TBPG – A Techniques-Based Program Generator

Alberto Castro

MSc Information Technology: Knowledge Based Systems

Department of Artificial Intelligence

University of Edinburgh

1994

Abstract

In this work we address program generation for simulation models in a restricted class of ecological domain. A first approach to deal with this problem, the EL Program Generation System was presented in [Robertson *et al.* 91]. That system produced a Prolog program by assembling modular predicate-level definitions obtained from a library of program schemata. The approach presented here is to use *Prolog Programming Techniques* as the element of program construction which acting in a clause-level, is expected to make the generation process more flexible and explicit. The resulting system, TBPG – A Technique-Based Program Generator, embodies the ideas developed through this dissertation. It is a two-level processing tool which uses a Techniques Editor to construct a Prolog predicate by using a techniques library, controlled by a Generation Control module which manages the generation process.

Acknowledgements

I wish to thank my supervisors, Dave Robertson and María Vargas-Vera for their guidance, support and enthusiasm. Thanks also to Wamberto Vasconcelos and Andy Bowles for helpful suggestions and support in the writing. During this year of study, I was on leave from University of Amazon, Brazil; sponsored by CAPES Foundation under grant n. 1651/93-8.

Table of Contents

List of Figures

Chapter 1

Introduction

The EL Program Generator System was one of the main tools produced within the Eco-Logic project [Robertson *et al.* 91]. This tool generated a complete simulation model, in a form of a Prolog program, by using some building blocks called program schemata. These schemata were concerned with the definition of whole predicates.

This work addresses an alternative approach to that problem – generation of programs for simulation models in a restricted class of ecological models, using as the basic element of programming construction, *Prolog Programming Techniques*, a concept presented in [Brna *et al.* 91]. This approach has some advantages, such as:

- It makes it easier to deal with the program generation at both clause and predicate levels.

- It makes more explicit the basic units (in this case, techniques) used to construct predicates in a specified domain.

- It may allow, by sharing the same basic structure (techniques manipulation), the use of a wide class of tools to assist the user and to formalise the manipulation of code in Prolog programming.

1.1 The Eco-Logic Project

In [Robertson *et al.* 91], there was described an alternative approach to building simulation models in a very complex and poorly structured domain: that of ecological modelling.

The basic idea was to use logic-based approaches to describe an ecological situation. From the representation of that situation, it is possible to obtain, by successive refinements, a simulation model.

The research carried out in the Eco-Logic project, had the following features:

- It described an alternative approach to building a simulation model. In that approach, simulation models were described using logical statements, addressing some questions about the use of logic programming and its techniques in that new domain.
- It supplied some practical tools to implement such ideas.
- It provided to the final users (ecological modellers) a standard (formal) notation and methodology suitable to people with little computing experience, making more explicit the process itself.

One of the main tools produced in the project was the EL system. EL can be viewed as an Intelligent Front End [Bundy 84] which utilises a sorted logic problem description language to provide an interface for constructing Prolog simulation program.

The EL system was composed of two subsystems:

1. a problem description system;
2. a program generator, which produces the completed Prolog simulation program.

In EL, the mechanism for generating a complete program lies in a structure of building block called Prolog schema.

A schema has the code needed for a part of a simulation model, as well as information about the conditions under which it should be used. These condition declarations and the code itself, were represented as Prolog structures, making good use of Prolog resources, such as instantiation. Therefore, a schema can be adapted according to the context within it is been manipulated.

However, schemata are concerned with the definitions of whole predicates. This makes it difficult to allow the program generator to alter the definition of predicates at clause level. Thus, it is impossible for users to customise schemata, other than through the predefined instantiation procedures of each schema.

It is argued that by allowing users to adapt at clause level, a number of improvements could be obtained, such as:

- To make explicit standard methods for constructing Prolog predicates, thus encouraging structured program development.

- To link the schema generating mechanism to hitherto unrelated work in the area of techniques editing.

- To allow the user to identify the main patterns of program development, typically used in a specific domain.

This leads us to explore the use of Prolog Program Techniques as the basic element of program construction.

1.2 Prolog Programming Techniques

It is common for programmers to follow standard patterns of development when constructing code. Some questions concerning the definition, recognising and using of such patterns, called *program techniques* were addressed by Brna *et al* in [Brna *et al.* 91]. Considering the Prolog language, Brna *et al* established the concept of *Prolog Program Techniques.*

Although a technique may be specific to a particular language, it is normally independent of any algorithm or problem domain. The same technique can thus be used in different tasks, as well as different techniques being combined together to do a more complex task. Beyond considerations about how to identify, extract and represent techniques, Brna proposed several uses for Prolog Techniques, such as: programming, teaching, intelligent tutoring, debugging, program construction and language design.

Some of the products of these research efforts are close to our interests:

- Techniques-based editors.

  Two examples of editors whose editing structures are techniques are [Robertson 91] and [Bowles 93]. Both of them are targeted at novices programmers, but they have different approaches to deal with techniques.

  A techniques editor can be used alone, for example in a tutorial task. On the other hand, such editor can be used as a component, in a more complex system, where a editing task is required for another component or another level of editing.

- Program transformation using techniques.

  Vargas-Vera in [Vargas-Vera *et al.* 93a], addressed the combination of two programs constructed by means of a specialised techniques editor. The aim is to produce a more efficient combined program, with little user interaction. The basic idea was the use of *program history*, a record of the enhancements made to the basic structure when using a techniques editor, to provide guidance during the process of combination.

  To be successfully combined, both programs must share the same basic structure, called *the skeleton*, or a restricted form of mutation of the basic skeleton.

- Managing techniques.

  There is no consensus over what is the "right" set of techniques for a particular domain, so it is useful to be able to extract techniques information from examples.

  In [Vasconcelos 94a], Vasconcelos has proposed a medium in which programmers can design, test and organise Prolog Programming Techniques. His goal was to provide an automated tool, a *Prolog Techniques Meta-Editor* which enables techniques to be abstracted from annotated source code and then combined to form new program.

- Several other systems involving the concepts of Prolog Techniques, have been developed, covering many contexts [Bowles & Brna 93,Bowles & Robertson 93, Bowles & Vasconcelos 93,Vargas-Vera *et al.* 93b,Vasconcelos 94b]. A survey of some of these topics is presented in [Bowles *et al.* 93].

There are two main approaches to deal with techniques when manipulating a Prolog code.

The first one, deals with techniques acting inside the predicate body, generally exploiting the relationship between the head and recursive arguments in the recursive clauses of programs. Such approach seems to be more useful in specific application, as that one described in [Bowles 93].

The other approach, more widely used, was introduced in [Kirschenbaum *et al.* 89], where was presented a methodology called *stepwise enhancement*, to build structured and standardised Prolog programs. The basic approach taken is to consider firstly, a structure embodying the basic control flow (*skeleton*) and incrementally apply over it, the standard Prolog Programming practices (*techniques*). So, this approach deals with techniques over a clause level, but acting over a complete program body, lying over a basic control flow. This approach will be discussed more fully in Section 2.3.2.

1.3 Goal

From the intersection of the two research topics presented briefly in sections 1.1 and 1.2, some issues were raised:

1. As pointed out in the section 1.1, the use of Prolog schemata to generate programs can lead to some limitations in a clause-level predicate definition. How can Prolog Programming Techniques be used as the basic element in the program generation? Is it feasible? Is it suitable?

2. It is argued in several works cited in section 1.2 that techniques are a context-free pattern (in fact, a small number of techniques can be used to construct a large range of Prolog programs). However, it has been considered that some techniques are more suitable for programs in a specific domain than others. So, it is worth identifying and/or defining a set of techniques to be used by, for example, ecologists.

3. Considering the answers to item 1, can we implement a basic system for applying techniques obtained in item 2, to partial predicate definition, ultimately producing complete predicates?

4. How would the system described in item 3 be integrated with a reconstructed version of predicate-level schema application mechanism?

Thus, the main goal of this work is to attend these issues above, which can be summed up as:

> *"To reconstruct the Eco-Logic Program Generation System, using Prolog Programming Techniques as the basic element of program construction."*

1.4 Structure of The Work

In the next chapter, the schema-based approach and the Prolog programming techniques are discussed in more detail. Chapter 3 discusses the domain considered, explaining how the techniques are used in such context, as well as defining which techniques must be used. Chapter 4 deals with the system itself, addressing considerations about the project and implementation in the two levels considered in the system (techniques application and predicate level mechanism). Chapter 5 addresses the conclusions of this work, revising what was presented, summing up the main results and pointing out directions to future research. Finally, the code of Techniques Editing and Generation Control modules are presented in Appendix A and B respectively.

Chapter 2

Prolog Schemata and Prolog Programming Techniques

The purpose of this chapter is to present the main concepts which will be further considered in this work, namely: Simulation Model, Prolog Schemata, Prolog Programming Techniques, Prolog Techniques Editing and Program Generation.

We begin, in Section 2.1, with a brief description of some aspects involving the construction of simulation models and the use of Prolog to do so. In Section 2.2, the approach of Prolog Schemata is described, including practical examples in the program generation. In Section 2.3, we address the main points about Prolog Program Techniques, particularly the use of Techniques Editors. Finally, in Section 2.4, the use of Programming Techniques to program generation is discussed.

2.1 Simulation Models and Prolog

A model reflects one's understanding, and allows that that understanding can be accessed, used and evaluated by others. For this reason, models are an important part of the scientific process.

More the area considered is complex, more the models are important. Considering a system from the domain considered, a model makes more explicit the behaviour of this system, and, depending of the characteristics of this model, the scientific rationale for that behaviour.

So, as scientific tools, models can be used to:

- representing one's understanding – A model is a very common way to make accessible a theory. A model can also embodying several ideas.

- evaluating one's understanding – The simplest way to check a model, and ultimately the one's understanding, is to compare the behaviour of the model with that of the real world. The evaluation can be qualitative or quantitative.

- increasing one's understanding – From the use and evaluation of a model, it is expected an increasing in the one's understanding.

Of course, the same system can be modelled in many ways. Actually, there are several modelling paradigms, and the choice of one of these, depends initially on the specific domain considered.

2.1.1 Why Prolog?

One of the main features of a simulation model is to communicate one's understanding and (in most cases) to do this not only by presenting a correct behaviour, but by making clear the criteria considered in its construction, thus allowing it to be scientifically validated. To do so, it is important that this model be properly represented and implemented, providing an adequate documentation for the modelling decisions. In the task of constructing programs to be simulation models, these considerations had to be still more carefully observed.

However, programs intended to be simulation models usually are seen by their users as 'black boxes' which exhibit certain external behaviours, although be very difficult to understand the mechanisms inside. This is due mainly to nature of imperative programming languages, which make difficult to meet the standards expected to a simulation model, named in [Robertson *et al.* 91]. These standards, along with some short comments about the suitability of conventional notation (CN), are showed below:

1. The notation should be:

- unambiguous – each symbol should have one, and only one, meaning. In CN, often it is difficult to know what a structure is meant to represent.

- modular – any part of our argument should make sense when considered separately from the others. Some resources in CN, such as global variables, make possible to break the modularity.

- parsimonious – there should only be as much information as is necessary to fully understand the problem. CN has many structures to specify the mechanics of computation, mixed in with the description of the ecological problem.

- explicit – the details of all assumptions should be easily accessible. In CN, the assumptions tend to be hidden in the code.

2. It should be possible to state our assumptions without committing ourselves to a particular method for deriving conclusions. Using CN, it is difficult to separate the information describing the problem from the information stating how the problem will be dealt.

3. The development of our model should be based on sound forms of inference, as well as open to retrospective analysis and comparison. There is no obvious way of to prove, in a CN, that each step taken was justifiable, especially if we consider the highly procedural interpretations of some structures (e.g. "do loop" or "goto"). And for these inadequacies, to make analysis or comparisons between models it is very difficult indeed.

Since that the paradigm of imperative programming often does not fit to our requirements for a model, we must consider the use of alternative approaches. The logic paradigm provides a clear separation between the axioms defining the assumptions and the inference methods used in a model and is very close satisfying all the standards cited above. If we accept the idea of to use logic as a notation to represent simulation models, a starting point is to adopt a logic programming language. Prolog is the most widely accepted of such languages.

Under this point of view, instead of a sequence of instructions telling the computer how to run a model, the programmer provides a set of Prolog clauses which describe the structure of the model and then, these clauses can be processed by a standard Prolog interpreter, providing answers to users' questions.

2.2 Prolog Schemata

The EL system was composed of two subsystems: a problem description system and a program generator. The EL Program Generator, destined to construct a complete Prolog simulation program, has utilised as the basic element to the construction of a structure called *Prolog schema*.

2.2.1 Definition – Examples

A Prolog schema is a structure to packaging the Prolog code necessary to implement a part of a complete simulation model. But a schema has packaged inside it, not only this piece of Prolog code, but also the information about the conditions under which that schema should be used, and the requirements to be used along with other schemata.

A schema is defined in the form:

schema(Name,Goal,SubGoals,Clauses,Actions,Preconditions)

Where:

- *Name* – Name for the schema.

- *Goal* – The simulation goal for which the schema solves.
  A schema solves only for one goal, conjunctions or disjunctions of goals are not allowed. If *Preconditions* and *Actions* execute successfully, and the SubGoals are satisfied, then the schema is guaranteed to solve this Goal.

- *SubGoals* – List of new goals which the schema produces.

- *Clauses* – Is the piece of Prolog code actually supplied by the schema to the final simulation model.

- *Actions* – Procedure calls which must be satisfied before the schema can be applied. These actions are not used to *check* if the schema is applicable (this is made by *Preconditions*), but a set of actions to be done after the schema be tested as applicable.

- *Preconditions* – A procedure call which must be satisfied before the schema can be used. This is actually the 'applicability checking'.

The structure of schemata are illustrated and commented in the examples below:

Example 1

```
schema(['assign a value for the time at which simulation begins'],
       initial_time(_),
       [],
       [(initial_time(X)
           :- X = N)],
       [prompted_input('Initial time >> ', number(N), [], [N])],
       true ).
```

This schema solves for the goal *initial_time(_)*, by adding the code:

```
initial_time(X) :- X = N
```

There are no subgoals generated by the application of this schema, and the precondition is the atom `true`, which always succeeds.

A schema is represented using the Prolog syntax and for this reason, embodies the resources of that language, for example, the unification. To interpret a schema, even the simple one above, it is necessary to be aware of the relationship between the parameters in the several parts of its structure, and how the instantiation process acts in this relationship.

Example 2

```
schema(['decide how to calculate a previous time in the simulation'],
       last_time(T,T1),
       [time_step(S)],
       [(last_time(T,T1)
           :- nonvar(T),
              var(T1),
              time_step(S),
              T1 is T - S),
        (last_time(T,T1)
           :- nonvar(T1),
              var(T),
              time_step(S),
              T is T1 + S)],
       [],
       true).
```

The main difference from the previous example, is that the code to be added to the simulation model is not a single definition. Two definitions are needed because different calculation must be done depending whether `T` or `T1` is instantiated when the goal is called.

In this case, a subgoal (`last_time(T,T1)`) is generated as a result of applying this schema, and there is no precondition nor action required.

Example 3

```
schema(['determine ',L,' of ',O,'as a random function of its previous',L],
       attribute(L,O,Tm,_),
       [initial_time(Tm),
        last_time(Tm,_),
        initial_value(L,O,_)],
       [(attribute(L,O,T,N1)
         :- \+ initial_time(T),
```

```
        last_time(T,T1),
        attribute(L,O,T1,N2),
        random_float(attribute(L,O,T), RND),
        N1 is RND * N2),
    (attribute(L,O,T,N3)
     :- initial_time(T),
        initial_value(L,O,N3))],
   [],
   (L == x_coordinate ; L == y_coordinate) ).
```

This example is slightly more complex than those previous, as explained below:

- The schema is applicable only if the parameter `L` is instantiated for `x_coordinate` or `y_coordinate` when the goal is called.
- There are two definitions for `attribute`/4, where one of them has a recursive call.
- There is a list of subgoals to be further accommodated. It is worth noticing that these need not correspond to those generated by the Prolog code which will be inserted as a part of the simulation model.
- Variables can be used even in the name of the schema.

Of course there are more complex schemata, which make more effective use of the structure of a schema and the Prolog syntax. Nevertheless, we believe the basic idea of schemata representation could be captured with these examples.

2.2.2 Manipulation by the EL System

There are two main sorts of simulation systems: domain independent and domain dependant.

Domain independent systems have no explicit knowledge of the problem to be solved using the simulation model and for this reason, cannot assist the user to make decisions about the model (e.g. how to convert the real problem in the representation language used in the system), except those possible to be obtained from syntactic resources of the programming language being used.

On the other hand, domain dependant systems can be tuned with the problem which is to be solved with the simulation model. The good aspect of this is that it is possible a strong support to the user, facilitating the tasks of problem specification and orientation of the solution process.

EL which was a domain dependant system, which has a program description module to help the user in the selection of which schemata must be applied, by reducing the range of schemata suitable for application in a determined context.

Defining the simulation program is the last of the four phases in a complete EL session, and this phase can be divided in two stages:

1. EL extracts the initial queries, which are properly instantiated.

2. EL constructs a program, by the recursive application of schemata.

An informal definition with some Prolog-like terms, of the recursive algorithm to schemata application,

Recursive_App(UD,PD,SL,RB,Goals,ClausesIn,ClausesOut)

where: *UD* is the universe of discourse,
PD is the problem description,
SL is the schemata library,
RB is a rules base (used as a basis for conducting dialogue during an extraction phase),
Goals is the list of goals (initially set to the list of extracted queries),
ClausesIn is the set of simulation program clauses already constructed, and
ClausesOut is the final set of simulation program clauses.

The algorithm for schema application is summarised below:

IF *Goals* is empty,
THEN *ClausesOut* is empty.
OTHERWISE:

- Select the first goal (*Goal*) from *Goals* leaving the remaining elements (*RestGoals*)
- IF *Goal* can be established from *ClausesIn*
  THEN apply
  Recursive_App(UD,PD,SL,RB,RestGoals,ClausesIn,ClausesOut).
  OTHERWISE:
  - Find the set *SS* of schemata which are valid means of satisfying *Goal*;
  - show the set *SS* to the user, who must select one of them (*S*);
  - fire the actions of schema *S*;
  - add the clauses in *S* to *ClausesIn*, generating the new set of clauses *NewClauses*;
  - apply
    Recursive_App(UD,PD,SL,RB,RestGoals, NewClauses,PartialClauses);
  - apply
    Recursive_App(UD,PD,SL,RB,RestGoals, PartialClauses,ClausesOut).

2.3 Prolog Programming Techniques

Prolog programmers have often associated their experience with the way them use some simple Prolog patterns in sophisticated applications. This section address some aspects of these pattern, called *Prolog programming techniques.*

2.3.1 Characterisation

Differently from other programming concepts, like algorithms or programming clichés, a small number of programming techniques are used in a wide sort of problems, which suggest they are context-free. However, programming techniques are peculiar to each programming language, i.e. they are language-dependant.

A variety of common constructs occurring in Prolog programs have been recognised and received names such as 'accumulator pair', 'difference structure', 'constructing data-structures in the clause head' and 'failure driven loop'. Despite these terms have a correspondent meaning at the code level, they have a lack of precise definition and for this reason, are generally only illustrated by giving examples of predicates containing instances of them.

An example from [Brna *et al.* 91], which uses a standard technique: 'constructing data-structures in the clause head', is shown below:

```
double([],[]).
double([H1|T],[H2|Ans]) :-
        H2 is 2 * H1,
        double(T,Ans).
```

In that work, Brna *et al* suggested some possible applications mainly related with programming environments and teaching programming skills:

- Programming – By providing generalised pieces of Prolog code that can be used to implement part of a specification.

- teaching – By supplying the student with the structures which must be manipulated in the programming task. These techniques can be used to form a well designed set of examples to illustrate the Prolog programming in several levels.

- intelligent tutoring systems – In a ITS for Prolog, techniques might be the main internal element, used for interpret a student's Prolog program. The ITS must relate this program with the correct and appropriate use of programming techniques.

- debugging – Static debuggers can have a library of buggy techniques, or a representation stating which technique is intended to be associated with the predicate. Dynamic debuggers can use the detection of a technique to simplify the checking in a search tree.

- program construction – By allowing designers of logic programs work with higher-level techniques, make the processes of design, edit, test and debug programs easier and more structured.

- language design – By expanding the basic Prolog language, through of the formal definitions of techniques.

Nevertheless, several aspects on programming techniques and the pragmatics of their applications are not yet precisely known. Among these points, we can cite:

- The initial point – how to identify and to capture techniques?

- The selective point – which techniques must be included to an application? There must be a small number or a very wide set? What are the criteria for this selection?

- The representational point – what is the best way to representing the set of techniques? Can this approach be used by another techniques-based system?

- The systematic point – what are the best methodologies to manipulate techniques?

There is no consensus about the answers for these questions, which it has motivated a reasonable number of works in these areas. A survey of works addressing the topics above can be found in [Bowles *et al.* 93]. The next subsection focus on some works resulted from the research topics above (which are related with our purpose here).

2.3.2 Developing Programs Using Techniques

Kirschenbaum *et al* proposed a widely accepted approach to developing logic programs – *stepwise enhancement*[Kirschenbaum *et al.* 89]. The basic idea is to conduct the construction of well structured, standardised Prolog programs, by separating the different basic control flows – the *skeletons*, from the various standard Prolog programming practices, which we refer as *additions*[1].

This approach can be summarised as follows:

1. The first step is to isolate the basic control flow needed to solve the problem and embody it in a skeleton. Alternatively, one can select a skeleton from a previously defined library.

2. Over the skeleton defined/selected, extra computations are included by applying additions to yield an *extension* of the basic skeleton.

3. This extension can now, be regarded as a (new) skeleton which allows to repeat the process until the final program has been developed.

In his work, Kirschenbaum state some skeletons as well as a basic set of techniques. For example, let's consider his skeleton to traverse a list:

```
traverse_n([X|Xs]) :-
   case_1(X),
   traverse_n(Xs).
.
.
.
traverse_n([X|Xs]) :-
   case_1(X),
   traverse_n(Xs).
traverse_n([]).
```

Consider an specialisation of the skeleton above, with only one recursive clause. Consider also the 'accumulate' addition which adds two arguments to a skeleton. One of them is used as a stack to push down data, and the other returns the final object from the stack.

---

[1]Kirschenbaum uses the term *technique*, but in this work, we will consider both skeleton and 'addition' as specialisations of technique.

The result of applying the addition to the skeleton, which includes the two arguments and provides an extra initialisation clause, is showed below:

```
traverse_1_accumulate(Xs,Total) :-
    traverse_1_accumulate(Xs,[],Total).

traverse_1_accumulate([X|Xs],Current,Final) :-
    update(Xs,Ys,Current,Next),
    traverse_1_accumulate(Ys,Next,Final).
traverse_1_accumulate([],Final,Final).
```

Of course further application of additions to this extension will be needed to define a definition of `update`/4. This approach to program construction has been used in the development of tools in some areas particularly interesting to the goals of this work. Some of these tools are briefly discussed in the next paragraphs.

The concepts of programming techniques were straight derived from the way in which programmers acquire and use their skills of a particular language. Most of this expert knowledge is still obtained from daily practical situation, which justify the effort to build better environments to guide the construction of programs. Not incidentally the two editors we mention here are oriented to novices Prolog programmers.

The first editor, described in [Robertson 91] was built directly from the step-enhancement methodology. This editor has a set of skeletons and additions represented in a DCG notation, which can be manipulated (under guidance) by the user, to ultimately produce a correct Prolog program. The second editor was presented in [Bowles 93], and it defines techniques as common patterns of code which capture the relationship between the head and recursive arguments in the recursive clauses of programs, i.e. techniques act inside the body of one predicate rather than over a whole program.

The main advantage of these two editors is to concentrate the attention of the user (student) in the main constructors used in Prolog programming, allowing him/her to experience (and to understand) standardised and structured program construction. Programming techniques have also been used to manipulate programs in more sophisticated ways than generation of simple programs in a novice-oriented system. In [Vargas-Vera *et al.* 93a,Vargas-Vera *et al.* 93b] for example, is addressed the combination of two programs to produce more refined and efficient programs.

In this approach it is assumed that the programs which are to be combined must be constructed in a specialised techniques editor, which can produce a record of the enhancements made to the basic program skeleton (the *program history*). The composition system contains a set of *methods* (a sequence of transformation operations) to combine two programs with the same flow of control or with specific forms of mutation of the basic skeleton. These methods are given to the user in an interactive system to help the construction of the desired program.

Techniques can be applied in several stages of the program construction, but is not clear that all user will find general techniques easy to use, and it has been argued that a better approach would be the user specify a set of techniques suitable for his/her use. In [Vasconcelos 94a], Vasconcelos has proposed a medium in which programmers can design, test and organise Prolog Programming Techniques. His goal was to provide an automated tool, a *Prolog Techniques Meta-Editor* (TME) which enables techniques to be abstracted from annotated source code and then combined to form new program. TME defines a most general technique and allows the user to specialise it. However, there is no consensus over what is the "right" set of techniques for a particular domain, so it is useful to be able to extract techniques information from examples. [Vasconcelos 94b] address the automatic extraction of programming techniques from working Prolog programs, using the formalism proposed in [Vasconcelos 94a].

Beyond simple examples which use Prolog programming techniques concepts, these works stimulate the use of this paradigm in new applications by supplying a significant theoretical and practical basis.

2.4 Contribution to Program Generation

As showed in Section 2.2, a schema is a predefined Prolog structure which can be adapted according to the context within it is been manipulated. However, schemata (and consequently their manipulation mechanisms) are concerned with the definitions of *whole predicates*, and it is impossible for users to customise schemata, other than through the predefined instantiation procedures of each schema.

About this problem, [Kirschenbaum *et al.* 89] pointed out:

> "Schemata are a general framework for a class of problems to be solved where the control flow and the additions are indistinguishable. This forces the programmer, who wants to use schemata, to find a particular instance of a schema that is similar to the problem being worked on."

While the schemata-based approach allows a high-level definition at cost of some lack of flexibility, basic Prolog syntax is very flexible but slightly difficult for non-experienced programmers. We argue that the Prolog Programming Techniques concepts can be used to balance the question *flexibility* $\times$ *simplicity of use*.

To support this point of view – to explore the use of Prolog Program Techniques as the basic element of program construction, the following aspects were be considered:

- About the task (program generation) – The use of programming techniques and particularly the step-enhancement approach, has been supported by several works addressing these concepts in the analysis and manipulation of Prolog programs (e.g. [Robertson 91,Vargas-Vera *et al.* 93b,Bowles *et al.* 93, Vasconcelos 94b]). This can provide us with both a practical and theoretical basis to this new application.

- About the final user (non-experienced programmers) – A similar group of users (novice programmers) was the focus in some Prolog editors, based on programming techniques [Robertson 91,Bowles 93].

- About the improvements expected – The use of programming techniques could lead to furtherly advantages such as:
  - To make explicit standard methods for constructing Prolog predicates, thus encouraging structured program development.
  - To link the schema generation mechanism to hitherto unrelated work in the area of techniques editing. This could enhance the flexibility of the system.
  - To allow the user to identify the main patterns of program development (the techniques),typically used in a specific domain.

In the next chapter we discuss some aspects of the domain considered and its relationship with the Prolog Programming Techniques concepts.

Chapter 3

Domain-Dependant Aspects

In this chapter, we take an example of an ecological model in Prolog and use it for defining the set of programming techniques which will then be used in program generation. In the same context, we address the basic features of ecology, as well as the whole ecological modelling process, with particular emphasis on in the knowledge acquisition tasks involved.

3.1 Our domain – Ecology

Ecology has been defined as "the scientific study of the interactions that determine the distribution and abundance of organisms" [Krebs 78]. This broad definition suggests how general, complex and poorly understood are the subjects in this science. Nevertheless, large-scale problems which human beings are facing nowadays (management of natural resources, pollution, worldwide climate changes, etc.) require better ecological understanding. The following aspects of ecology make it difficult to adopt a conventional scientific approach:

- in ecology, subjects are distributed over several "levels of organisation". The focus of interest depends directly on the level in which the problem has been considered (e.g. individual level, population level, ecosystem level, etc.).

- ecological systems are characterised by complex interactions. An object of interest affects many other ecological components and in turn is affected by them. Therefore, in most cases is not possible to understand one object (and its behaviour) separately.

- often the units of study (individual organisms, species, sub-systems, etc.) to a problem are not clearly defined, and can be different depending on the point of view from which a problem is addressed.

- ecological systems are characterised by many sources of uncontrolled variation and therefore it is very difficult to know the initial state, and the exact influence of external factors. Therefore, it is difficult to conduct a well controlled ecological experiment.

Thus, an ecological system must be studied considering its inherent features, its interaction with other systems, and in the context of appropriate external factors (point of view considered, specific focus of interest, etc.).

3.2 Our Example of Simulation Model

We take as our basic example, a program produced by EL system, listed in Appendix C.5 of [Robertson *et al.* 91]. This program was constructed to represent and to calculate the attribute `biomass` of an object of the sort `wolf`. This attribute varies with the attributes `age` and `location` of wolf.

To facilitate the understanding of the code produced, it was divided in three groups: *initialisation parameters, attribute definitions* and *complementary definitions*, which are described in the following sections.

The code includes basic Prolog built-in predicates and a "utilities set" which is supplied by another part of EL system, and it is independent of any model.

3.2.1 Initialisation Parameters

In our example there are some definitions which are intended only to supply basic data for other more complex definitions. From now on, these definitions are called *initialisation parameters.* These data are obtained from interaction with the user during the problem description stage and their representation must be structurally as simple as possible, of the same sort used to represent primitive knowledge (facts).

Every set of the following definitions represents a different piece of knowledge; has different uses; and imposes a different influence on the final program.

parameters

Parameter values for the competition coefficient of `young` and the self inhibition coefficient of `oldpop`.

```
parameter(competition_coefficient,young,A) :-
    A=0.3.
parameter(self_inhibition_coefficient,oldpop,A) :-
    A=0.2.
```

initial values

Initial values to all subpopulations of `young` and of `oldpop`.

```
initial_value(biomass,young_pop3,A)  :-  A=8.
initial_value(biomass,young_pop2,A)  :-  A=7.
initial_value(biomass,young_pop1,A)  :-  A=5.
initial_value(biomass,oldpop,A)      :-  A=20.
```

initial time

Initial time point in the simulation.

```
initial_time(A) :-
    A=1.
```

initial step

Initial step in the simulation.

```
time_step(A) :-
   A=1.
```

initial location

Initial location of for the spatially represented objects in the model (the subpopulations of `young` and of `oldpop`).

```
initial_location(oldpop,grid(3,3)).
initial_location(young_pop3,grid(2,1)).
initial_location(young_pop2,grid(1,2)).
initial_location(young_pop1,grid(1,1)).
```

adjoining grids

These predicates inform all grid squares which are nexts to each other.

```
adjoining(grid(3,4),grid(2,4)).
adjoining(grid(3,4),grid(3,3)).
adjoining(grid(3,3),grid(2,3)).
...
adjoining(grid(1,2),grid(1,1)).
```

sub-objects

Denotes a membership relation, in this case, the sub-objects of the population `young`.

```
subobj(young,young_pop3).
subobj(young,young_pop2).
subobj(young,young_pop1).
```

3.2.2 Attribute Definitions

These clauses define the attribute of `biomass` for each of objects of population: `young` and `old`, as well as to the sub-objects of young population: `young_pop1`, `young_pop2`, and `young_pop3` The calculation varies to each object considered.

biomass of old population

Defines the biomass for `old` population. The calculation is based in a competition equation which involves the `young` population.

```
attribute(biomass,oldpop,A,B) :-
    initial_time(A),
    initial_value(biomass,oldpop,B).
attribute(biomass,oldpop,A,B) :-
    \+initial_time(A),
    last_time(A,C),
    attribute(biomass,oldpop,C,D),
    attribute(biomass,young,C,E),
    parameter(self_inhibition_coefficient,oldpop,F),
    parameter(competition_coefficient,young,G),
    B is 1-D*F-E*G.
```

biomass of young population

Defines the biomass for `young` population. The calculation is based on the sum of the the biomass of the sub-populations `young_pop1`, `young_pop2`, and `young_pop3`.

```
attribute(biomass,young,A,B) :-
    setof(C,constituent_object(young,C),D),
    for_each(E,D,attribute(biomass,E,A,F),F,G),
    sum_elements(G,B).
```

biomass of sub-objects of young population

Defines the biomass of each sub-object of young population. The calculation is based on the detection of the populations in proximity to the given population at the appropriate time, as well as the calculation of the biomass of the populations in proximity. To avoid repetition, is showed only the code for `young_pop1` (for the other sub-objects, the code is the same, except by the name of the correspondent sub-object).

```
attribute(biomass,young_pop1,A,B) :-
   initial_time(A),
   initial_value(biomass,young_pop1,B).
attribute(biomass,young_pop1,A,B) :-
   \+initial_time(A),
   last_time(A,C),
   attribute(biomass,young_pop1,C,D),
   setof_or_empty(E,in_proximity(young_pop1,E,A),F),
   for_each(G,F,attribute(biomass,G,C,H),H,I),
   sum_elements(I,J),
   length(I,K),
   (   J=0,
       B=D
   ;   B is D+J/K/2
   ).
```

3.2.3 Complementary Definitions

In this category are the remaining definitions (most used by `attribute`/4).

location

Defines the location (`B`) of an object `A`, at time `C`.

```
located_at(A,B,C) :-
   initial_time(C),
```

```
    initial_location(A,B).
located_at(A,B,C) :-
    \+initial_time(C),
    last_time(C,D),
    located_at(A,E,D),
    setof(F,next_to(E,F),G),
    move_to_square(G,A,D,E,B).
```

proximity

Succeeds if an object `A` is in proximity to an object `B`, at time `C`.

```
in_proximity(A,B,C) :-
    located_at(A,D,C),
    located_at(B,D,C),
    \+A==B.
in_proximity(A,B,C) :-
    located_at(A,D,C),
    adjoining(D,E),
    located_at(B,E,C).
```

last time

Defines the previous time point (`A`) of a time point `B`.

```
last_time(A,B) :-
    integer(B),
    var(A),
    time_step(C),
    A is B+C.
last_time(A,B) :-
    integer(A),
    var(B),
    time_step(C),
    B is A-C.
```

constituent object

Denotes that the object B belongs to the sort A.

```
constituent_object(A,B) :-
    subt(A,C),
    constituent_object(C,B).
constituent_object(A,B) :-
    subobj(A,B).
```

3.3 Knowledge Acquisition in Ecological Modelling

Before beginning *techniques identification*, which were characterised in [Brna *et al.* 91] as a knowledge acquisition task, we shall discuss a little about other knowledge acquisition tasks involved in the ecological modelling process.

In moving from a complex ecological domain to a simulation model implemented in a Prolog program, there is a large amount of knowledge which needs to be identified, represented, organised, manipulated and if necessary, adjusted. In addition, it is unlikely that these tasks can be accomplished by a single person[1]. The "path" from the basic domain (ecology) to a final simulation program is showed in Figure 3–1[2]. In this figure, we can see four main stages of the process:

1. The ecological domain – which is the starting point in the process. The knowledge is subjected to those features cited in Section 3.1.

[1] Probably two experts, an ecologist and a programmer, are involved in this process. As in the Eco-Logic project, we are looking to reduce (ultimately to eliminate) the necessity of a programmer in this process

[2] We are considering the context of this thesis i.e., to use programming techniques to generate the final code.

2. The model – which is intended to represent a piece of the ecological domain (a specific system). The conception of a model involves specifying the aspects of the domain which must be represented (which part of the original knowledge must be manipulated).

3. The program specification – which states how a program must be, which knowledge it must embody and which behaviour must present in order that stands for a simulation model.

4. The final code – a program constructed to suit the specifications, embodying the knowledge about *what* a simulation model is and *how* it presents its behaviour.

Figure 3–1: Knowledge Acquisition in Ecological Modelling

In Figure 3–1, we also show three "steps" in the path we are considering. These steps involve three sorts (or levels) of knowledge acquisition. They are:

I – To define a model to represent a system from the ecological domain. Among the tasks involved, we can cite:

   – To select an appropriate model type (statistical, theoretical/analytical, numerical simulation, other).

   – To select an appropriate modelling paradigm i.e., a mathematical framework in which the system to be modelled is expressed (differential equations, difference equations, discrete-event simulation, matrix population models, Markov models, systems dynamics, etc.) or a more general paradigm which can combine aspects of several of these basic paradigms.

   – To represent, using the suitable sort of model selected, the general parameters of the system.

   – To make the basic assumptions.

   – To make the hypothesis to be tested or the general assumptions to be refined.

   Suitable model definition is itself, a specialised task whose criteria are still largely informal. The insertion of this task into an automated-oriented process leads us to reflect about methodologies to formalise it.

   If the modelling paradigm allows, the next step could be the specification of a program.

II – To implementing the model in a program, a formal specification of the system is necessary. The first difficulty in this task is to transfer an informal knowledge about the features of the system (an expert knowledge) in a formal representation, which must also be understood by the expert. This is a typical knowledge acquisition task which shares the same problems faced in the development of expert systems[3]. However, conventional approaches often are not concerned about the peculiarities of a dynamic modelling process.

III – This part of the process is where the topics presented in this thesis are really included (the program generation task). The general approach points to use a expert knowledge in programming, the programming techniques to generate a Prolog code which must attend the program specification (derived from expert knowledge in ecological domain).

Nevertheless, the path we describe here, is not a top-down, unidirectional process. From the nature of a model as a scientific tool which allow to develop one's understanding, the results obtained using a simulation model must be evaluated, and the process initiated again, hearing in mind the new knowledge obtained. Thus model construction is an iterative process.

3.4 Techniques Identification - Our Example

The essential first step, using our approach, is to define a set of suitable Prolog programming techniques. Beyond the program specifications, experience in the generation of other models must be used to drive the techniques definition, thus "tuning" the techniques set, and reducing its size at the same time.

The simplified approach we take here is to define the techniques set from the example showed in Section 3.2. The basic idea is to extract techniques from that code[4]. An example of technique which was identified is showed in Figure 3–2, where three different definition blocks have a common programming pattern which can be described as follows:

- in each block there are two clauses: one recursive and one non-recursive.

---

[3]See [Durkin 94] for a good survey on methodologies of knowledge acquisition.

[4]An automated approach to do this proposed in [Vasconcelos 94b].

- in each block, the "selective" predicate, which determines whether a clause is applicable or not, is `initial_time`/1.

- in each block, the recursive application of the predicate is based on a new value given by the predicate `last_time`/2.

These characteristics are common to the three blocks independently from the name and arity of the predicate considered (`attribute`/4 in the two firsts and `located_at`/3 in the last). Similar process was used to define all the techniques.

```
attribute(biomass,young_pop3,A,B):-
 initial_time(A),
 initial_value(biomass,young_pop3,A,B).
attribute(biomass,young_pop3,A,B):-
 \+ initial_time(A),
 last_time(A,C),
 attribute(biomass,young_pop3,C,D),
 setof_or_empty(E,in_proximity(young_pop3,E,A),F),
 for_each(G,F,attribute(biomass,G,C,H),H,I),
 sum_elements(I,J),
 length(I,K),
 ( J=0,
 B=D
 ; B is D+J/K/2
 ).


attribute(biomass,oldpop,A,B):-
 initial_time(A),
 initial_value(biomass,oldpop,B).
attribute(biomass,oldpop,A,B):-
 \+ initial_time(A),
 last_time(A,C),
 attribute(biomass,oldpop,C,D),
 attribute(biomass,young,C,E),
 parameter(self_inhibition_coefficient,oldpop,F),
 parameter(competition_coefficient,young,G),
 B is D + (1 - D * F - E * G).


located_at(A,B,C):-
 initial_time(C),
 initial_location(A,B).
located_at(A,B,C):-
 \+ initial_time(C),
 last_time(C,D),
 located_at(A,E,D),
 setof(F,next_to(E,F),G),
 move_to_square(G,A,D,E,B).
```

Figure 3–2: Technique – Recursion Over Time Points

3.4.1 Representation and Taxonomy

To facilitate the techniques manipulation as well as their portability, we decided to represent techniques in a simple Prolog structure which syntax is presented (in a BNF-like notation) below:

```
technique(name,[internal_variables,definition_1,...,definition_n])
```

where:

```
internal_variables   = [v_1 , ... , v_n]
definition_i         = base_definition |
                       recursive_definition
base_definition      = [Head , BodyPart_1 ,..., BodyPart_n]
recursive_definition = rec( base_definition )
Head                 = [Functor , Parameter_1 ,..., Parameter_n]
BodyPart_i           = [Functor , Parameter_1 ,..., Parameter_n]
Functor              = A Prolog atom
Parameter_i          = A Prolog list
v_i                  = A Prolog variable
```

To explain each one of these terms, we shall use the technique `time_recursion` as an example.

```
P(T) :- initial_time(T).
P(T) :- \+ initial_time(T),
        last_time(T,Tp),
        P(T).
```

Of course, this is not an "runnable" Prolog code, the name of the predicate cannot be a variable (in this case: `P`). So, to transfer this technique for our proposed representation, we need to consider:

- internal variables – beyond the parameters in a predicate, we need to consider which "internal" variables might be useful in the future, when this code be manipulated by a techniques editor. In this example, the value `Tp` which corresponds to the last time point before T, is very likely to be needed by another predicate included by a techniques application.

- clauses – There are two sorts of clauses which define a Prolog predicate: recursive and non-recursive. It is important to known promptly whether a clause is recursive. In our representation, a recursive clause requires be represented as the argument of the predicate `rec/1`.

- terms and lists – In our representation, all terms are transformed in their equivalent lists i.e., the functor will be the first element of such list and the arguments will be the remainder elements in the list, and this definition is recursively applied. (eg: `p(X,Y)` is represented as `[p,X,Y]`; `p(a(X))` is represented as `[p,[a,X]]`).

Thus, this technique is represented as:

```
technique(time_recursion,
   [      [Tp] ,
          [ [P,T] , [initial_time,T] ],
          rec([ [P,T] , [\+,[initial_time,T]],
                        [last_time,T,Tp] ,
                        [P,Tp] ])
   ] ).
```

The techniques were divided in two groups: *plain techniques* and *basic definitions*. The first group is used to represent the general class of techniques, which is intended be used in the program generation. The other group is in fact an specialisation of the first, and is used to represent initialisation parameters, as mentioned in Section 3.2.1.

Certainly the nomenclature used here is different from those cited in the references on programming techniques, but this can be justified by a domain-oriented approach which searches to be closer with the expert's terminology. Nevertheless, it is possible to describe the techniques presented here in more usual programming terms (search, traverse, etc.), and even to make this translation automatically.

3.4.2 Plain Techniques

recursion over time points

This technique embodies the basic flow of control to do a recursion over time points. If a time point `T` is not the initial time, the same definition must be applied over the previous time `Tp` until the initial time 'be reached'. It is the skeleton for most of the recursive definitions in our example.

```
technique(time_recursion,
   [      [Tp] ,
          [ [P,T] , [initial_time,T] ],
          rec([ [P,T] , [\+,[initial_time,T]],
                        [last_time,T,Tp] ,
                        [P,Tp] ])
   ] ).
```

membership relation

It represents a recursive definition of membership relation. If an object is not a immediate member of a class, it could be a member in any sub-class of the class considered.

```
technique(constituent_object,
   [      [] ,
          [ [P,A,B] , [subobj,A,B] ] ,
          rec([ [P,A,B] , [subt,A,C] ,
                          [P,C,B] ])
   ] ).
```

base case with initial value

This technique embodies an initial value declaration which can be used to form or to extend a base case in some recursive definition.

```
technique(based_on_value,
   [      [ ] ,
          [ [P,What,Over,Value] , [initial_value,What,Over,Value] ] ,
          [ [P,What,Over,Value] ]
   ] ).
```

set generation

This technique contains two mutually exclusive forms of to form a set:

- by enumerating all objects which belong to a class;
- by enumerating all objects in proximity of a spatial location.

```
technique(set_generation,
   [      [ ] ,
          [ [P,In1,In2,Set] ] ,
          [ [P,In1,In2,Set] ,
            [or,[and,[setof,X,[constituent_object,In1,X],Set],!],
                [setof_or_empty,X,[in_proximity,In1,X,In2],_MetaS]] ]
   ] ).
```

operation over a set

It defines a specific sort of operation over a set. After the generation of the set, it is obtained the sum of some attribute common to every object in the set.

```
technique(sum_over_a_set,
   [      [Set,NewSet]     ,
          [ [P,In1,In2,In3,Sum] ] ,
          [ [P,In1,In2,In3,Sum] ,
            [set_generation,In1,In2,Set] ,
            [for_each,X,Set,[attribute,In1,X,In3,Int],Int,NewSet],
            [sum_elements,NewSet,Sum] ]
   ] ).
```

alternative calculations

It defines which calculation must be made over some parameters, based on their original values.

```
technique(alternative,
   [       [K] ,
           [ [P,I,J,D,B] ] ,
           [ [P,I,J,D,B] ,
             [length,I,K] ,
             [or,[and,[=,J,0],[=,B,D]],[is,B,[+,D,[/,J,[/,K,2]]]]] ]
   ] ).
```

location definition

It defines the set of neighbouring squares and the selection of a movement.

```
technique(location,
   [       [] ,
           [ [P,A,B,D,E] , [initial_location,A,B] ] ,
           [ [P,A,B,D,E] ,
             [setof,F,[next_to,E,F],G],
             [move_to_square,G,A,D,E,B] ]
   ] ).
```

double

This technique defines a competition equation.

```
technique(double,
   [       [] ,
           [ [P,In1,In2,B,D,C] ,
             [initial_value,In1,In2,B] ] ,
           [ [P,In1,In2,B,C,D] ,
             [attribute,In1,young,C,E],
```

```
          [parameter,self_inhibition_coefficient,In2,F],
          [parameter,competition_coefficient,young,G],
          [is,B,[+,D,[-,[-,1,[*,D,F]],[*,E,G]]]] ]
   ] ).
```

proximity

It defines when two positions are proximate – in the same grid square or in adjoining grid squares.

```
technique(proximity,
   [      [] ,
         [ [P,A,B,C] ,
           [located_at,A,X,C] ,
           [located_at,B,X,C] ,
           [not,[==,A,B]] ] ,
         [ [P,A,B,C] ,
           [located_at,A,X,C] ,
           [adjoining,X,E] ,
           [located_at,B,E,C] ]
   ] ).
```

increase or decrease

It determines the value of a time point given a anterior or posterior (next) point.

```
technique(increase_decrease,
   [      [],
         [ [P,A,B] ,
           [integer,A],
           [var,B],
           [time_step,C],
           [is,B,[-,A,C]] ],
         [ [P,A,B] ,
           [integer,B],
```

```
          [var,A],
          [time_step,C],
          [is,A,[+,B,C]] ]
   ] ).
```

3.4.3 Basic Definitions

```
technique(time_step,         % time step in the simulation
   [      [] ,
          [ [time_step,1] ]
   ] ).
technique(initial_time,      % initial time point in the simulation
   [      [] ,
          [ [initial_time,1] ]
   ] ).
technique(initial_value,     % initial value of the populations
   [      [] ,
          [ [initial_value,biomass,young_pop3,8] ] ,
          [ [initial_value,biomass,young_pop2,7] ] ,
          [ [initial_value,biomass,young_pop1,5] ] ,
          [ [initial_value,biomass,oldpop,20] ]
   ] ).
technique(initial_location,  % initial locations of the populations
   [      [] ,
          [ [initial_location,oldpop,[grid,3,3]] ] ,
          [ [initial_location,young_pop3,[grid,2,1]] ] ,
          [ [initial_location,young_pop2,[grid,1,2]] ] ,
          [ [initial_location,young_pop1,[grid,1,1]] ]
   ] ).
technique(parameter,         % parameters
   [      [] ,
          [ [parameter,competition_coefficient,young,0.3] ] ,
          [ [parameter,self_inhibition_coefficient,young,0.2] ]
   ] ).
```

```
technique(subobj,            % membership relation
   [       [] ,
           [ [subobj,young,young_pop3] ] ,
           [ [subobj,young,young_pop2] ] ,
           [ [subobj,young,young_pop1] ]
   ] ).
technique(move_to_square,    % movement between grid squares
   [       [] ,
           [ [move_to_square,A,B,C,_,[grid,E,F]] ,
             [random_select_member,[move,B,C],[grid,E,F],A] ]
   ] ).
technique(adjoining,         % adjacency of grid squares
   [       [] ,
           [ [adjoining,[grid,3,4],[grid,2,4]] ] ,
       ...
           [ [adjoining,[grid,1,2],[grid,1,1]] ]
   ] ).
```

Chapter 4

A Technique-Based Program Generator

In this chapter, we describe TBPG - A Technique-Based Program Generator which implements the ideas we have introduced in the previous chapters. In the first Section, we show the conceptual aspects of the system, the two levels of control, and the objects involved. In the second Section, we discuss the implementational aspects, explaining how modules are built and methodology utilised. Finally, we present an example of TBPG at work, showing some stages in program generation.

4.1 Conceptual Aspects - Project

To define a program from a simple specification is not a task which can be entirely automated. Consider for example that the predicate `attribute`/4 must be defined. To facilitate the task, consider that some information about the purpose of this predicate is also supplied. Even if we consider some formal approach to program generation (for example, use of Programming Techniques), without additional guidance there is no straight way to avoid a combinatorial explosion when solving this problem[1]. For this reason, rather than a fully automatic program generator, our system is intended to be a tool to *support* the user in program generation, making more explicit *how* this process is in fact done.

[1] [Laurière 90] presents a good introduction on problem solving and AI

To understand how the system was conceived and implemented, it is worth remembering how a manual program generation is conducted. Obviously we shall consider the construction of a *Prolog* program, as well as the use of the concepts of skeletons and additions previously presented. The stages in program generation were showed in Figure 3–1. We are mainly interested in step III: to produce a final code from a specification.

To construct a program, can be informally described as a process which, starting from a problem description, involves three main tasks:

t-1. to determine the problem solving approach – the programmer analyses the problem and adopts a way to solve it. Of course there is a large number of factors which must be carefully considered in this phase, but two questions must be clearly defined at the end of this stage. They are:

- how the objects which compose the problem will be represented – this representation must consider aspects such as: strengths and limitations of the programming language adopted, fidelity with the thing represented, relationship with the solution process adopted, etc.
- the solution process – ie: how the problem progresses from input arguments to results, which side-effects must be produced, which assumptions need to be made, and how to use the language resources.

t-2. to define the program's skeleton – with the knowledge of *what* the problem is, and *how* to solve it, the programmer must define the basic flow of control to be adopted. As mentioned in the previous chapters, this tasks is made by using the programmer's expertise to select a suitable skeleton from an "internalised" library. If one have in mind to automate some parts of this process, he/she might be interested to know what this library if made from, as well as how to access it.

t-3. to extend the skeleton – starting with a basic skeleton, the programmer make successive adaptations in the code, by using pieces of code intended to attend specific sub-goals (additions). Again, programmer's expertise defines the association of sub-goals to be attended and the internalised library of additions which can be used in that context.

Nonetheless, it is unlikely that a final program can be obtained after only one "cycle" in the process above. A more pragmatic view is to consider some trial-and-error process in the task `t-3`, until a suitable code be obtained. Sometimes it is necessary to return to `t-2` and occasionally to task `t-1`.

In addition, to be able to work, the final code resulting from the above process must have all its components (another predicates) completely defined as well. In other words, all predicates used in the final code which are not built-in Prolog predicates nor some pre-defined "utility", must be specified by using the same generation process, recursively.

Thus, we can recognise two different, albeit related, levels in program generation. These levels define the basic architecture of our system. The first level deals with the definition of one predicate, and from now on will be called the *Predicate Definition Level*. The second level controls the chaining of definitions required to complete a program, it will be called the *Control Generation Level*.

4.1.1 The Predicate Definition Level

Our system deals with the definition of a predicate in a similar way to that presented as "the human approach". Though the distinct point is the explicit use of the techniques set defined in Chapter 3 as building unit in that process. Basically, our system is a techniques editor with an approach to program construction similar to that presented in [Robertson 91] and therefore it uses the step-enhancement methodology, as presented in [Kirschenbaum *et al.* 89].

In this simple techniques editor, the definition process[2] involves three main steps:

- Initialisation – This must be the first task in the process. It consists of selecting the skeleton of the predicate. Although some techniques are more likely to be a skeleton, the user is in charge of to make this selection.

[2] The code's construction of a Prolog predicate.

- Technique Application – Starting from an input code (initially a skeleton), the user selects another technique which he/she believes can attend some part of the solution process defined. Of course this task involves checking if the input code and the technique selected are compatible, which can involve a task called *parameter adjust*, to extend the number of arguments in the input code. The code resulting from this task is an *extension* of the basic skeleton and can be used as the input code in the next technique application.

- Progress/Undo – The definition process can progress through successive techniques application until a final code be reached or, when bad extensions were recognised, it undoes one or more steps until returning to a state whose code can be again extended using different techniques.

4.1.2 The Control Generation Level

The other level concerns with the chaining of predicates which are required be defined properly, in order to obtain a final Prolog code able to work in a standard interpreter. These definitions will be considered the *goals* in the system, and the contents of the goals list along with the code previously generated, constitute a *state* of the code generation.

This sort of control over the generation process is a reconstruction of the original EL predicate-level schema application mechanism.

The basic tasks in this level are:

- Goal Selection – Which defines, using some appropriate criterion, what must be the next goal, that is the next definition which must be made.

- Techniques Editing Request – Which "fires" the process described in Section 4.1.1.

- Updating – This task, which must be done after techniques editing, involves *code updating* and *process updating* stages. The code updating consists of extending the Prolog code defined until that point. From the new code generated, the system must extract another list of definitions required. From this list, the definitions which have already not been in the original list are then included, updating the generation process.

- Process Information – The system must inform the user about the state of the process. This task is relevant mainly if options for saving and retrieving a state are also supplied in the system.

- Saving/Retrieving – The store option allow the user to interrupt a generation process saving all information needed to restart later.

4.1.3 Functional Diagram

Figure 4–1 shows the functional diagram of our system. Rectangular boxes represent the two process levels in the system and the boxes with rounded corners represent the main objects. The methodology can be summed up as: from a first predicate, or set of predicates to be defined (*goals*), the *generation control* module requires the definition of that predicate. This definition is be made in an interactive *techniques editor*, which has a *techniques library* manipulated by the user in successive interactions, ultimately producing *Prolog code*. This code is then evaluated by deriving new goals which are in turn added to the goals remained. The process is started again until there are no goals generated/remained, or the user demands a stop.

Figure 4–1: TBPG Functional Diagram

4.2 Implementation

The system was implemented using a common Prolog interpreter – SICStus 2.1 #8, running in SUN or HP workstations available in the Department of Artificial Intelligence (DAI) of the University of Edinburgh. All the code uses only basic Prolog syntax and no external libraries are needed.

4.2.1 Techniques Editing

The typical process in the techniques editor is to apply a technique over some input code. This code is internally represented in the same way as the clauses within a techniques[3]. For example, the code:

```
xpto(X) :- first(X).
```

[3]See Section 3.4.1

```
xpto(X) :- \+ first(X),
           last(X,Xo),
           xpto(Xo).
```

It is represented as[4]:

```
[ [Xo] , [[xpto,X],[first,X]] ,
         [[xpto,X],[\+,[first,X]],[last,X,Xo],[xpto,Xo]] ]
```

Obviously, we need to display the code for the user, and to include that definition in a Prolog session, or even to save and retrieve in a proper way. So, the system must be able to swap these representations.

Figure 4–2 shows the main modules in our editor. The first and simplest module is the *initialisation*. It uses the predicate's name and the technique informed by the user to produce a first version of the code, starting this instance of the definition process. The *parameters adjust* module is used to adapt the number of arguments in a predicate's definition, which can be used to determine the applicability of a technique[5]. As implemented at moment, the parameter adjust acts only over the head in non-recursive clauses, and both head and body in recursive clauses. The *techniques application* module extends an input code by adding "customised" technique code. The main difficulty in this task is matching the variables in a technique definition with the constants and variables in the input code. Rather than suggest possible matchings, the editor just asks the user to provide this matching[6]. A list of matches is created and is used to generate the new extended code.

Finally, to let the user know the form of the code before and after each step in the editing process, there is a module to transform the internal code representation in a standard Prolog code, allowing "pretty printing".

---

[4]We are considering that variable `Xo` can be used later

[5]The user is asked for supply an estimate for this number.

[6]Optionally, a unmatched variable can be assumed

Figure 4–2: TBPG Techniques Editor

4.2.2 Generation Control

This is the main module in TBPG. It embodies the interface and conducts all necessary tasks in program generation.

The interface of the system is very simple. It consists of selecting menus and asking several objective questions. The user controls the "navigation" through the menus by selecting a suitable item in a presented list[7]. The highest level of control in the system is represented by predicates `menu/1`, which transfer the control from a module to another, and `menu/2`, which works inside each module. The control over modules (using `menu/1`) is explicitly represented by `next/3` (eg: `next(initial,1,new_code)`). Also the interface defines a predicate `read0/1` which gets from the user an input data whose end-of-sentence marker used in Prolog (dot) is optional.

Figure 4–3 shows the TBPG modules at generation control level. The first module is *start/retrieving*. The basic way of beginning program generation is to start the process with a simple goal – a predicate to be defined. The user must

[7] See Section 4.3.

provide the name and arity (estimate) of the predicate[8]. However, the user may want to continue a previous process whose last state was stored in a file. In this case, after all information stored is read, the user is conducted to a menu within the goals manipulation module. There is, actually, another way to starting a process: the user may exit the system, continue in the same Prolog session and enter TBPG again and continue the process from the last state obtained. This works as a rough "shell" between the system and the Prolog interpreter. Nevertheless, it is necessary that the structures generated[9] by the system to represent a state remain untouched. These structures are:

- `goals_list`/1 – List with the pairs [name , arity] remaining in the process.
- `goal_now`/1 – Pair [name , arity] of the next goal to be selected (if already defined).
- `code`/1 – List with all the code already defined in the process.

After some predicates are defined, or after (re)starting a process from a previously stored state, the user needs to know the state of the process at that time. The printing module displays to the user the information about that state that is, *missing definitions* in the goals list, and *previous definitions*, showing all predicates that have been already defined in that process.

The main module in the system is the *goals manipulation*. It controls the "core" tasks in the process. Its sub-modules are:

- Goal selection – The usual option in the process is to take the first predicate in the list, but it is important to allow the user select any predicate.
- Techniques editing – Described in the Section 4.2.1.

---

[8] Optionally, the system can suggest a predicate to be defined.

[9] Once that these structures need be modified in execution time, assert/retract resource is used.

- Derivation – After the code for a predicate has been generated by the techniques editor, it is necessary to check that all predicates used inside the code, are available to be used in a Prolog session.

  Every predicate in the code should then be checked whether it must be included in the goals list to be defined in a posterior step. A predicate does not need be included in the goals list if:

  - it is a Prolog built-in or a pre-defined utility (which can be checked by `system/1`);
  - it has already been defined by the system in a previous step, or
  - it already is on the goals list.

- Updating – Firstly, the goals list must be updated to include the new goals determined in the derivation task. The code generated must be actually included as a valid definition in a Prolog session. The internal representation is then transformed in a suitable Prolog standard code (a temporary file – eco.tmp, is used). and made "executable" by using assert/1.

Finally, the *end/saving* module allows the user to save a state in the process. There is no difference between saving an intermediate state or a final state, since the basic structures that must be stored are the same. Optionally, the user may want to use the Prolog interpreter (to run some predicate recently defined, for instance) and can exit the system to come back later, continuing from the same state.

4.3 A Running Example

To illustrate the features described in this chapter, we describe the main stages of a TBPG session.

Assuming that TBPG has been loaded correctly, the first menu presented to the user is shown in Figure 4–4{left}. If the user wants to continue a previously saved session he/she must select option 2 or 3, and the system goes directly to the generation control menu. In our example we selected option 1 and the system

Figure 4–3: TBPG Generation Control

asked us for the name and arity of the (first) predicate to be defined. We used `attribute`/4, which is the default predicate provided by the system.

The next menu, shown in Figure 4–4{**right**}, allows the user to see all the techniques library before starting techniques editing. An extract from that list is shown in Figure 4–5. In TBPG, the following standard is used to display input codes and techniques:

- input code – variables: `v_` followed by an integer.
  predicate name: normal.

- techniques – variables: `$_` followed by an integer.
  predicate name: `Tech`. The name of the technique is printed just before the code.

When the user selects option 2 in that menu, techniques editing is started, and the system asks the user to select the skeleton. We selected `time_recursion`. After the skeleton has been provided, the system proceeds to instantiate the skeleton and shows the resulting code which will be used as input code in the next steps in techniques editing. The techniques editing menu allow the user to adjust the parameters or to apply a technique over the input code. In our example, we have added three parameters over the basic code. Figure 4–6{**left**} shows the

Figure 4–4: {left}Entrance — {right}Selecting the skeleton

first inclusion of a parameter, in this case, a variable[10]. When the parameter to be added is a variable, the system asks if that variable will pass on without modification in a recursive process. If the answer is no, a different (new) variable is placed in the body of the recursive clause. The effect of including a new parameter can be observed comparing the input code and the resulting code.

Figure 4–6{right} shows the most important task in techniques editing, the application of a technique over a code. In our example, we have applied three techniques: `based_on_value`, `sum_over_a_set` and `alternative`. In this figure, we can see the input code and the technique code. The system asked us to provide the matchings, and then the resulting code was shown. After a suitable sequence of parameter adjustings and techniques applications[11], the code is saved and the control is transferred to the generation control module, which asserts that code in

[10]Regardless of the name of a variable indicated, the system assumes its internal representation and display the code as the standard mentioned before.

[11]The Undo option is very useful in this process

```
 ===== time_recursion ======
Tech($_1) :-
   initial_time($_1).
Tech($_1) :-
   \+(initial_time($_1)),
   last_time($_1,$_2),
   Tech($_2).
 --- Arguments/Variables ---
[$_1,$_2]
 ==========================
 === constituent_object ====
Tech($_1,$_2) :-
   subobj($_1,$_2).
   ...
```

Figure 4–5: Extract from Techniques Listing

a Prolog session, proceeds to the derivation of new goals and updates the goals list.

After leaving the techniques editing menu, the system presents the control generating menu, as shown in 4–7{left}. At that time, we requested to see the list of goals. There were three goals in that list; two of them could be defined directly by using basic definitions(`initial_time`/1 and `initial_value`/3). When the goal selected can be defined by using a basic definition, the system simply informs the user that the predicate has been asserted in a Prolog session, and then extracts it from the goals list, returning to the generation control menu.

Back to our example, when the remaining predicate in the goals list (`set_generation`/3) was selected, the system requested another techniques editing and the process of defining `attribute`/4 was followed again. After the predicate `set_generation`/3 had been defined, the state of the system was as shown in 4–7{right}.

This process could be continued until a empty goals list is produced, but we selected the `End` option to save that state and quit the system.

Figure 4–6: Techniques Editor — {left}Adj parameters {right}Apply technique

Figure 4–7: Generation Control — {left}Goals list {right}Show predicates

Chapter 5

Conclusions

In this chapter we review the main points which define this project: the problem addressed and the solution approach adopted. This is followed by an evaluation in terms of achievements and limitations of our system. Then directions for future work are given.

5.1 Review

In this dissertation, we addressed the problem of generating a Prolog program to represent a simulation model in part of an ecological domain.

This problem was first addressed in the Eco-Logic project, described in [Robertson *et al.* 91]. The program generation approach used by Eco-Logic[1] was *Prolog schemata*, which are concerned with the definition of whole predicates. It was argued that to use clause level definitions in the program generation can lead to several improvements. In this context, our approach to solving this problem was to use Prolog program techniques as the basic element of program construction. It was argued that this approach makes the generation process more flexible, by allowing the user to act in a clause level, and more explicit, by allowing the user to know and to manipulate the "building units". In addition, our approach links the

[1]Implemented in the EL Program Generator System

schema generation mechanism to hitherto unrelated work in the area of techniques editing, as well as the use of programming techniques in a very specific domain.

5.2 Evaluation

The system described here is a tool to support the modeller to define all the predicates in a program ultimately producing a code for simulation model. A model presented in Appendix C.5 of [Robertson *et al.* 91] was used as our basic example.

This system has two level of processing: Generation Control, which is the reconstruction of EL Program Generator System application and manages the process in the other level – the Techniques Editor, which allows the user to manipulate a techniques library to generate a predicate's code.

5.2.1 Achievements

The stated goals of this project were:

- To take the example program and define techniques which would enable the construction of the predicates in the example.

- To construct a basic system for applying these techniques to partial predicate definitions, ultimately producing complete predicates.

- To integrate these with a reconstructed version of the predicate-level schema application mechanism.

As described in previous chapters, all these stages were addressed and therefore the main goal – to produce a reconstruction of the EL Program Generation System, using techniques as the basic element of program construction – was satisfied.

A survey of the main points of the proposal was made and resulted in the construction of this system. So, we contributed to making the generation process:

- more flexible – the user can compose a program by acting at clause level, and

- more explicit – the building units (*what*), and the structured methodology of program construction (*how*) are explicit to the user.

Of course, another result of this project was the identification of further work.

5.2.2 Limitations

As a prototype, our system has several features which need to be improved. The most important are:

- guidance – To define a predicate by using a techniques editor, the user must select a series of techniques to be applied over some code. During this task our system gives no guidance, which is unhelpful, specially if we consider an inexperienced user.

- interface – the rough user interface was implemented using the standard Prolog predicates. For this reason the input task is sometimes tiring, it requires for example that the user types one by one all the matchings in a techniques application.

- techniques library – the techniques library was defined based only on one example. Although many predicates can be generated from the same set of techniques, it seems unlikely that this set can be used in a significant number of models without refining and extending it.

In order to overcome these limitations and to investigate other aspects of the problem, in the next section we suggest some extensions to the present work.

5.3 Further Work

5.3.1 User Interface

As mentioned before, our system needs a more friendly user interface, which could be obtained using an external Prolog library with graphical resources which are required to construct a Windows-like environment. This could lead to the following improvements:

- Our system would be easier to use – the user can do most tasks only clicking buttons in selection panels. This could be very useful in indicating the matches between variables in a technique and the constants and variables in input code, during techniques editing.

- A more informed environment could be produced – it is easier for the user to know all relevant information about the process through display windows.

- A help option could be available – this can supply guidance on *how* to use every part of the system, as well as *what* the rôle of a task in the whole process is.

In addition, the methodology of implementation adopted allows us to implement easily a user-friendly interface as described above.

5.3.2 Expanding to a Third Level

The techniques editor described here allows the user to build a predicate in terms of techniques. These techniques are stored in a library which was formed in an external process. Nevertheless, the task executed by our system is not an isolated action, rather it is part of a dynamic knowledge-based process as described in Chapter 3. For this reason, the final code generated must itself be used to measure the adequacy of the techniques library utilised. Thus, our system could be integrated with a tool to act over the techniques library, adjusting it to represent effectively the knowledge acquisition process. [Vasconcelos 94b] describes a Prolog Meta-Editor which is used to formalise an expert's knowledge. By extending our system to a three-level approach, incorporating a techniques meta-editor, we shall be able to represent and to manipulate more effectively the different sorts of knowledge involved in ecological modelling.

5.3.3 Guidance

At the moment no guidance is given to the user, during techniques editing. However, independently from the description of a specific model, the techniques editor can give some suggestions, based simply on the features of every technique and a record of previous techniques selected – a sort of *program history* as presented in [Vargas-Vera *et al.* 93a]. For example, if 'recursion over time points' was the skeleton adopted, techniques which defines (an additional) base case or those which include some calculations are more likely to be applied over the code, while techniques with another kind of recursion normally are very unlikely to be used.

5.3.4 Program Specification

In our system, we adopted a very simple description of every predicate which is required to be defined, the pair *name* and *arity*. A more complete description, with details about each argument, as well as the general goal of a predicate, could be useful at least in two aspects:

1. Guidance – by allowing the user to know more about what is expected from a predicate. In addition, the same information can be used by the system to give details about which techniques are more suitable to be used.

2. Generation Control – by allowing more accurate forms of check if a predicate, which must correspond to a description, has already been defined.

Enhancements of program specification may result from further investigation about how to transform internalised ecologist's knowledge in a formal specification – represented as task II in Figure 3–1.

Bibliography

[Bowles & Brna 93] A. Bowles and P. Brna. Programming plans and programming techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Word Conference on Artificial Intelligence in Education*, pages 378–385. Prentice-Hall, 1993.

[Bowles & Robertson 93] A. Bowles and D. Robertson. A case-based reasoning approach to supporting novice programmers. Internal note, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Bowles & Vasconcelos 93] A. Bowles and W. Vasconcelos. Characterizing prolog programming techniques. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Bowles 93] A. Bowles. A techniques editor for prolog novices. Internal note submitted for publication, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Bowles *et al.* 93] A. Bowles, D. Robertson, W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying prolog programming techniques. Research paper 641, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Brna *et al.* 91] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C.K. Looi, H. Pain, B. Smith, and M. Someren. Prolog programming techniques. *Instructional Science*, 20(2/3), 1991.

[Bundy 84] A. Bundy. Intelligent front ends. In J. Fox, editor, *State of the Art Report on Expert Systems*, pages 15–24. Pergamon Infotech, 1984.

[Durkin 94] J. Durkin. *Expert Systems – Design and Development.* Macmillan Pub. Co., 1994.

[Kirschenbaum *et al.* 89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for prolog programming. Technical report, Case Western Reserv University, 1989.

[Krebs 78] C.J. Krebs. *Ecology: the Experimental Analysis of Distribuition and Abundance.* Harper and Row, 1978.

[Laurière 90] J.L. Laurière. *Problem Solving and Artificial Intelligence.* Prentice Hall, 1990.

[Robertson 91] D. Robertson. A simple prolog techniques editor for novice users. In G.A. Wiggins, C. Mellish, and T. Duncan, editors, *3rd UK Annual Conference on Logic Programming*, pages 190–205. Berlin: Springer-Verlag, 1991.

[Robertson *et al.* 91] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M. Eschold. *Eco-Logic: Logic-Based Approaches to Ecological Modelling.* MIT Press, 1991.

[Vargas-Vera *et al.* 93a] M. Vargas-Vera, D. Robertson, and R. Inder. Combinining prolog program in a techniques editing system. Research paper 636, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Vargas-Vera *et al.* 93b] M. Vargas-Vera, W. Vasconcelos, and D. Robertson. Building large-scale prolog programs using a techniques editing system. Research paper 635, Department of Artificial Intelligence, University of Edinburgh, 1993.

[Vasconcelos 94a] W. W. Vasconcelos. Designing prolog programming techniques. In *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1994.

[Vasconcelos 94b] W. W. Vasconcelos. Extracting prolog programming techniques. In *Proceedings of the XI Brazilian Symposium on Artificial Intelligence*. 1994.

Appendix A

Techniques Editor

```
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------
%
%     PREDICATES TO TECHNIQUES EDITING
%
%     Author: Alberto Castro    -    Last Updating: 20/08/94
%
%-------------------------------------------------------------------------------
%-------------------------------------------------------------------------------

%-------------------------------------------------------------------------------
%     A SIMPLE PREDICATE INSTANTIATION
%-------------------------------------------------------------------------------
%
% unif_pred_name(+TechName,+ProgName,-Code).
%
unif_pred_name(TechName,ProgName,Code) :-
  technique(TechName,Code),
  Code = [_,[[ProgName|_]|_]|_],
  show_program(Code), !.
unif_pred_name(TechName,_,_) :-
  nl, write('## Error - '),
  write(TechName),write(' is not defined'),nl.



%-------------------------------------------------------------------------------
%     INCLUSION OF AN ARGUMENT IN A CODE
%-------------------------------------------------------------------------------
%
% include_argument(+ArgName,+Code,-NewCode).
%
include_argument(ArgName,[ IntVar | Code ],
                         [ NewIntVar | NewCode ] ) :-
  Code = [[[_|Params]|_]|_],
  length(Params,Arid),
```

```
  show_param(ArgName),
  inc_argument(ArgName,Code,IntVar,Arid,NewCode,NewIntVar),
  nl,write(' ========= New  ==========='),
  show_program([NewIntVar|NewCode]),!.


show_param(ArgName) :-
    var(ArgName),
    write('## Including a variable in the parameters.'),nl, !.
show_param(ArgName) :-
    write('## Including a new parameter: '),write(ArgName),write('.'),nl.


inc_argument(_,[],IntVar,_,[],IntVar).
inc_argument(ArgName,[[[ProgName|Parameters]|Body]|RestClauses],IntVar,Arid,
    [[[ProgName,ArgName|Parameters]|Body]|RestNewCls],NewIntVar) :-
    inc_argument(ArgName,RestClauses,IntVar,Arid,RestNewCls,NewIntVar).
inc_argument(ArgName,
    [rec([[ProgName|Parameters]|Body])|RestClauses],IntVar,Arid,
    [rec([[ProgName,ArgName|Parameters]|NewBody])|RestNewCls],
    NewIntVar) :-
    inc_arg_body(ArgName,ProgName,Arid,Body,IntVar,NewBody,Var2),
    inc_argument(ArgName,RestClauses,Var2,Arid,RestNewCls,NewIntVar).


inc_arg_body(_,_,_,[],IntVar,[],IntVar).
inc_arg_body(ArgName,ProgName,Arid,[[ProgName|Parameters]|RestProc],IntVar,
      [[ProgName,ArgName|Parameters]|RestNewProc],NewIntVar) :-
      unchanged_arg(ArgName),
      inc_arg_body(ArgName,ProgName,Arid,RestProc,
                   IntVar,RestNewProc,NewIntVar).
inc_arg_body(ArgName,ProgName,Arid,[[ProgName|Parameters]|RestProc],IntVar,
      [[ProgName,X|Parameters]|RestNewProc],NewIntVar) :-
      inc_arg_body(ArgName,ProgName,Arid,RestProc,
                   [X|IntVar],RestNewProc,NewIntVar).
inc_arg_body(ArgName,ProgName,Arid,[Proc|RestProc],IntVar,
      [Proc|RestNewProc],NewIntVar) :-
      inc_arg_body(ArgName,ProgName,Arid,RestProc,
                   IntVar,RestNewProc,NewIntVar).

unchanged_arg(ArgName) :-
        var(ArgName), nl,
        write('<> Has this variable, (always) the same value as in '),
        write('the `head''? '),nl,
        write('   Please, (yes/no)'),nl,nl,
        read_yes_no(Answer),
        ( Answer=yes,! ; Answer=no,!,fail ).
unchanged_arg(_ArgName).



%-----------------------------------------------------------------------------
```

```
%     TECHNIQUE APPLICATION
%-------------------------------------------------------------------------------
%
% add_technique(+Tech,+Code,-NewCode).
%
add_technique(Tech,Code,[NewIntVar|NewCls]) :-
  match_params(Code,Tech,[TechIntVar|TechCls],IntVar),
  append(IntVar,TechIntVar,NewIntVar),
  Code = [_|Cls],
  add_clause(Cls,TechCls,NewCls),
  nl,write(' ========= New  ==========='),
  show_program([NewIntVar|NewCls]),!.


%-------------------------------------------------------------------------------
% match_params(+Code,+Tech,-TechCode).
%     From the original code and the technique definition, make the matching
%     between the technique variables and the variables in the original code
%
match_params([IntVar,[[_NameCode|ParamCode]|_BodyCode]|_RestClsCode],Tech,
    [TechIntVar,[[NameTech|ParamTech]|BodyTech]|RestClsTech],NewVar) :-
    technique(Tech,[TechIntVar,[[NameTech|ParamTech]|BodyTech]|RestClsTech]),
    append(ParamCode,IntVar,CodeVar),
    read_matches([IntVar,[[_NameCode|ParamCode]|_BodyCode]|_RestClsCode],
                 [TechIntVar,[[NameTech|ParamTech]|BodyTech]|RestClsTech],
                 MatPos),
    make_matchings(CodeVar,ParamTech,MatPos,IntVar,NewVar),!.


%-------------------------------------------------------------------------------
% read_matches(+Code,+Tech,-Matches).
%     Build a list of Matches. Each element is a integer which points to the
%     correspondent parameter/variable in the code. So, for example: [5,2,0]
%     means that the argument 1 in the tech definition must be matched with
%     the 5th element in the list of code parameters, argument 2 in the tech
%     must be matched with 2nd elem in the code and the 3rd argument in the
%     tech has no correspondent in the code (must be a new variable).
%
%
read_matches([IntVar,[[NameCode|ParamCode]|BodyCode]|RestClsCode],
    [TechIntVar,[[NameTech|ParamTech]|BodyTech]|RestClsTech],
    Matches) :-
    show_program([IntVar,[[NameCode|ParamCode]|BodyCode]|RestClsCode],
                 CodeVar),
    show_program([TechIntVar,[[NameTech|ParamTech]|BodyTech]|RestClsTech],
                 TechVar),
    append(ParamCode,IntVar,OriginalCodeVar),
    change_value(ParamTech,TechVar,NewVar,'$_'), nl,
    write('<> Inform the variable/argument in the Code,'),nl,
    write('   to matching with the technique parameters.'),nl,
    write('   <Return> when none.'),nl,
    length(CodeVar,Nvar),
```

```
    ask_user(NewVar,CodeVar,Nvar,OriginalCodeVar,Matches).

%
% change_value(+ParamTech,+TechVar,-NewVar,+Symbol). Symbol = 'v_'|'$_'
%     Build a list joining parameters and internal variables.
%     Variables are represented by Symbol+sequential order. Ex: v_4 , $_12
%
change_value([],_,[],_).
change_value([Var|RestVar],VarEnv,[VarName|NewVar],SymVar) :-
      var(Var),
      belong(Var,VarEnv,Seq),
      name(SymVar,ListSymVar),
      name(Seq,ListNum),
      append(ListSymVar,ListNum,ListVar),
      name(VarName,ListVar),
      change_value(RestVar,VarEnv,NewVar,SymVar).
change_value([Var|RestVar],VarEnv,[Var|NewVar],SymVar) :-
      change_value(RestVar,VarEnv,NewVar,SymVar).

%
% ask_user(+NewVar,+CodeVar,+Nvar,+OriginalCodeVar,-Matches).
%
ask_user([],_,_,_,[]).
ask_user([Par|RestPar],CodeVar,Nvar,OriginalCodeVar,[P|Matches]) :-
      is-param(Par),
      read_one(Par,Selected),
      check_one(Selected,CodeVar,Nvar,OriginalCodeVar,P),
      ask_user(RestPar,CodeVar,Nvar,OriginalCodeVar,Matches), !.
ask_user([Par|RestPar],CodeVar,Nvar,OriginalCodeVar,[Par|Matches]) :-
      ask_user(RestPar,CodeVar,Nvar,OriginalCodeVar,Matches).

read_one(Par,Selected) :-
        write('<> '),write(Par),write('?'),nl,nl,
        repeat,
        read0(Selected).

check_one(' ',_CodeVar,_Nvar,_OriginalCodeVar,0).
check_one(Selected,CodeVar,Nvar,OriginalCodeVar,P) :-
        is-var(Selected),
        name(Selected,[118,95|Numbers]),
        name(N,Numbers),
        N =< Nvar,
        member([OrigVar,N],CodeVar),
        pos(OrigVar,OriginalCodeVar,1,P), !.
check_one(Selected,_CodeVar,_Nvar,OriginalCodeVar,P) :-
        member2(Selected,OriginalCodeVar),
        pos(Selected,OriginalCodeVar,1,P),!.
check_one(_,_,_,_,_) :-
        write('## Error - Value is not valid. Try again...'),nl,
        fail.

get_original(Selected,CodeVar,OrigVar) :-
```

```
        name(Selected,[118,95|Numbers]),
        name(N,Numbers),
        belong(OrigVar,CodeVar,N).

pos(OrigVar,[X|_],P,P) :-
          OrigVar == X.
pos(OrigVar,[_|Rest],P1,P) :-
          P2 is P1+1,
          pos(OrigVar,Rest,P2,P).



%
% make_matchings(+CodeVar,+ParamTech,+MatPos,+IntVar,-NewVar).
%
make_matchings(_,_,[],IntVar,IntVar).
make_matchings(CodeVar,[Par|RestPar],[MatPos|RestMatPos],IntVar,NewVar) :-
    var(Par),
    attrib(Par,MatPos,CodeVar,1,IntVar,Inter),
    make_matchings(CodeVar,RestPar,RestMatPos,Inter,NewVar).
make_matchings(CodeVar,[_Par|RestPar],MatPos,IntVar,NewVar) :-
    make_matchings(CodeVar,RestPar,MatPos,IntVar,NewVar).

attrib(Par,0,_,_,IntVar,[Par|IntVar]).
attrib(_,_,[],_,IntVar,IntVar).
attrib(Par,Pos,[Par|_RestParam],Pos,IntVar,IntVar).
attrib(Par,Pos,[_Param|RestParam],Order,IntVar,NewVar) :-
      NewOrder is Order+1,
      attrib(Par,Pos,RestParam,NewOrder,IntVar,NewVar).



%
% add_clause(+Cls,+TechCls,-NewCls).
%
add_clause([],[],[]).
add_clause([rec([Head|Body])|RestCls],[[_TechHead|TechBody]|TechRestCls],
           [rec(NewCls)|NewRestCls]) :-
    append([Head|Body],TechBody,NewCls),
    add_clause(RestCls,TechRestCls,NewRestCls),!.
add_clause([[Head|Body]|RestCls],[[_TechHead|TechBody]|TechRestCls],
           [NewCls|NewRestCls]) :-
    append([Head|Body],TechBody,NewCls),
    add_clause(RestCls,TechRestCls,NewRestCls).



%-----------------------------------------------------------------------------
%     PRETTY PRINTING OF THE CODE
%-----------------------------------------------------------------------------
%
% show_program(+WholeCode).         Just print a code
```

```
% show_program(+WholeCode,-VarEnv). Print a code and creates a Var environment
%

show_program(WholeCode) :-
  show_program(WholeCode,_).


show_program([IntVar,[[ProgName|ExtVar]|Body1]|RestCls],VarEnv) :-
  \+ var(ProgName),
  nl,write(' ========== Code ==========='),nl,
  show_program2([[[ProgName|ExtVar]|Body1]|RestCls],[],VarEnv,'v_'),
     write(' --- Arguments/Variables ---'),nl,
  show_vars(ExtVar,IntVar,VarEnv,'v_'),
     write(' ==========================='),nl, !.
show_program([IntVar,[[ProgName|ExtVar]|Body1]|RestCls],VarEnv) :-
  var(ProgName),
  technique(TechName,[IntVar,[[ProgName|ExtVar]|Body1]|RestCls]),
  ProgName = 'Tech',
  print_head(TechName,'=',27),
  show_program2([[[ProgName|ExtVar]|Body1]|RestCls],[],VarEnv,'$_'),
     write(' --- Arguments/Variables ---'),nl,
  show_vars(ExtVar,IntVar,VarEnv,'$_'),
     write(' ==========================='),nl, !.    % 27 =s

print_head(TechName,Char,N) :-
    name(TechName,L),
    length(L,Num),
    Quant is (N-Num-2) // 2,
    Quant1 is Quant-3,
    write(' '),
    writeChar(Char,3),
    writeChar(Char,Quant1),
    write(' '), write(TechName), write(' '),
    writeChar(Char,Quant),
    Rest is N - ((Quant*2)+2+Num),
    writeChar(Char,Rest),nl.
writeChar(_Char,0).
writeChar(Char,N) :-
      write(Char),
      N2 is N-1,
      writeChar(Char,N2).


show_program2([],VarSoFar,VarSoFar,_).
show_program2([Clause|RestClauses],VarSoFar,VarEnv,SymVar):-
  display_clause(Clause,VarSoFar,InterVar,SymVar),nl,
  show_program2(RestClauses,InterVar,VarEnv,SymVar).


display_clause(rec(Definition),VarSoFar,VarEnv,SymVar) :- % recursive clause
    display_clause(Definition,VarSoFar,VarEnv,SymVar).
display_clause([Head],VarSoFar,VarEnv,SymVar):-
```

```
    display_functor(Head,VarSoFar,VarEnv,SymVar),
    write('.').
display_clause([Head|Body],VarSoFar,VarEnv,SymVar):-
    display_functor(Head,VarSoFar,VarEnvSoFar,SymVar),
    write(' :- '),
    nl, write('    '),
    display_body(Body,VarEnvSoFar,VarEnv,SymVar).


display_functor([Functor|Arguments],VarSoFar,VarEnv,SymVar):-
      write(Functor),
      write('('),
      display_arguments(Arguments,VarSoFar,VarEnv,SymVar),
      write(')').


display_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
      \+ atomic(Arg),
      \+ var(Arg),
      display_functor(Arg,VarSoFar,VarEnvSoFar,SymVar),
      display_rest_arguments(RestArgs,VarEnvSoFar,VarEnv,SymVar).
display_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
      var(Arg),
      get_value(Arg,Value,VarSoFar,VarEnvAfter),
      write(SymVar),write(Value),
      display_rest_arguments(RestArgs,VarEnvAfter,VarEnv,SymVar).
display_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
      write(Arg),
      display_rest_arguments(RestArgs,VarSoFar,VarEnv,SymVar).


display_rest_arguments([],VarSoFar,VarSoFar,_).
display_rest_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
        \+ atomic(Arg),
        \+ var(Arg),
        write(','),
        display_functor(Arg,VarSoFar,VarEnvSoFar,SymVar),
        display_rest_arguments(RestArgs,VarEnvSoFar,VarEnv,SymVar), !.
display_rest_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
        var(Arg),
        get_value(Arg,Value,VarSoFar,VarEnvAfter),
        write(','),
        write(SymVar),write(Value),
        display_rest_arguments(RestArgs,VarEnvAfter,VarEnv,SymVar).
display_rest_arguments([Arg|RestArgs],VarSoFar,VarEnv,SymVar):-
        write(','),
        write(Arg),
        display_rest_arguments(RestArgs,VarSoFar,VarEnv,SymVar).


get_value(Var,Value,VarSoFar,VarSoFar) :-
          belong(Var,VarSoFar,Value), !.
```

```
get_value(Var,N,VarSoFar,[[Var,N]|VarSoFar]) :-
          length(VarSoFar,L),
          N is L + 1.


display_body([Term],VarSoFar,VarEnv,SymVar):-
      display_functor(Term,VarSoFar,VarEnv,SymVar),
      write('.').
display_body([Term|RestTerms],VarSoFar,VarEnv,SymVar):-
      display_functor(Term,VarSoFar,VarEnvSoFar,SymVar),
      write(','),
      nl, write('   '),
      display_body(RestTerms,VarEnvSoFar,VarEnv,SymVar).


show_vars(ExtVar,IntVar,VarEnv,SymVar) :-
    append(ExtVar,IntVar,Var),
    change_value(Var,VarEnv,NewVar,SymVar),
    write(NewVar),nl, !.



%--------------------------------------------------------------------------------
%     AUXILIAR DEFINITIONS
%--------------------------------------------------------------------------------

append([],L,L).
append([X|Xs],L,[X|Ys]):-append(Xs,L,Ys).


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

member2(X,[H|_]) :- X == H.
member2(H,[_|T]) :- member2(H,T).


belong(Elem,[[Var,Seq]|_],Seq) :-
  Elem == Var.
belong(Elem,[_|T],Seq) :-
  belong(Elem,T,Seq).


is-var(Symbol) :-
        atomic(Symbol),
        name(Symbol,[118,95|Numbers]),
        only-numbers(Numbers).
only-numbers([]).
only-numbers([First|Rest]) :-
        First > 47,
        First < 58,
        only-numbers(Rest).
```

```
is-param(Symbol) :-
        atomic(Symbol),
        name(Symbol,[36,95|Numbers]),
        only-numbers(Numbers).
```

Appendix B

Generation Control

```
%------------------------------------------------------------------------------
%------------------------------------------------------------------------------
%
%     GENERATION CONTROL
%
%     Author: Alberto Castro    -    Last Updating: 20/08/94
%
%------------------------------------------------------------------------------
%------------------------------------------------------------------------------

%------------------------------------------------------------------------------
%     INITIAL DEFINITIONS
%------------------------------------------------------------------------------

start :-
  nofileerrors,
  clear,
  print_head,
  print_intro,
  menu(initial).

clear :-
  unix(shell(clear)).

print_head :-
  write('=================================================================='),nl,
  write('                Techniques-Based Program Generator     '),nl,
  write('                          Version 1.0      '),nl,
  write('                  by Alberto Castro - Aug/94     '),nl,
  write('=================================================================='),
  nl,nl,nl.

print_intro :-
  write('<> This system was developed to assist you in the manipulation of'),nl,
  write('   Prolog Techniques, with the goal of program generation  in  an'),nl,
  write('   ecologic domain.'),nl,nl,
```

```
  write('   Details about use and references, are in the file README.'),nl,
  nl,nl.



%------------------------------------------------------------------------------
%     STARTING A TBPG SESSION
%------------------------------------------------------------------------------

menu(initial) :-
  write('<> Select one of the options below:'),
  nl,nl,
  write('   1 - Start a new code generation'),
  nl,nl,
  write('   2 - Load a file with a previous generation'),
  nl,nl,
  write('   3 - Continue a generation started in this Prolog session'),
  nl,nl,
  write('   4 - End.'),
  nl,nl,
  read_option(Op,1,4),
  next(initial,Op,Next),
  clear,
  print_head,
  menu(Next).

%------------------------------------------------------------------------------

menu(new_code) :-
    retractall(goal_now(_)),
    retractall(goals_list(_)),
    retract_all_code,
    close_all_streams,
    write('   Please, inform the name and arity of the first predicate'),nl,
    write('   to be defined, or type ''default'' to system''s default.'),nl,nl,
    repeat,
    nl, write('<> Name of the predicate.'), nl,nl,
    read0(Name),
    ( atom(Name)
      ;
      write('## Error - Name is not an atom.'),nl,fail
    ),
    ( Name = 'default',
      default(D),
      assert(goal_now(D)),
      assert(goals_list([]))
      ;
      repeat,
      nl, write('<> Arity of the predicate.'), nl,nl,
      read0(Arity),
      ( number(Arity)
        ;
```

```
        write('## Error - Arity is not an integer.'),nl,fail
      ),
      assert(goal_now([Name,Arity])),
      assert(goals_list([]))
    ),
    !,
    clear,
    print_head,
    menu(start_tech).

%------------------------------------------------------------------------------

menu(load_file) :-
    load_file,
    redef_codes,
    continue,
    clear,
    print_head, !,
    menu(deal_goals).
menu(load_file) :-
    clear,
    print_head,
    menu(initial).
load_file :-
      write('<> Please, inform the name of the file.'),nl,nl,
      read0(File),
      atom(File),
      thereis(File),
      retractall(goal_now(_)),
      retractall(goals_list(_)),
      retract_all_code, !,
      [File].
load_file :-
      nl, write('## Error - File cannot be read.'), nl,
      continue,
      fail.
thereis(File) :-
        open(File,read,S),
        close(S).
retract_all_code :-
        code([_,[[Name|Params]|_]|_]),
        length(Params,Arity),
        list_vars(Arity,L),
        Term =.. [Name|L],
        retractall(Term),
        fail.
retract_all_code :-
        retractall(code(_)).

close_all_streams :-
        current_stream(_,_,S),
        close(S),
```

```
        fail.
close_all_streams.

redef_codes :-
      code(Code),
      store_code(Code),
      fail.
redef_codes.

%-------------------------------------------------------------------------------

menu(same_session) :-
    goals_list(_),
    goal_now(_), !,
    menu(deal_goals).
menu(same_session) :-
    write('   Sorry, the data needed are not present'),nl,
    write('   It''s not possible to continue the session'),nl,
    continue,
    clear,
    print_head,
    menu(initial).

%-------------------------------------------------------------------------------

menu(end) :-
    write('   Good bye for now...'),nl,nl,
    write('====================  End of Session  ========================='),nl,
    nl, fileerrors.



%-------------------------------------------------------------------------------
%     SELECTING A TECHNIQUE TO START GENERATION
%-------------------------------------------------------------------------------

menu(start_tech) :-
  msg_pred_name,
  write('<> Select one of the options below:'),
  nl,nl,
  write('   1 - Show the techniques set'),
  nl,nl,
  write('   2 - Start code generation'),
  nl,nl,
  write('   3 - Cancel.'),
  nl,nl,
  read_option(Op,1,3),
  next(start_tech,Op,Next),
  clear,
  print_head,
  menu(Next).
```

```
msg_pred_name :-
    goal_now([Name,Arity]),
    write('   Defining predicate '),
    write(Name), write('/'), write(Arity), write('.'), nl, nl, nl.

%------------------------------------------------------------------------------

menu(show_tech) :-
    show_all_techniques,
    continue,
    clear,
    print_head,
    menu(start_tech).

show_all_techniques :-
      technique(_,Code),nl,
      show_program(Code),
      fail.
show_all_techniques :-
      write('   ... And nothing else'),nl.

continue :-
      nl,write('<> Press <Return> to continue'),nl,
      read0(_Any).

%------------------------------------------------------------------------------

menu(start_gen) :-
    retractall(undo_(_)),
    ask_technique(Tech),
    \+ Tech = cancel,
    goal_now([Prog,_]),
    unif_pred_name(Tech,Prog,Code),
    continue,
    clear,
    print_head, !,
    menu(deal_code,Code).
menu(start_gen) :-
    clear,
    print_head,
    menu(start_tech).

ask_technique(Tech) :-
      write('<> Which technique you intend to use?'),nl,
      write('   <Return> to see the names list, or'),nl,
      write('   ''cancel'' to go back.'),nl, nl,
      repeat,
      read0(Tech),
      check_tech(Tech), !.

check_tech(' ') :-
        bagof(X,list_tech(X),List),
```

```
        write(List),nl, !, fail.
check_tech(Tech) :-
        technique(Tech,_), !.
check_tech(cancel) :- !.
check_tech(_) :-
        write('## Error - Please inform one name from the techiniques set'),
        nl,
        fail.

list_tech(X) :-
          technique(X,[_,[[Name|_]|_]|_]),
          var(Name).                % To avoiding the 'constant techniques'



%-------------------------------------------------------------------------------

menu(cancel_1) :-
    retractall(goal_now(_)),
    menu(initial).


%-------------------------------------------------------------------------------
%     MANIPULATING THE CODE
%-------------------------------------------------------------------------------

menu(deal_code,Code) :-
  msg_pred_name,
  write('<> Select one of the options below:'),
  nl,nl,
  write('   1 - Include arguments in the code'),
  nl,nl,
  write('   2 - Apply (add) technique to a code'),
  nl,nl,
  write('   3 - Show the code'),
  nl,nl,
  write('   4 - Save the code'),
  nl,nl,
  write('   5 - Undo'),
  nl,nl,
  write('   6 - Cancel.'),
  nl,nl,
  read_option(Op,1,6),
  next(deal_code,Op,Next),
  menu(Next,Code).

%-------------------------------------------------------------------------------

menu(include_arg,InCode) :-
    \+ var(InCode),
    check_n_args(InCode),
    show_program(InCode),
```

```
    ask_parameter(Par),
    include_argument(Par,InCode,OutCode),
    asserta(undo_(InCode)),
    continue,
    clear,
    print_head, !,
    menu(deal_code,OutCode).
menu(include_arg,Code) :-
    nl,write('## You''re trying to add more arguments than prevised'),nl,nl,
    write('<> Proceed? (yes/no)'),nl, nl,
    read_yes_no(Answer),
    Answer = yes,
    goal_now([Name,Arity]),
    NewArity is Arity + 1,
    retractall(goal_now(_)),
    assert(goal_now([Name,NewArity])), !,
    menu(include_arg,Code).
menu(include_arg,Code) :-
    clear,
    print_head,
    menu(deal_code,Code).

check_n_args([_IntVar,[[_ProgName|ExtVar]|_Body1]|_RestCls]) :-
      length(ExtVar,N),
      Next is N + 1,
      goal_now([_,Arity]),
      Arity >= Next.

ask_parameter(Par) :-
      nl,
      write('<> Please, inform the parameter to be added:'),nl, nl,
      read0(Par).

%------------------------------------------------------------------------------

menu(add_tech,InCode) :-
    \+ var(InCode),
    ask_technique(Tech),
    add_technique(Tech,InCode,OutCode),
    asserta(undo_(InCode)),
    continue,
    clear,
    print_head, !,
    menu(deal_code,OutCode).
menu(add_tech,InCode) :-
    clear,
    print_head,
    menu(deal_code,InCode).

%------------------------------------------------------------------------------

menu(undo,InCode) :-
```

```
    \+ var(InCode),
    undo_(LastCode),
    retract(undo_(LastCode)),
    show_program(LastCode),
    continue,
    clear,
    print_head, !,
    menu(deal_code,LastCode).
menu(undo,InCode) :-
    clear,
    print_head,
    menu(deal_code,InCode).

%-------------------------------------------------------------------------------

menu(show_code,Code) :-
    \+ var(Code),
    show_program(Code),
    continue,
    clear,
    print_head, !,
    menu(deal_code,Code).

%-------------------------------------------------------------------------------

menu(save_code,Code) :-
    \+ var(Code),
    arity_ok(Code),
    assert(code(Code)),
    store_code(Code),
    goals_list(InCode),
    check_code(Code,InCode,OutCode),
    retractall(goals_list(_)),
    assert(goals_list(OutCode)),
    nl,write('<> Code is now in Prolog working memory'),nl,
    continue,
    clear,
    print_head, !,
    menu(deal_goals).
menu(save_code,Code) :-
    clear,
    print_head,
    menu(deal_code,Code).

arity_ok([_IntVar,[[_ProgName|ExtVar]|_Body1]|_RestCls]) :-
      length(ExtVar,N),
      goal_now([_,Arity]),
      Arity = N.
arity_ok([_IntVar,[[ProgName|ExtVar]|_Body1]|_RestCls]) :-
      nl,write('## Code to save has fewer arguments than the expected'),nl,nl,
      write('<> Proceed? (yes/no)'),nl,
      read_yes_no(Answer),
```

```
      Answer = yes,
      length(ExtVar,New),
      retractall(goal_now(_)),
      assert(goal_now([ProgName,New])).

store_code(Code) :-
      \+ var(Code),
      store_pred(Code),
      assert_pred, !.

store_pred(Code) :-
        open('eco.tmp',write,S),
        set_output(S),
        show_program3(Code),
        set_output(user_output),
        close(S).
show_program3([_IntVar,[[ProgName|ExtVar]|Body1]|RestCls]) :-
          \+ var(ProgName),
          show_program2([[[ProgName|ExtVar]|Body1]|RestCls],[],_,'X'), !.

assert_pred :-
        open('eco.tmp',read,S),
        read_ass(S).
read_ass(S) :-
          read(S,X),
          \+ X = end_of_file,
          assert(X),
          read_ass(S).
read_ass(_).

retract_pred :-
        open('eco.tmp',read,S),
        read_ret(S).
read_ret(S) :-
          read(S,X),
          \+ X = end_of_file,
          retract(X),
          read_ass(S).
read_ret(_).

%-------------------------------------------------------------------------------

menu(cancel_2,_) :-
  clear,
  print_head,
  menu(initial).



%-------------------------------------------------------------------------------
%     NEXT STEP IN THE GENERATION PROCESS
%-------------------------------------------------------------------------------
```

```
menu(deal_goals) :-
  write('   Selecting a new predicate to define'),nl,nl,
  write('<> Select one of the options below:'),
  nl,nl,
  write('   1 - Define the next predicate in the list'),
  nl,nl,
  write('   2 - Show list of predicates to be defined'),
  nl,nl,
  write('   3 - Select any predicate in the list'),
  nl,nl,
  write('   4 - Show all predicates already defined'),
  nl,nl,
  write('   5 - End.'),
  nl,nl,
  read_option(Op,1,5),
  next(deal_goals,Op,Next),
  menu(Next).

%------------------------------------------------------------------------------

menu(next_goal) :-
    goals_list([Next|Rest]),
    retractall(goal_now(_)),
    assert(goal_now(Next)),
    retractall(goals_list(_)),
    assert(goals_list(Rest)), !,
    chk_const_tech(Next).
menu(next_goal) :-
    clear,
    print_head,
    menu(deal_goals).


% chk_const_tech([+Predicate,+Arity]).
%     If Predicate IS NOT a constant tech, starts new definition.
%     Otherwise, proceeds the definition of a constant technique, and returns
%                to deal_goals menu.
%
chk_const_tech([Pred,Arity]) :-
      technique(Pred,[IntVar,[[Name|Args]|RestDef1]|RestCls]),
      \+ var(Name),
      Name = Pred,
      length(Args,Arity), !,
      clear,
      print_head,
      nl,write('<> The predicate '),
      write(Name), write('/'), write(Arity),
      write(' is a predicate of basic definition.'),nl,
      menu(save_code,[IntVar,[[Name|Args]|RestDef1]|RestCls]).
chk_const_tech(_) :-
    clear,
```

```
    print_head,
    menu(start_tech).



%-----------------------------------------------------------------------------

menu(show_goals) :-
    goals_list(L), nl,
    write(L),nl,
    continue,
    clear,
    print_head,
    menu(deal_goals).

%-----------------------------------------------------------------------------

menu(pick_goal) :-
    goals_list(List),
    \+ List = [],
    ask_goal(Goal),
    \+ Goal = cancel,
    retractall(goal_now(_)),
    assert(goal_now(Goal)),
    comp([Goal],List,NewList),
    retractall(goals_list(_)),
    assert(goals_list(NewList)), !,
    chk_const_tech(Goal).
menu(pick_goal) :-
    clear,
    print_head,
    menu(deal_goals).

ask_goal(Goal) :-
      nl,write('<> Which predicate you intend to define now?'),nl,
      write('   <Return> to see the list, or ''cancel'' to go back.'),nl, nl,
      repeat,
      read0(Answer),
      check_goal(Answer,Goal), !.

check_goal(' ',_) :-
        goals_list(L),
        write_goals_list(L),
        nl, !, fail.
check_goal(Answer,[Answer,Arity]) :-
        goals_list(List),
        member([Answer,Arity],List), !.
check_goal(cancel,_) :- !.
check_goal(_,_) :-
        write('## Error, try again'), nl, fail.
write_goals_list([]) :-
          write('[].').
```

```
write_goals_list([[Name,_]]) :-
          write(Name),write('.').
write_goals_list([[Name,_]|Rest]) :-
          write(Name),write(','),
          write_goals_list(Rest).

%comp(+L1,+L2,-Complement)
%    Complement of L1 with relation to L2 or L2-L1 (L1 must be a sublist of L2).
%
comp(_,[],[]).
comp(L1,[H2|T2],[H2|Complement]) :-
        \+ member(H2,L1),
        comp(L1,T2,Complement), !.
comp(L1,[_|T2],Complement) :-
        comp(L1,T2,Complement).

%------------------------------------------------------------------------------

menu(show_previous) :-
    clause(code(_),_),
    show_all_preds,
    continue,
    clear,
    print_head, !,
    menu(deal_goals).
menu(show_previous) :-
    write('   No predicates salved'), nl,
    continue,
    clear,
    print_head,
    menu(deal_goals).

show_all_preds :-
      code(Code),
      show_program(Code),
      fail.
show_all_preds :-
      write('   ... And nothing else'),nl.

%------------------------------------------------------------------------------

menu(end2) :-
    clear,
    print_head,
    write('<> Do you want to save the information so far? (yes/no)'),nl, nl,
    read_yes_no(A),
    ( A = yes,
      save_all
      ;
      A = no
    ), !, nl,
    menu(end).
```

```
save_all :-
      nl,write('<> Please, inform the name of file'),nl, nl,
      read0(File),
      atom(File),
      open(File,write,S),
      set_output(S),
      goal_now(X),
       write(':- assert(goal_now('),
       write(X),
       write(')).'), nl,
      goals_list(Y),
       write(':- assert(goals_list('),
       write(Y),
       write(')).'), nl,
      print_all_codes,
      set_output(user_output),
      close(S).
print_all_codes :-
        code(X),
         write(':- assert(code('),
         write(X),
         write(')).'),nl,
        fail.
print_all_codes.


%-------------------------------------------------------------------------------
%     MENU SEQUENCE
%-------------------------------------------------------------------------------

next(initial,1,new_code).
next(initial,2,load_file).
next(initial,3,same_session).
next(initial,4,end).
next(start_tech,1,show_tech).
next(start_tech,2,start_gen).
next(start_tech,3,cancel_1).
next(deal_code,1,include_arg).
next(deal_code,2,add_tech).
next(deal_code,3,show_code).
next(deal_code,4,save_code).
next(deal_code,5,undo).
next(deal_code,6,cancel_2).
next(deal_goals,1,next_goal).
next(deal_goals,2,show_goals).
next(deal_goals,3,pick_goal).
next(deal_goals,4,show_previous).
next(deal_goals,5,end2).
```

```
%-----------------------------------------------------------------------------
%     AUX SET 1 - INTERFACE's READ STUFF
%-----------------------------------------------------------------------------
%
% read0(?Var)  -  Similar to read/1, but doesn't need '.' to signal eod.
%
read0(X) :-
  word(W),
  transform(W,X), !.

word(W) :-                              % Gets the data as a list of chars.
    get0(C),
    chars(32,C,W).
chars(46,C,[]) :-                       % User insists in the '.' at end :-)
      end(C), !.
chars(Previous,C,[Previous]) :-         % Start list with last char.
      end(C), !.
chars(Previous,C,[Previous|Nexts]) :-
      \+ invalid(Previous),
      get0(NextC), !,
      chars(C,NextC,Nexts).
chars(_,C,Nexts) :-
      get0(NextC),
      chars(C,NextC,Nexts).
end(10). end(13).                        % 10=nl, 13=cr
invalid(32). invalid(9).                 % 32=space, 9=tab

transform([],'').                        % <Return> pressed
transform([32],' ').                     % Space(s)
transform([First|Rest],X) :-             % Number
    number_(First),
    name(X,[First|Rest]).
transform([First|_],_) :-                % Variable
    ( upp_letter(First) ; ( First = 95 ) ), !.
transform([First|Rest],X) :-             % Atom
    valid_sequence([First|Rest]), !,
    name(X,[First|Rest]).
transform([First|Rest],X) :-             % List or Term
    name(Term,[First|Rest]),
    open('io.tmp',write,S1),
    write(S1,Term),write(S1,'.'),nl(S1),
    close(S1),
    open('io.tmp',read,S2),
    read(S2,X),                          % IF syntax error, THEN prg abort(!)
    close(S2).
valid_sequence([]).
valid_sequence([H|T]) :-
      low_letter(H), !,
      valid_sequence(T).
valid_sequence([H|T]) :-
      upp_letter(H), !,
      valid_sequence(T).
```

```
valid_sequence([H|T]) :-
      number_(H), !,
      valid_sequence(T).
valid_sequence([95|T]) :-
      valid_sequence(T).
low_letter(H) :-      H>96 , H<123.
upp_letter(H) :-      H>64 , H<91.
number_(H) :-         H>47 , H<58.

%
% read_yes_no(?Var)   Exactly what the name says
%
read_yes_no(X) :-
  read0(X),
  ( X = 'yes'
    ;
    X = 'no' ), !.
read_yes_no(X) :-
  write('## Error - Please, enter ''yes'' or ''no'''),nl,
  read_yes_no(X).

%
% read_option(-Var,+MinNumber,+MaxNumber)
%
read_option(Op,Min,Max) :-
  repeat,
  write('<> Inform a number between '),
  write(Min),write('-'),write(Max),write(' (inc).'),nl,nl,
  read0(Op),
  number(Op),
  Op >= Min,
  Op =< Max, !.



%-----------------------------------------------------------------------------
%     AUX SET 2 - TO SEARCHING ALL NON-DEFINED PREDICATES IN A CODE
%-----------------------------------------------------------------------------

%check_code(Code,GoalsIn,GoalsOut).
%
% Code: Programming code to be "checked" if all sub-goals have been defined
% GoalsIn: Queue of predicates to be defined -before- the goal being dealt,
%          in the structure: [ [name_1,arity_1] , ... , [name_N,arity_N] ].
% GoalsOut: New queue of predicates to be defined.

check_code([_IntVar,[[ProgName|ExtVar]|Body1]|RestCls],GoalsIn,GoalsOut) :-
  count(ExtVar,Arity),
  check_def(ProgName,Arity,
          [[[ProgName|ExtVar]|Body1]|RestCls],GoalsIn,GoalsOut).
```

```
check_def(_Prog,_Arity,[],GoalsIn,GoalsIn).
check_def(Prog,Arity,[[[Prog|_Params]|Body]|RestCls],GoalsIn,GoalsOut) :-
    check_body(Prog,Arity,Body,GoalsIn,NewIn),
    check_def(Prog,Arity,RestCls,NewIn,GoalsOut), !.
check_def(Prog,Arity,[rec([[Prog|_Params]|Body])|RestCls],
          GoalsIn,GoalsOut) :-
    check_def(Prog,Arity,[[[Prog|_Params]|Body]|RestCls],GoalsIn,GoalsOut).

check_body(_Prog,_Arity,[],GoalsIn,GoalsIn).
%check_body(Prog,Arity,[rec(Pred)|RestBody],GoalsIn,GoalsOut) :-
%      check_body(Prog,Arity,[Pred|RestBody],GoalsIn,GoalsOut),!.
check_body(Prog,Arity,[Pred|RestBody],GoalsIn,GoalsOut) :-
      check_pred(Prog,Arity,Pred,GoalsIn,NewIn),
      check_body(Prog,Arity,RestBody,NewIn,GoalsOut).


check_pred(_Prog,_Arity,[],GoalsIn,GoalsIn).
check_pred(Prog,Arity,[Prog|Terms],GoalsIn,GoalsOut) :-
        count(Terms,Arity),     % pred is the same in the head (prog).
        check_terms(Prog,Arity,Terms,GoalsIn,GoalsOut),!.
check_pred(Prog,Arity,[Pred|Terms],GoalsIn,GoalsOut) :-
        system([Pred|Terms]),   % pred is a built-in.
        check_terms(Prog,Arity,Terms,GoalsIn,GoalsOut),!.
check_pred(Prog,Arity,[Pred|Terms],GoalsIn,GoalsOut) :-
        count(Terms,C),
        defined(Pred,C),        % pred was defined by this program.
        check_terms(Prog,Arity,Terms,GoalsIn,GoalsOut),!.
check_pred(Prog,Arity,[Pred|Terms],GoalsIn,GoalsOut) :-
        count(Terms,C),
        member([Pred,C],GoalsIn),  % pred is in the list of goals.
        check_terms(Prog,Arity,Terms,GoalsIn,GoalsOut),!.
check_pred(Prog,Arity,[Pred|Terms],GoalsIn,GoalsOut) :-
        count(Terms,C),         % pred must be included in the list of goals.
        append(GoalsIn,[[Pred,C]],NewIn),
        check_terms(Prog,Arity,Terms,NewIn,GoalsOut).


check_terms(_Prog,_Arity,[],GoalsIn,GoalsIn).
check_terms(Prog,Arity,[Term|RestTerms],GoalsIn,GoalsOut) :-
          ( atomic(Term) ; var(Term) ),
          check_terms(Prog,Arity,RestTerms,GoalsIn,GoalsOut), !.
check_terms(Prog,Arity,[Term|RestTerms],GoalsIn,GoalsOut) :-
          check_pred(Prog,Arity,Term,GoalsIn,NewIn),
          check_terms(Prog,Arity,RestTerms,NewIn,GoalsOut).


defined(Pred,Arity) :-   % At moment, just check if there is ANY definition
  list_vars(Arity,L),    %    but must be extended to check if there is a
  Term =.. [Pred|L], !,  %    unifiable definition.
  clause(Term,_N).

list_vars(0,[]) :- !.
```

```
list_vars(N,[_Var|Rest]) :-
    N2 is N - 1,
    list_vars(N2,Rest).


%------------------------------------------------------------------------------
%     AUX SET 3 - CHECKING OF PROLOG BUILT-IN PREDICATES (Reduced)
%------------------------------------------------------------------------------

system([Pred|Terms]) :-
  count(Terms,Arity),
  system(Pred,Arity), !.
system2(Goal):-              % To be used with a non-list (normal)
  Goal =.. [Pred|Terms],     %                          structure.
  count(Terms,Arity),
  system(Pred,Arity), !.

system(Pred,Arity):-
        test_operator(Pred,Arity).
system(Pred,Arity):-
        system_test(Pred,Arity).
system(Pred,Arity):-
        io_predicate(Pred,Arity).
system(Pred,Arity):-
        misc(Pred,Arity).
system(Pred,Arity):-
        el_utility(Pred,Arity).

test_operator(\+,1).
test_operator(=,2).
test_operator(=..,2).
test_operator(==,2).
test_operator(>,2).
test_operator(>=,2).
test_operator(<,2).
test_operator(=<,2).
test_operator(\==,2).
test_operator(=:=,2).
test_operator(=\=,2).

system_test(atom,1).
system_test(atomic,1).
system_test(float,1).
system_test(integer,1).
system_test(number,1).
system_test(var,1).
system_test(nonvar,1).
system_test(ground,1).

io_predicate(read,1).
io_predicate(get,1).
io_predicate(get0,1).
```

```
io_predicate(write,1).
io_predicate(nl,0).
io_predicate(display,1).
io_predicate(put,1).

misc(is,2).
misc(setof,3).
misc(+,2).
misc(-,2).
misc(*,2).
misc(/,2).
or(X,Y) :- X ; Y.
misc(or,2).
and(X,Y) :- X , Y.
misc(and,2).
misc(length,2).

count(L,C):- count(L,0,C).    % counts the elements of list L and returns
                              % value in C
count([],SoFar,SoFar).
count([_|Rest],SoFar,Final):-
        SoFar1 is SoFar + 1,
        count(Rest,SoFar1,Final).
```