(101)$ By 74
- $max\_appearance(c2, 1)$ By 73
- $max\_appearance(c2, 87)$ By 73
- $max\_appearance(c3, 35)$ By 73
- $max\_appearance(c5, 15)$ By 73
- $max\_appearance(c5, 56)$ By 73
- $max\_appearance(c5, 57)$ By 73
- $max\_appearance(c6, 93)$ By 73
- $inland\_reserve(54)$ By 71
- $inland\_reserve(78)$ By 71
- $inland\_reserve(79)$ By 71
- $inland\_reserve(70)$ By 71

Constraints satisfied by final set of reserves, as defined by expression 79:

- $test(\{99, 100, 101, 1, 87, 35, 15, 56, 57, 54, 78, 79, 70\}, min\_appearance(2))$
- $test(\{99, 100, 101, 1, 87, 35, 15, 56, 57, 54, 78, 79, 70\}, max\_proportion\_reserve\_area(0.4))$
- $test(\{99, 100, 101, 1, 87, 35, 15, 56, 57, 54, 78, 79, 70\}, min\_ratio\_coastal(0.2))$
- $test(\{99, 100, 101, 1, 87, 35, 15, 56, 57, 54, 78, 79, 70\}, min\_ratio\_inland(0.2))$
- $test(\{99, 100, 101, 1, 87, 35, 15, 56, 57, 54, 78, 79, 70\}, contains\_all\_existing\_reserves)$

Figure 9: Sequence of selection conditions satisfied in the example.

Relating Domain Languages to Logic

All the problems which we tackle in the book require us to avoid exposing some of the people involved in the design of models from the formal language used to reason about them. We have demonstrated a range of ways to do this, which we summarise below:

23.0.1 Mapping Diagrams to Logical Expressions

The commonest way in which we bridge between informal and formal notation is by defining translations from diagrammatic notations to logic. In Section 2.1 we use a box and arrow diagram as an interface for a system of binary relations. In Section 2.3.1 we use an operator diagram to give us the first stage in refinement of our formal operator definitions. Similarly, in Section 2.3.2 we use a diagrammatic representation of functions applied to data as a means of obtaining the beginnings of our decision procedure definitions. In Section 4.1 we compare two different entity description diagrams by translating each into logic. Section 4.2 describes a more abstract form of diagrammatic language, built purely with the aim of translating to set expressions and thence to FOPC. The structure of the shutdown logic used in Section 5.3 is described using a diagram similar to that used in some types of electronic circuit design. In Section 7.1 we use the diagrammatic conventions of system dynamics modelling and translate these to the logical expressions for our formal model, prior to describing some of the sources of uncertainty which it contains.

23.0.2 Describing Consequences of Inference with Diagrams

In some cases the formal structure of expressions is not the most effective way of communicating particular features of the consequences derived from formal models. Diagrams can sometimes be used to highlight the most pertinent features. In Section 2.4.1 we use tree diagrams to summarise the proof trees for particular inference strategies and to highlight inconsistent steps. Similar diagrams are used in Sections 3.4, 3.6 and 3.8 to describe more conventional proof strategies. In Section 3.9.2 we summarise the results of a basic inductive generalisation method by plotting the paths from conditions to conclusions using a tree-structured notation. Section 4.1.4 uses a box and arrow diagram to show permitted sequences of processes from a given initial state, with branches making it easy to spot non-determinism in the model. A simpler, linear, sequence is used to show individual causal chains when exploring causal interactions between requirements and design in Section 5.3. Section 5.2.1 contains a plot of the solutions obtained from a numerical model against time, while Section 7.1.2 uses a diagram to annotate the components of another numerical model with descriptions of some sources of uncertainty in their definition.

23.0.3 Using Tabular Notation to Obtain Binary Relations

A common use of tabular descriptions is as a convenient way of choosing an appropriate subset of binary relations from a set of potential relations. In Section 4.1.4 we construct a matrix of processes against entity types and, by putting differently shaped marks in each cell, choose the roles which are anticipated for appropriate types of entity in each process. In Section 5.3 we use a cause-effect matrix to supply binary relations between the inputs of the system and the outputs to which they are causally linked.

23.0.4 Constrained Input of Natural Language Expressions

Sometimes natural language, rather than a diagram, feels like the most natural way to describe some concept. However, our formal systems cannot disambiguate natural language descriptions unless they are supplied in a highly structured way. For this reason, many of our systems use natural language only at points in the interaction where there is a fixed set of choices of formal expression, which allows the system to generate natural language paraphrases of the formal expressions and present these as menus of choices to the designer. This means that the designer has responsibility only for recognising appropriate natural language expressions, not for inventing them. In Section 5.2 a menu based interface is used to assist ecological modellers in providing a problem description. In Section 7.2 we use menus of natural language statements to allow reserve selectors to choose selection strategies.

23.0.5 Linking Logical Descriptions to Text

In some of the examples we have encountered, the role of our formal expressions has been secondary to that of texts in natural language. The main aim of the design endorsement system of Section 5.3 is to allow parts of the codes of practice which regulate design to be linked to appropriate components of the design. Our interest in logic is to provide the "connective tissue" linking design to text, rather than as a way of determining the style of the text itself. A similar principle applies in Section 5.1 when generating WWW pages in HTML. Our main interest is in generating sets of pages which "look right" and the structured terms which we use to generate HTML must be able to satisfy those expectations.

Patterns of Problem Solving

If we look back at the examples from previous chapters we can see that standard patterns of refinement and analysis recur in different application domains. We summarise these in Sections 23.1 and 23.2.

23.1 Patterns of Design

Most of the methods which we use to guide the design of our models can be understood abstractly as operations on logical expressions. We summarise these below, referring back to the sections in which they were used.

23.1.1 Using Sort Restrictions to Cue Predicate Conditions

In Section 2.3.1 we begin with expressions of the form:

$$V_1 \in S_1 \ldots V_n \in S_n.P$$

where each V_i is a variable appearing in term P and each S_i is a sort restriction on the corresponding V_i. We then define a condition for each variable which conforms to the sort restriction on the variable and may relate it to other variables, thus producing the refined expression:

$$P \leftarrow C_1 \wedge \ldots C_n$$

where each C_i is a condition on V_i conforming to the sort restriction S_i and involving predicates which we define in subsequent design. A similar pattern was followed in Section 2.3.2.

23.1.2 Using Entity Roles to Cue Process Conditions

In Section 4.1.4 we define sorts of entity in our model using expressions of the form:

$$E(V_1, \ldots, V_n) \in S \rightarrow V_1 \in S_1 \wedge \ldots \wedge V_n \in S_n$$

where E is an entity name and each V_i is a variable restricted by a sort S_i of entity on which E depends. We then take each sort of entity, S, and for each process, P, which involves S we categorise the involvement of S in P. The three roles which sorts of entity can have in processes are to be originated, to be required or to be changed, which we define using predicates of the form:

$$originates(P, S)$$
$$requires(P, S)$$
$$changes(P, S)$$

We then construct more detailed process descriptions using the expression:

$$process(P, Or, Re, Ch, De, X)$$

and use our category information to assign appropriate instantiations of entities ($E(V_1, \ldots, V_n)$) or their attributes to either:

- Or if $originates(P, S)$ and $E(V_1, \ldots, V_n) \in S$
- Re if $requires(P, S)$ and $E(V_1, \ldots, V_n) \in S$
- Ch if $changes(P, S)$ and $E(V_1, \ldots, V_n) \in S$ and P changes the value of an attribute of $E(V_1, \ldots, V_n)$.
- De if $changes(P, S)$ and $E(V_1, \ldots, V_n) \in S$ and P removes $E(V_1, \ldots, V_n)$.

23.1.3 Set Inclusions Translated to Conditionals

In Section 4.2 we have sort inclusions such as:

$$Q \subseteq P$$

which we reinterpret as set comprehensions over predicates - in this case:

$$\{V_1 | Q(V_1)\} \subseteq \{V_2 | Q(V_2)\}$$

and we then translate these to more standard FOPC expressions - in this case:

$$P(V) \leftarrow Q(V)$$

23.1.4 Instantiation of Parameterisable Components

In Sections 5.2 and 5.3 we define parameterisable design components, each of which is a set of predicate definitions in which there may be gaps for predicate names; relations; or sometimes entire subsets of predicate definitions. These gaps are filled by constraints attached to the component. In addition, each component may introduce new predicate names for which we need to construct definitions using the application of other parameterisable components. For example a parameterisable component might contain a set of predicates:

$$\{P(X, Y) \leftarrow Q(X) \wedge \mathcal{F}\}$$

with the constraint:

$$function(\{X\}, Y, \mathcal{F})$$

denoting that we must construct a function from X to Y which fills gap $\mathcal{F}$, and that we need to define Q via some other component. Although the parameterisation mechanisms are domain independent, constraints are satisfied by referring to problem descriptions which are domain specific.

23.1.5 Structured Predicate Construction

Chapter 6 describes a method for constructing the definitions of individual predicates by applying sequences of transformations. Each transformation is intended to correspond to a key concept in the design of the predicate, such as recursive construction of a data structure. An example of this sort of transformation would be to extend a predicate:

$$\begin{array}{rcl} P(X) & \leftarrow & Q(X) \\ P(F_1(A_1, B_1)) & \leftarrow & P(B_1) \end{array}$$

where $F_1(A_1, B_1)$ is a structured term, to the predicate:

$$\begin{array}{rcl} P(X, Y) & \leftarrow & Q(X) \wedge \mathcal{R}_1(X, Y) \\ P(F_1(A_1, B_1), F_2(A_2, B_2)) & \leftarrow & \mathcal{R}_2(A_1, A_2) \wedge P(B_1, B_2) \end{array}$$

where $F_2(A_2, B_2)$ is a structured term built recursively by relating each A_1 to A_2 via the relation $\mathcal{R}_2$, and the relation $\mathcal{R}_1$ ensures that the base case of the constructed term (Y) is instantiated from the base case of the deconstructed term (X).

23.2 Patterns of Analysis

Our analysis methods, like those of design, can be summarised as operations on logical expressions. We recap on these below.

23.2.1 Properties of Networks of Binary Relations

In Section 2.1 we define an argumentation network using typed binary relations. We then write definitions for properties of the network. The example we used was a transitive relation over the network:

$$\begin{array}{rcl} P(A, B) & \leftarrow & L(A, B) \\ P(A, B) & \leftarrow & L(C, B) \wedge P(A, C) \end{array}$$

where L is the binary relation defining the network (we ignore typing of links here for simplicity) and P is the transitive relation.

23.2.2 Meta-Interpretation to Accumulate Data

In Section 2.4.1 we use meta-interpretation to describe a control strategy for the dynamics of our model. In that example, we used a different meta-interpreter for the operator model than for the decision procedure model but the same principle applies to both. To emphasise the basic principle let us use a simplified interpreter, which is:

$$\begin{array}{rcl} P(input(X)) & \leftarrow & p\_output(Op, X) \\ P((A \wedge B)) & \leftarrow & P(A) \wedge P(B) \\ P(D) & \leftarrow & clause(D, C) \wedge P(C) \end{array}$$

where: the first expression finds a solution for some input, X, to the operator model if there is an output via some operator, Op, which yields X; the second expression finds a solution for a conjunction from solutions for each side of the conjunction; and the third expression finds a solution for a decision goal, D, if there is a clause giving a condition, C for D and there is a solution for that condition. This gives us a strategy for testing whether certain results are derivable from the model but it doesn't give us information about how these were derived. For example, we might want to have a record of the operators which were used in deriving a given result. To do this we extend the meta-interpreter with a pair of arguments which accumulate this extra information, giving the new interpreter:

$$\begin{array}{rcl} P(input(X), S, [s(Op, X)|R]) & \leftarrow & p\_output(Op, X, S, R) \\ P((A \wedge B), S, F) & \leftarrow & P(A, S, N) \wedge P(B, N, F) \\ P(D, S, F) & \leftarrow & clause(D, C) \wedge P(C, S, F) \end{array}$$

where the first of the two additional arguments to P is the sequence of operators applied so far and the second of the additional arguments is the final sequence of operators once the result is obtained.

23.2.3 Adapting Solution Strategies to Test Hypotheses

At the end of Section 2.4.1 we find that conflicting results can be obtained from our model and suspect a problem in the decision procedure. Our difficulty is to find out where this occurs. To help us investigate, we construct a revised interpreter which tests for limited forms of contradiction. The advantage to this form of extension to our meta-interpreter is that it makes explicit the method by which we are testing our hypothesis (that conflicts in decisions cause inconsistencies in our model) and automates part of the testing of that hypothesis.

23.2.4 Using Entity Roles to Test Process Descriptions

Recall that in our summaries of design we described how definitions of key roles of entities in process were used in Section 4.1.4 to cue more detailed process descriptions. One of the entity roles was for the entity, of sort S, to be originated in a process, P - denoted by $originates(P, S)$. Each process is defined by the predicate $process(P, Or, Re, Ch, De, X)$, where Or is the set of entities actually originated when the process is run. However, our method for generating process definitions doesn't guarantee that all the entities which should play a role in the process actually do so. Therefore we construct tests for non-adherence to these roles which can be applied to a process which has run to completion. A simplified version of one of these is:

$$problem(P, Or) \leftarrow originates(P, S) \wedge not(E \in Or \wedge valid\_state\_type(E, S))$$

which detects a problem with the set of originated entities, Or, if there is an origination role for an entity of sort S but no entity, E, in Or which is in sort S.