Domain-specific synthesis of simulation models

Alberto Castro* Dave Robertson* Robert Muetzelfeldt†

August 9, 1998

Abstract

The formal specification community has produced many languages but few structured design methods. Those which exist tend to be abstract, providing little guidance in tackling problems in particular domains. One way of devising domain-specific design methods is by reconstructing an example in the domain using the target method; then generalising the design structures to cover a class of designs in the domain; then building an environment in which these structures can more easily be re-applied to new problems. We demonstrate this approach using animal population dynamics models as the domain and Prolog techniques as the target method.

keywords: program synthesis, design methods, logic programming and Prolog, modelling environments

1 Introduction

Modelling is not always an easy task and in some domains it is particularly difficult. In ecology, for example, a modeller has to deal with peculiarities that makes it difficult to devise a model in a structured way. The complex interactions between a system's constituents and the different levels of organisations (individuals, populations, ecosystems, etc.) blurs the focus of interest and the units of study. Ideally, an ecological model should not only exhibit correct behaviour, but should also make explicit the criteria considered in its construction so it could be evaluated and augment experience in building other models. In the task of constructing those models, usually implemented as computer programs, such considerations should be carefully observed. However, programs intended as simulation models are still usually seen as "black boxes" which exhibit certain external behaviour, whilst it is very difficult to understand the mechanisms inside.

Apart from few alternative proposals [Muetzelfeldt 95], the common practice in ecology still is to define and implement every model "from scratch" and modelling decisions and implementation aspects are mingled according to modeller's expertise. That process is costly and especially obstructive to novices

*Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN – UK. e-mail: *albertoc@dai.ed.ac.uk*; *dr@dai.ed.ac.uk*

†Institute of Ecology and Resource Management, University of Edinburgh, The Kings Buildings, Mayfield Road, Edinburgh EH9 3JU – UK. e-mail: *r.muetzelfeldt@ed.ac.uk*

with little practice in modelling and/or programming. The logic paradigm may provide a clear separation between the axioms defining the assumptions and the inference methods used in a model, and that could make it easier to evaluate, understanding and emulate the modelling process.

[Robertson *et al.* 91] described *Eco-Logic*, a project which introduced logic-based approaches to tackle these issues in the domain of ecology. One of the results of that project was the *EL* system, a tool to generate a model using instantiable pieces of Prolog code called *program schemata.* But in practice schemata are limited on flexibility, assembling groups of parameterised predicates. Another method, known as *Prolog Programming Techniques* (PPT), represents standard patterns for constructing individual predicates [Kirschenbaum *et al.* 89]. *Techniques editing* is an established general method for addressing

Apart from few alternative proposals [Muetzelfeldt 95], the common practice in ecology still is to define and implement every model "from scratch" and modelling decisions and implementation aspects are mingled according to modeller's expertise. That process is costly and especially obstructive to novices with little practice in modelling and/or programming. The logic paradigm may provide a clear separation between the axioms defining the assumptions and the inference methods used in a model, and that could make it easier to evaluate, understanding and emulate the modelling process.

[Robertson *et al.* 91] described *Eco-Logic*, a project which introduced logic-based approaches to tackle these issues in the domain of ecology. One of the results of that project was the *EL* system, a tool to generate a model using instantiable pieces of Prolog code called *program schemata.* But in practice schemata are limited on flexibility, assembling groups of parameterised predicates. Another method, known as *Prolog Programming Techniques* (PPT), represents standard patterns for constructing individual predicates [Kirschenbaum *et al.* 89]. *Techniques editing* is an established general method for addressing program design at a more fine grained level and earlier research has dealt with different tasks within the programming domain[Bowles *et al.* 94, Vasconcelos 93, Vargas-Vera *et al.* 93, Bowles 94, Robertson 91].

However, there have been no experiments in tailoring the general methods of techniques editing to the specific demands of a domain of application. This paper describes our attempt to do this, by building a system named TeMS (Techniques-based Model Synthesiser). We describe how PPT, within a domain-dependent modelling framework, can be used for model generation. We have used as starting-point a model taken from the ecological modelling literature and from it, defined domain-dependent PPTs which were used by a modelling framework to generate models from user specifications. We address the construction of models (in a very specific domain) by automated synthesis and validate it using conventional human model construction as a frame of reference. Figure 1 is a diagrammatic representation of the issues discussed here – rectangles stand for precisely defined concepts (as opposed to informal ones, represented by curvy boxes), processes are written in italics.

The rest of this paper is structured as follows: Section 2 reviews the use of Prolog Programming Techniques and Program Schemata in program construction and puts our work in the context of those approaches. Section 3 describes the reconstruction of a model from the ecological modelling literature, its use as a case-study to analyse the design decisions as well as the programming patterns (techniques and schemata) used in models of the same sort. Section 4 explains those structures and Section 5 shows how they are used to compose a model. The last section summarises the work.

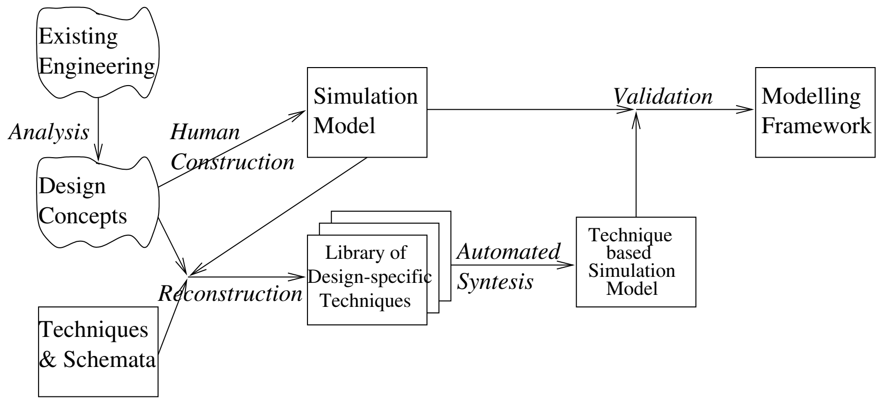


Figure 1: Defining a Techniques-based Modelling Framework

2 Techniques Editing and Program Schemata

2.1 Prolog Programming Techniques

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 having a commonly understood meaning at the code level, they often lack precise definition and for this reason, are usually illustrated by giving examples of predicates containing instances of them.

[Sterling & Kirschenbaum 93] proposed a now widely accepted approach to develop logic programs, called *stepwise enhancement*. 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 choose the basic control flow needed to solve the problem and embody it in a skeleton. Usually this is done by selecting a skeleton from a previously defined library.

2. Over this skeleton 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) partial program which allows to repeat the process until the final program has been developed.

To illustrate this method, take the skeleton `traverse` which traverses a list partitioning the control of flow into several branches according to some criteria,

[1] Sterling uses the term *technique*, but in this work, we will consider both skeletons and "additions" as specialisations of techniques.

in our case depending whether the head of the list is a prime number or not, as follows:

```
%partition traverse
 traverse([X|Xs]) :-
   prime(X),
   traverse(Xs).
 traverse([X|Xs]) :-
   non_prime(X),
   traverse(Xs).
 traverse_n([]).
```

Consider the addition *build* which rewrites a program producing a new one which creates a new list by selectively including elements from the original list. *build* adds an argument (the new list) to the defining predicate and an extra goal to the body of the clause, where the new list will be constructed.

The following is a representation for this addition using the rewrite notation explained in Section 4, where $\mathcal{P}$ is the defining predicate and $\{A_1, \ldots, A_n\}$ its arguments; $\mathcal{C}$ is a conjunction of subgoals for $\mathcal{P}$. $Case(X)$ is a conjunction of subgoals involving X and $Relate(X)$ is a conjunction of subgoals involving $X, Ps1$ and Ps.

$$\begin{array}{l} \mathcal{P}(A_1, \ldots, A_n) \leftarrow Case(X), \\ \qquad\qquad \mathcal{P}(B_1, \ldots, B_n), \\ \qquad\qquad \mathcal{C} \end{array} \Rightarrow \left\{ \begin{array}{l} \mathcal{P}(A_1, \ldots, A_n, Ps) \leftarrow Case(X), \\ \qquad\qquad \mathcal{P}(B_1, \ldots, B_n, Ps1), \\ \qquad\qquad \mathcal{C}, \\ \qquad\qquad Relate(X, Ps1, Ps) \end{array} \right.$$

$$\bullet\ \dot{A} = [X|Xs] \wedge \dot{A} \in \{A_1, \ldots, A_n\}$$

$$\mathcal{P}(A'_1, \ldots, A'_n) \leftarrow \mathcal{C} \quad \Rightarrow \quad \mathcal{P}(A'_1, \ldots, A'_n, []) \leftarrow \mathcal{C}$$

$$\bullet\ \mathcal{P} \notin \mathcal{C}$$

The result of applying extension *build* on skeleton *traverse* is then:

```
only_primes([X|Xs],Ps) :-
  prime(X),
  only_primes(Xs,Ps1),
  Ps = [X|Ps1].
only_primes([X|Xs],Ps) :-
  non_prime(X),
  only_primes(Xs,Ps1),
  Ps = Ps1.
only_primes([],Ps) :-
  Ps = [].
```

This approach to program construction has been used in the development of general purpose tools in different areas of programming [Kirschenbaum *et al.* 94, Bowles *et al.* 94, Vasconcelos 93, Vargas-Vera 95].

2.2 Building models using schema definitions

A Prolog *schema* is a structure for packaging the Prolog code necessary to implement a part of a Prolog program. A schema has packaged inside it not only a piece of code, but also the information about the conditions under which that schema should be used, and the requirements for it to be used along with other schemata. An example of using schemata to generate simulation models is the EL system[Robertson *et al.* 91], a domain dependent system which helps the user in the selection of which schemata must be applied under a determined context. A typical schema definition, like the one used on EL, has the following information:

- *Name* for the schema.
- The simulation *goal* for which the schema solves. If *preconditions* and *actions* execute successfully, and the *subgoals* are satisfied, then the schema is guaranteed to solve this goal.
- List of *subgoals* which the schema produces.
- The piece of Prolog *code* actually supplied by the schema to the final program.
- Procedure calls (*actions*) which must be satisfied before the schema can be applied.
- A condition call which must be satisfied before the schema can be used (*precondition*).

The following is an example of a Prolog schema defining the predicate `attribute/4`. As can be observed, a large part of instantiation work in schemata is done by pattern matching of the parameters.

```
schema([A,' of ',O,' grows logistically'],           % name
       attribute(A,O,T,N),                           % goal
       [initial_time(T),
        last_time(T,T1),
        initial_value(A,O,N),
        parameter(carrying_capacity,O,K),
        parameter(intrinsic_rate_of_increase,O,R)],  % subgoals
       [(attribute(A,O,T,N1) :-
            \+ initial_time(T),
            last_time(T,T1),
            attribute(A,O,T1,N2),
            parameter(carrying_capacity,O,K),
            parameter(intrinsic_rate_of_increase,O,R),
            N1 is N2+R*N2*(1-N2/K)),
        (attribute(A,O,T,N3) :-
            initial_time(T),
            initial_value(A,O,N3))],                 % code
       [],                                           % actions
       (valid_object_for_type(size, A)
        ; valid_object_for_type(energy, A))          % preconditions
       ).
```

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. It is impossible for users to customise schemata, other than through the predefined instantiation procedures of each schema. In the schema above for example, one might like to change the last subgoal in the first clause of the code supplied, let's say "`N1 is N2+R*N2*(1-N2/K)*1.2`" instead of the original expression. Although there is no need to modify any other subgoals in either clauses, another very similar schema would have to be defined, along with additional instantiation/selection procedures. Of course, we could overcome this difficulty in our example by making the equation itself parameterisable but, in general, it can be difficult to predict which parts of a schema to make flexible in this way.

2.3 Making techniques accessible to modellers

It has been argued that by allowing users to adapt predicates at clause level a number of improvements could be obtained, including to make explicit standard methods for constructing Prolog predicates, thus encouraging structured yet flexible program development.

Techniques-editing is a general method which, depending on the environment using it, requires different degrees of assistance from the user to make decisions about program generation [Vargas-Vera 95]. In some environments, such as Intelligent Tutoring Systems (ITS), this feature is often considered desirable. Those systems need to maintain a tight control over users' actions either to gain information about their plan or to detect errors/misunderstandings on their decisions and act accordingly.

In ITS, especially if they are aimed to teach or improve programming expertise (*e.g.* [Robertson 91, Bowles 94]), users may interact directly with the pieces of code resulting from the generation process (or some direct mapping to other visual presentation) making it easier to conduct that process from direct user actions. In that case there is a "short distance" between the external actions and the internal processes controlling the system, therefore a mapping from user actions to (internal) control decisions is not expected to be complex. This analogy is similar to that of *semantic distance* – the distance between the concepts that the system uses and the ones the user has [Hutchins *et al.* 86].

In our case, the scenario is quite different – we assume the user wants only to describe, in familiar terms, the model s/he wants to represent, intentionally left unaware of those issues involved in the program generation process. There is then a long distance between external actions and internal control decisions. The problem is to bridge between user's knowledge about modelling and the system's knowledge of techniques editing and use that to guide the model generation process.

3 Translating existing models to logic programs

In order to understand how Prolog techniques relate to this form of modelling we must reconstruct a representative example in Prolog. For this, we used a model from the ecological modelling literature[Crête *et al.* 81]. This model was selected because:

- It is a representative population dynamics model, used to explore a predator-prey relationship and it appears to contain reliable data supporting a plausible hypothesis.
- It is small enough to be implemented and tested/refined in a realistic time, yet it embodies the main characteristics which are expected to be needed in this sub-domain as well as allowing us to analyse possible extensions.

3.1 Main components

The main components of our case-study were wolves (the predator) and moose (the prey) populations. The goal of the model was to study how wolves could regulate a moose population. Reproduction was represented in either of two ways: using constant values or individual reproduction rates. Predation (the main cause of mortality for moose) varied according to season, as in winter wolves have access to alternative food. A variant of the basic model included hunting as another mortality factor for moose. Starvation (or indirect consequences of malnutrition) was the only mortality factor for wolves. Both populations were structured in sex and age classes so there is a component in the model for each sex $\times$ age combination. This is known in the domain as *disaggregation*. The year was divided into two seasons, summer (June–October) and winter (November–May).

3.2 Modelling approach

We have used a process-centred paradigm to implement the model. Most of the simulation models we have seen on the literature are described in terms of the ecological processes involved. Such processes are standard in the literature and represent phenomena affecting the main focus of the simulation. For example, in a model to evaluate the progress of some population over time, reproduction and mortality are typical processes affecting the size of population over a time interval. This *modularity* in the representation of aspects of a problem helps to provide a structured approach to modelling.

The core part of the model is an iteration over discrete time-steps. At each time-step, the processes affecting moose and wolf populations are computed and the size of their populations updated. This procedure is applied iteratively until the final time-point be reached. We next summarise some aspects of the reconstruction of the model.

Two levels of time representation and processing were used in the model. This was because most actions over the population should be taken every season, although some were taken annually. Thus, a *season* was considered the basic time-unit, and a mapping between *season* and *year* was provided.

Both populations were affected by the same categories of ecological processes, namely: *reproduction*, *mortality* and *ageing*, with different instances of them for each population. Some of those eco-processes required the definition of variables – *starvation* of wolves, for example, depended on the availability of prey which had to be converted from the number of elements to biomass. Thus, a *weight table* giving average moose's weights according to their age was set up.

3.3 Prolog implementation of the model

The following is part of a simulation program in Prolog which illustrates how the model described above can be implemented as a logic program. This is the "Simulation Model" box in Figure 1.

The predicate which encapsulates the whole model is `model/2`. The first of its two arguments is the time-point where the simulation will end and the other is a data-structure embodying each population, which is the result of the simulation. The core predicate of the model is `population/2`. It implements an interaction over time-points implemented as a recursive Prolog predicate in which the base-case sets the initial time and values for the simulation - that provides the "top-level" control of the model. TeMS doesn't confront ecologists with definitions like these, instead, it uses a domain-specific interface to control the selection and synthesis of them, as we describe in Section 5.

```
%
% encapsulation
%
model(A, B) :-
  open_dc,
  population(A, B),
  close_dc.

%
% core predicate - ticking clock
%
population(A, B) :-
  initial_time(B),
  start_value(A),
  collect(B, A).
population(A, B) :-
  \+ initial_time(B),
  previous_time(B, D),
  population(C, D),
  update_population(B, C, A),
  collect(B, A).

%
% update all components
% - adjust parameters
update_population(A,B,C) :-
  update_components(A, B, B, C).

%
% update all components
% - traverse data-structure
update_components(_,_,[],[]).
update_components(A,B,[[G,F,E]|H],
                 [[G,F,C]|D]) :-
  update(A, B, [G,F,E], C),
  update_components(A, B, H, D).
```

Population updating is done by traversing the population data-structure (`update_population/3` and `update_components/4`).

For each component, population updating (`update/4`) is done according to the combined effect of several processes. Each process has its equation (stored in `selected_proc/5`) computed using a Prolog meta-interpreter (`compute_formula/6`). Those processes can be grouped in "categories" according to the way they affect the population of a component, and they are implemented by predicates specific to each of those categories (*e.g.* `distribute_on_first_level/5` for natality and `subtract_list_values/3` for mortality).

```
%
% update each component
%
update(A, B, C, D) :-
  C=[J,_,I],
  J=moose,
  rep_rate(A, B, C, I, H),
  predation1(A, B, C, H, G),
  hunting(A, B, C, G, F),
  ageing(A, B, C, F, E),
  adj_values(E, D).
update(A, B, C, D) :-
  C=[I,_,H],
  I=wolf,
  rep_rate(A, B, C, H, G),
  starvation(A, B, C, G, F),
  ageing(A, B, C, F, E),
  adj_values(E, D).

%
% natality for moose
%
rep_rate(A, B, C, D, E) :-
  C=[K,J,_],
  K=moose,
  selected_proc(moose, _,
                rep_rate, f_(G,H), I),
  apply_on_time(A, I),
  compute_formula(A, moose, B, G, H, F),
  distribute_on_first_level(moose,
                            J, D, F, E).
rep_rate(_, _, A, B, B) :-
  A=[C,_,_],
  C=moose.

%
% mortality (type 1) for moose
%
predation1(A, B, C, D, E) :-
  C=[J,_,_],
  J=moose,
  selected_proc(moose, _,
                predation1, f_(G,H), I),
  apply_on_time(A, I),
  compute_formula(A, moose, B, G, H, F),
  subtract_list_values(D, F, E).
predation1(_, _, A, B, B) :-
  A=[C,_,_],
  C=moose.

%
% adding newborns to population
%
distribute_on_first_level(_,[G],
       [[E,D]|F], A, [[E,C]|F]) :-
  atom(G),
  list_sum(A, B),
  C is D+B.
distribute_on_first_level(A,B,C,D,E) :-
  list_sum(D, K),
  first_level(A, B, J),
  findall(H,member([J,H],C),I),
  length(I, G),
  F is K/G,
  add_parcel(J, F, C, E).
```

In the following sections we show how programs like this one can automatically be produced.

4 Typical model structures

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 94] presents a methodology for extracting techniques used in a Prolog program. Similarly to its counterpart in the procedural paradigm [Weiser 84], it uses evaluation over the arguments of a predicate to find out all relevant clauses to that argument and to partition the procedure into a set of argument slices which are considered separable techniques. However, in that method techniques are extracted *with respect to a specific query* and techniques must always have *only one argument*; conditions which may not be met in our domain. Furthermore, example-based methods require a representative case library which doesn't exist for ecological modelling.

The following techniques and schemas were identified by studying the implemented model, associating those structures with features which are likely to be present in many other population dynamics models. This corresponds to the "Design-specific techniques" box in Figure 1. Instead of using more abstract representations for techniques, we show them simply as pieces of Prolog code upon which rewrites must be done, with variables beginning with a capital letter. We shall later demonstrate how these are used in model construction.

recursion over time points is the technique which defines the flow of control used in the main predicate of the simulation. It supplies a "ticking clock" which allows all the processes in the model to be calculated at appropriated "ticks" in the sequence of time points determined by the clock. From our implementation of Crête's model, the following skeleton for *regression over time points* was extracted ($\mathcal{P}$ must be substituted by the name of the predicate).

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

A more general form for this skeleton is:

```
𝒫(T) :-
    TimeLim(T).
𝒫(T) :-
    \+ TimeLim(T),
    Adjacent(T,Tadj),
    𝒫(Tadj).
```

With the latter, more instantiation would be necessary. A possible substitution is {*TimeLim*/`final_time`, *Adjacent*/`next_time`}, which produces the skeleton for *progression over time points*.

population updating as the name suggests, incorporates into the code being edited (*working code*) an updating of the data-structure representing the size of one of the populations in the model. An extra argument is added to the defining predicate and extra goals are added to the body of recursive and non-recursive (base-case) clauses, those extra goals relate the updating from the body with the final value in the head of a clause.

We represent the application of this techniques as a rewrite ($\Rightarrow$) upon a program. As used here, a program is a finite set of clauses of the form:

$\mathcal{P}(A_1, \ldots, A_n) \leftarrow \mathcal{C}$

where:
$\mathcal{P}$ is the defining predicate and $\{A_1, \ldots, A_n\}$ its arguments;
$\mathcal{C}$ is a set of subgoals $\{P_1, \ldots, P_m\}$;
$n \geq 0,\ m \geq 0$;
For clarity, we distinguish different variables in schemata and techniques using prime ($'$) symbols.

This technique can be represented as:

$$\mathcal{P}(A_1, \ldots, A_n) \leftarrow \mathcal{C} \quad \Rightarrow \quad \mathcal{P}(A_1, \ldots, A_n, V) \leftarrow \mathcal{C}, \texttt{start\_value}(A, V)$$

- if $\mathcal{C}$ does not contain $\mathcal{P}$ as a goal (non-recursive clauses), henceforth: $\mathcal{P} \notin \mathcal{C}$, $A \subseteq \{A_1, \ldots, A_n\}$

$$\begin{array}{l} \mathcal{P}(A'_1, \ldots, A'_n) \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B_1, \ldots, B_n) \end{array} \Rightarrow \begin{cases} \mathcal{P}(A'_1, \ldots, A'_n, V') \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B_1, \ldots, B_n, X), \\ \qquad\qquad \texttt{update\_population}(A', X, V') \end{cases}$$

- $A' \subseteq \{A'_1, \ldots, A'_n\}$

ancillary procedures – It might be the case that the core procedure (`population/2` in our example) needs to be preceded ($\mathcal{C}_{pre}$) and/or followed ($\mathcal{C}_{post}$) by ancillary procedures (*e.g.* opening/closing files or loading external programs). The following is a technique for encapsulating the core procedure within an outermost predicate (`model/2` in our example). We assume an editing environment with ability to add goals on top or bottom of the body of a clause and pre-defined instantiations for $\mathcal{C}_{pre}$ and $\mathcal{C}_{post}$.

$$\mathcal{P}(A_1, \ldots, A_n) \leftarrow \mathcal{C} \quad \Rightarrow \quad \mathcal{P}(A_1, \ldots, A_n) \leftarrow \mathcal{C}_{pre}, \mathcal{C}, \mathcal{C}_{post}$$

traverse_pop is an instance of the skeleton *traverse_n* presented in [Sterling & Kirschenbaum 93]. It embodies the main flow of control for traversing a list partitioning it according to some criteria (see Section 2.1). In our case, the list is partitioned in two cases which represent the way the population of components are referred to, namely: disaggregated (population is organised in sub-groups) and non-disaggregated (see Section 5.2 for more details on population organisation).

```
𝒫([H|T]) :-
    disaggregated(H,none),
    𝒫(T).
𝒫([H|T]) :-
    \+ disaggregated(H,none),
    𝒫(T).
𝒫([]).
```

get_value is an addition which may be made to the technique above. It adds an argument to the defining predicate and add extra subgoals to the body of each clause in order to create a new data object with initial population information. We assume that `initial_st/3` is the predicate from the model's specification with initial values for each parcel of the population – In `initial_st`(C, D, V) for example, C is a component name, D is some disaggregation dimension and V is the value of that sub-population.
For components with non-disaggregated population, `initial_st/3` and another goal relating its arguments with the final object in the head (for that reason called "constructor goal") are added to the body of the clause. For components of a disaggregated population, `findall/3` (a SICStus Prolog built-in predicate) is used to determine all sub-groups and initial-values for all those sub-groups of a population, along with the constructor goal as before. The technique is then as follows:

$$\begin{array}{l}\mathcal{P}(A_1,\ldots,A_n) \leftarrow \mathcal{C}, \\ \qquad\qquad \mathcal{P}(B_1,\ldots,B_n)\end{array} \Rightarrow \left\{\begin{array}{l}\mathcal{P}(A_1,\ldots,A_n,V) \leftarrow \mathcal{C}, \\ \qquad\qquad \mathcal{P}(B_1,\ldots,B_n,T), \\ \qquad\qquad \mathtt{initial\_st}(C, none, X), \\ \qquad\qquad V = [[C, none, X]|T]\end{array}\right.$$

$$\bullet\ \dot{A} = [C|C_s],\ \ \dot{A} \in \{A_1,\ldots,A_n\}$$

$$\begin{array}{l}\mathcal{P}(A'_1,\ldots,A'_n) \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B'_1,\ldots,B'_n)\end{array} \Rightarrow \left\{\begin{array}{l}\mathcal{P}(A'_1,\ldots,A'_n,V') \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B'_1,\ldots,B'_n,T'), \\ \qquad\qquad \mathtt{findall}(D_2, \mathtt{disaggregated}(C', D_2), D), \\ \qquad\qquad \mathtt{findall}([D_1, V_1], \mathtt{initial\_st}(C', D_1, V_1), X'), \\ \qquad\qquad V' = [[C', D, X']|T']\end{array}\right.$$

$$\bullet\ \dot{A}' = [C'|C'_s],\ \ \dot{A}' \in \{A'_1,\ldots,A'_n\}$$

$$\mathcal{P}(A''_1,\ldots,A''_n) \leftarrow \mathcal{C}'' \quad \Rightarrow \quad \mathcal{P}(A''_1,\ldots,A''_n,[]) \leftarrow \mathcal{C}''$$

As this example illustrates, domain-specific techniques depend not only on data-structures parsed as arguments but also on other predicates from the model specification (*e.g.* `initial_st/3`).

The predicate `initial_value/2` associates the main identifier of a component (usually its name) with a data-structure containing the initial values of the population for that component. The following is the actual code for `initial_value/2`. It is easy to see that the skeleton *traverse_pop* with the addition *get_value* might be used to generate it.

```
initial_value( [C|T] , Pop ) :-
  disaggregated(C,none),
  initial_value( T , T2 ),
  initial_st(C,[none],P),
  Pop = [[C,[none],P]|T2].
initial_value( [C|T] , Pop ) :-
  \+ disaggregated(C,none),
  initial_value( T , T2 ),
  findall(C,disaggregated(C,Dim),All_Dim),
  findall([Dim,Vl],initial_st(C,Dim,Vl),All_P),
  Pop = [[C,All_Dim,All_P]|T2].
initial_value( [] , [] ).
```

conversion is another addition to *traverse_pop* which adds arguments and goals to the defining predicate in order to implement a "conversion" of values. It might be used for example, to compute biomass from population values. It uses a predicate (`factor/3`) as a table for conversion factors. The predicate `for_each/5` applies `factor/3` to every sub-group of the population. In the end, a converted value is structured in the same way of the original.

$$\begin{array}{l} \mathcal{P}(A_1,\ldots,A_n) \leftarrow \mathcal{C}, \\ \qquad \mathcal{P}(B_1,\ldots,B_n), \\ \qquad V = [[C, none, X]|T] \end{array} \Rightarrow \left\{ \begin{array}{l} \mathcal{P}(A_1,\ldots,A_n,V_2) \leftarrow \mathcal{C}, \\ \qquad \mathcal{P}(B_1,\ldots,B_n,T_2), \\ \qquad V = [[C, none, X]|T], \\ \qquad \mathtt{factor}(C,F,X,X_2), \\ \qquad V_2 = [[C, none, X_2]|T_2] \end{array} \right.$$

$$\begin{array}{l} \mathcal{P}(A'_1,\ldots,A'_n) \leftarrow \mathcal{C}', \\ \qquad \mathcal{P}(B'_1,\ldots,B'_n), \\ \qquad V' = [[C', D, X']|T'] \end{array} \Rightarrow \left\{ \begin{array}{l} \mathcal{P}(A'_1,\ldots,A'_n,V'_2) \leftarrow \mathcal{C}', \\ \qquad \mathcal{P}(B'_1,\ldots,B'_n,T'_2), \\ \qquad V' = [[C', D, X']|T'], \\ \quad \mathtt{for\_each}(P_y, X', (Y = [D_y, P_y], \mathtt{factor}(C', F', P_y, P)), P, X'_2), \\ \qquad V'_2 = [[C', D, X'_2]|T'_2] \end{array} \right.$$

$$\bullet\ D \neq none$$

$$\mathcal{P}(A''_1,\ldots,A''_n) \leftarrow \mathcal{C}'' \quad \Rightarrow \quad \mathcal{P}(A''_1,\ldots,A''_n,[]) \leftarrow \mathcal{C}''$$

The goals $V = [[C, none, X]|T]$ and $V' = [[C', D, X']|T']$ on the left hand side of this technique assures that a technique (*get_value*) defining an initial data object had already been applied.

chained composition joins one or more processes[2] to the body of a clause, composing arguments in such way that the new information supplied by one process is used as an input-argument to the following and so on. An argument in the last process is linked to an argument in the head of the defining predicate.

This technique adds two arguments to the defining predicate. The first one is the initial structure which will be "passed through" a series of processes (each process is a goal to be added to the body of the clause). The second argument is the composed effect of all processes, that is, the

[2]See Section 3.2 for a definition of "process" as mentioned here.

updated data-structure. An extra goal defines the initial structure from a subset of the original arguments.

$$\mathcal{P}(A_1,\ldots,A_n) \leftarrow \mathcal{C} \Rightarrow \begin{cases} \mathcal{P}(A_1,\ldots,A_n,I,O) \leftarrow \mathcal{C}, \\ \qquad\qquad Relate(A,I), \\ \qquad\qquad P_1(A,I,L_1), \\ \qquad\qquad P_2(A,L_1,L_2), \\ \qquad\qquad \vdots \\ \qquad\qquad P_m(A,L_{m-1},O) \end{cases}$$

$$\bullet\ \mathcal{P} \notin \mathcal{C}, A \subseteq \{A_1,\ldots,A_n\}, m \geq 0$$

$$\begin{array}{l} \mathcal{P}(A'_1,\ldots,A'_n) \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B'_1,\ldots,B'_n) \end{array} \Rightarrow \begin{cases} \mathcal{P}(A'_1,\ldots,A'_n,I',O') \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B'_1,\ldots,B'_n,L_m,O), \\ \qquad\qquad Relate(A',I'), \\ \qquad\qquad P'_1(A',I',L'_1), \\ \qquad\qquad P'_2(A',L'_1,L'_2), \\ \qquad\qquad \vdots \\ \qquad\qquad P'_m(A',L'_{m-1},L_m) \end{cases}$$

$$\bullet\ A' \subseteq \{A'_1,\ldots,A'_n\}, m \geq 0$$

partition is a skeleton where the first goal (which can be unfolded as a composition of several checking subgoals) defines the applicability of that clause.

```
𝒫( Key, St, NewSt ) :-
    Case_1(Key),
    P1(St,NewSt).
⋮
𝒫( Key, St, NewSt ) :-
    Case_n(Key),
    Pn(St,NewSt).
```

Applicability of ecological processes according to seasons, where each case condition determines whether the current time point in the simulation is within a particular season, is an example of where this technique would be used.

In our example, the skeleton *partition* with the addition of *chained composition* is used to define the predicate `update/4`, which represents population updating for a certain component. At every time-step, the effect caused by ecological processes on the component's population is computed by this predicate. The following is a listing of `update/4`, where `Comp = wolf` is the case condition.

```
update(T,Ref,P,Pout) :-
  Comp = wolf,
  P = [Comp,_,Pin],
  ageing(Comp,T,Ref,Pin,S1),
  natality(Comp,T,Ref,S1,S2),
  mortality(Comp,T,Ref,S2,S3),
  adj_values(S3,Pout).
```

shift moves part of the contents of every "cell" within a data structure to the next (adjacent) one. Two arguments are included to the defining predicate: the first is the part of the structure to be shifted and the second is the data object to be created. The two *Relate* goals added to the body of recursive clauses relate the two adjacent cells in the data structure and the third goal is the constructor goal for the new object.

$$\mathcal{P}(A_1,\ldots,A_n) \leftarrow \mathcal{C} \quad \Rightarrow \quad \mathcal{P}(A_1,\ldots,A_n,X,[]) \leftarrow \mathcal{C}$$
$$\bullet\, \mathcal{P} \notin \mathcal{C}$$

$$\begin{array}{l}\mathcal{P}(A'_1,\ldots,A'_n) \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B_1,\ldots,B_n)\end{array} \Rightarrow \left\{\begin{array}{l}\mathcal{P}(A'_1,\ldots,A'_n,I,S) \leftarrow \mathcal{C}', \\ \qquad\qquad \mathcal{P}(B_1,\ldots,B_n,V,T), \\ \qquad\qquad \mathit{Relate}(H,\dot{A},V), \\ \qquad\qquad \mathit{Relate}(H_2,\dot{A},I), \\ \qquad\qquad S=[H_2|T]\end{array}\right.$$
$$\bullet\, \dot{A}=[H|R] \wedge \dot{A} \in \{A'_1,\ldots,A'_n\}$$

In our example, the population was organised in age-classes and this technique would be used to shift the values of each class to the adjacent one, a task needed when representing *ageing* in the model. Using the skeleton *traverse* (Section 2.1) with the addition of *shift*, it is possible to obtain the definition of `move_elem/3`:

```
move_elem([],_,[]).
move_elem([H|T],In,O) :-
      move_elem(T,N,T2),
      H = [AgeClass,N],
      H2 = [AgeClass,In],
      O = [H2|T2].
```

The preceding examples demonstrate that domain-dependent techniques depend upon other domain-specific data-structures. Therefore a library of such techniques, which will probably include other standard data-structures, may only be complete for the set of operations or features of a narrow class of applications. That means that the range of operations carried out by an application must be known prior to definition of the techniques library: that is, design decisions shape the techniques and schemata in the library.

5 Automated Synthesis

As a prerequisite for building a plausible model, we assume that the builder is knowledgeable about the ecological system which is the basis for the model. However, to have knowledge about a system is not enough to start building a model. One can have a good idea of a model's main parameters and/or the final shape it should have, but not know how to put all together in a neat piece of code in some programming language.

TeMS leads ecologists through a structured sequence of design decisions to guide the synthesis of this class of models. A library of design-specific techniques and schemata, including those presented in Section 4, was defined and used by a Techniques Editor to generate the programs. Thus, the system presented here:

- is a tool that leads ecologists through several structured steps for the specification of a model;
- is based on its own knowledge-base as well as the knowledge acquired from the user, and generates a runnable Prolog program that is the implementation of the model;
- can execute the constructed Prolog model;
- may record the path taken during the definition process, allowing the display of the choices made during the modelling process.

Modelling using such a system consists of going through three phases, as shown on Figure 2. On the first phase the user is led through six stages of model description where key features of the model are defined. On the second phase the knowledge elicited is used to select and apply an appropriate set of techniques to build the model described. Finally, on the last phase the user can see the Prolog code for the model and run it. If a redefinition or adjust of the model is needed, the user may start again from some stage on phase I.

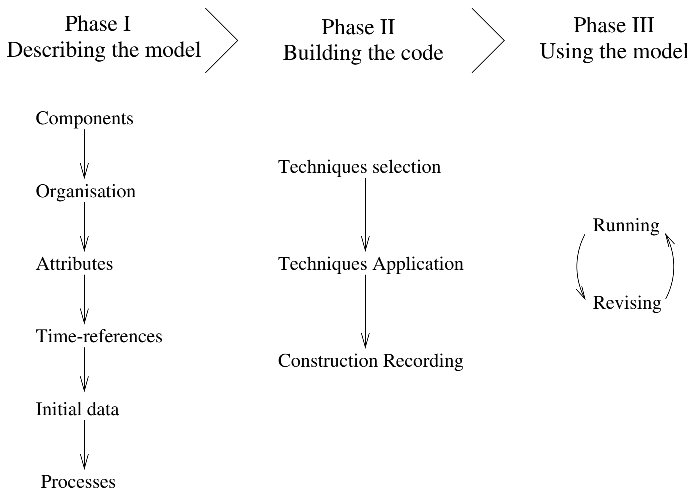


Figure 2: 3-phase modelling

5.1 A model synthesiser

The Techniques-based Model Synthesiser (TeMS) elicits knowledge from an ecologist, using his/her design decisions to set up the program generation stage (phase II on Figure 2). Figure 3 shows the general structure of the tool. The generation system is the core of the tool, defining which predicates will form the final program and in which order they will be generated. For each predicate, there must be specified the set of techniques editing operations to be carried out by the techniques editor. A Prolog meta-interpreter to deal with equations of the processes is also used during model definition and execution. Finally, for each model the knowledge base is extended to be used by the generation system. The next sections show how the 3-phase modelling framework is used by TeMS.

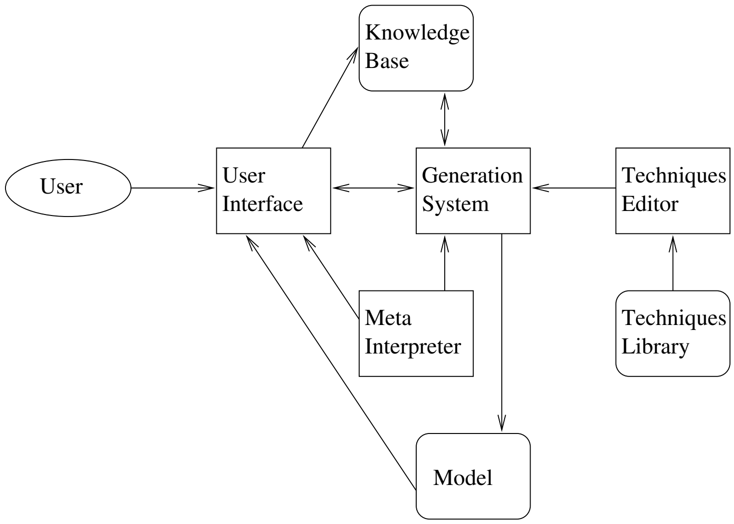


Figure 3: TeMS

5.2 Describing the model

In this phase the user outlines the model's structure. The user's task consists of selecting options from menus and answering prompted questions according to characteristics of the system he/she wants to model. The set of questions and standard options presented to the user are stored as predicates in a knowledge base.

After each answer is given (either by selecting one among a set of options, typing some value or sketching a graph), the contents of the knowledge base will be extended and used to define the next step on this elicitation process, that is, the next menu will be presented or a question will be prompted.

The menu/questions presented to the user cover the following elements of the model: components, organisation, attributes, time-references, initial-data and ecological processes. The result of this phase is a knowledge-base "tuned" to a specific model, which will be used on the next phase. Those elements are described in the following sections.

5.2.1 Components

When defining a model on population dynamics it is essential to know what are the *components* whose population behaviour we will observe during the simulation. A user can either select among a list of typical components or introduce a new one.

If the latter, the user must also categorise the new component according to standard categories used when somebody talks about the animal kingdom. Thus, every component is associated with one element in each of the following sets: {*herbivore, carnivore, omnivore*}, {*vertebrate, invertebrate*} and {*insect, bird, fish, mammal, other*}.

Such knowledge may later be used to infer relationships between components. The following, for example, is a rule in the knowledge base expressing a predator-prey relationship between two components of a model: Two different elements are taken from the list of components, if one of them is "carnivore" or "omnivore", the other might be subject to predation. Rules such this are used to select which pre-defined processes will be suggested to the user, but these are a guide only. Hence, it is not a requirement that the ecological knowledge base should be complete.

```
would_be_predator(A, B) :-
  components(L),
  member(A,L),
  member(B,L),
  \+ A = B,
  ( component_sort(A,carnivore)
    ; component_sort(A,omnivore) ).
```

5.2.2 Organisation

The population in a model is usually categorised according to its components. However, each component may have its own way of referring to the elements which constitute its population. The elementary unit in a population may be an individual or, more commonly, a group of individuals which have common attributes.

These groups are defined by the combination of the dimensions according to which a population is *disaggregated.* Those *dimensions of classification* are attributes of a component as shown in the next section. There will be as many sub-populations as there are elements in the Cartesian product of the attributes. The size of a *non-disaggregated* population, on the other hand, is represented by a simple number. These different representations makes disaggregation a crucial selective factor in the way population values are computed, as seen on technique *traverse_pop* for example.

5.2.3 Attributes

Every component may have a set of *attributes* associated to it. Attributes are used to define characteristics which are meaningful to a modeller when talking about a component. Attributes also are important to represent properties which should only apply to elements of certain categories.

Attributes can be inherent in components (*e.g.* age, sex, weight and size) or they may represent a quality that was for some reason "associated" to it (*e.g.* location). Attributes' values also can be constant (*e.g.* sex), or variable on time (*e.g.* size, location). As mentioned before, attributes also inform how a population is disaggregated. That information is embodied in rules on the knowledge base and it is used as selection criteria in the other stages of the process.

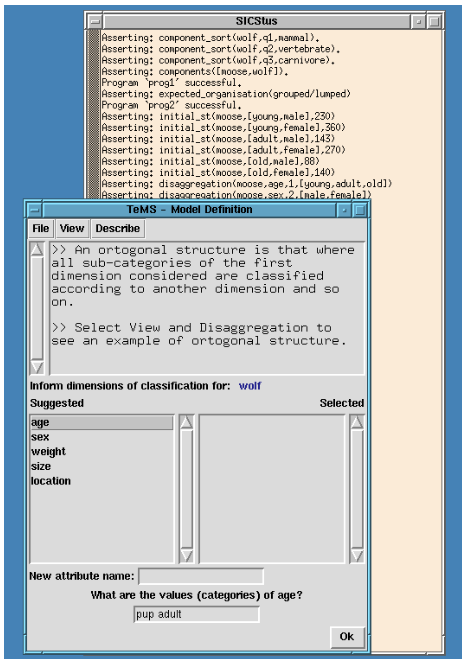


Figure 4: Attributes definition

Figure 4 shows a "snapshot" of model description using TeMS. The window at front shows the user suppling the information that wolf population will be disaggregated by *sex* ({*male, female*}) and *age* ({*pup, adult*}) classes. The window behind is a log of the Prolog session (normally invisible to the user), where previously elicited knowledge has already been asserted in the knowledge base.

5.2.4 Initial data

Once the dimensions of disaggregation are known, the system can prompt the user for the initial values of population that the simulation will start with. If the population is not disaggregated, a simple value is asked for. If the component has a disaggregated population, the system will ask as many values as the number of sub-populations considered. The actual order in which the values are asked depends on the order in which the attributes are given by the user.

5.2.5 Processes

Among the aspects of reality a model represents, there are actions or operations affecting the population of one or more components, responsible either for increasing, decreasing, or rearranging it. Those actions, called *processes*, must be mirrored during the simulation. Typical processes are natality, mortality, immigration, emigration and ageing. Some of them may be the accumulated effect of several processes (mortality for example, might be a result of starvation and predation), others are instances of more general definitions (ageing for example, may be a specific sort of "movement between classes").

We have grouped processes in categories, according to their common effect upon a component's population. To illustrate those different effects, consider that a population is disaggregated into age classes and is affected by two processes: *mortality* and *natality*. When representing mortality, whatever way is used to define the number of deaths in each age-class, the standard procedure is to subtract that amount from the current population on each age-class. Natality, on the other hand, requires the number of births to be computed at every age-class and to be added only to the first age-class. Currently, we have the following categories of processes: *natality, mortality, progress on category*[3] and *migration*.

For each component in the model, the system prompts the user for the processes affecting the size of population of that component. Then they must either choose one of the pre-defined processes shown (the system infers from the knowledge-base which processes may apply) or define a new one.

A process is defined by an equation which may include user-defined variables (*e.g.* individual reproductive rate, rate of predation). An equation is either an arithmetic expression, an if-then-else declaration or a two-dimensional graphical function. If the latter is used, the graph is represented as a set of lines of the sort $y = ax + b$ and the actual value of a variable is obtained by an interpolation of the corresponding interval.

A pre-defined process consists of a standard equation along with a set of parameters which must be given values and, optionally, a condition which that process may apply. predation1 for example, is an instance of mortality where the new value of a prey population P_{t+1} is the number of deaths D subtracted from the previous value P_t. The number of deaths in its turn is the product of the current population of the predator $Pred_t$ by the value of a variable representing the predation rate (eat). That is:

$$P_{t+1} = P_t - D \qquad \text{and} \qquad D = eat \times Pred_t$$

The condition `would_be_predator(X,C)` (`C` instantiated to the component's name) completes the definition of that process. In the case of predation1 be selected, the user would be requested to supply a value (or equation) for eat.

Figure 5 shows the selection of pre-defined process predation1 where the user is defining the value of eat_moose by sketching a graph of it as a function of moose population size.

5.2.6 Time Reference

Every process represented in the model will have its effects over the population according to some time reference. The main time-reference used throughout the simulation is the time-step, that is, the interval of time in which the state of

[3]When a population is disaggregated in *ordered* classes (*e.g.* age, size), it represents the movement of elements from one category to the following one (*e.g.* ageing, growing in size)

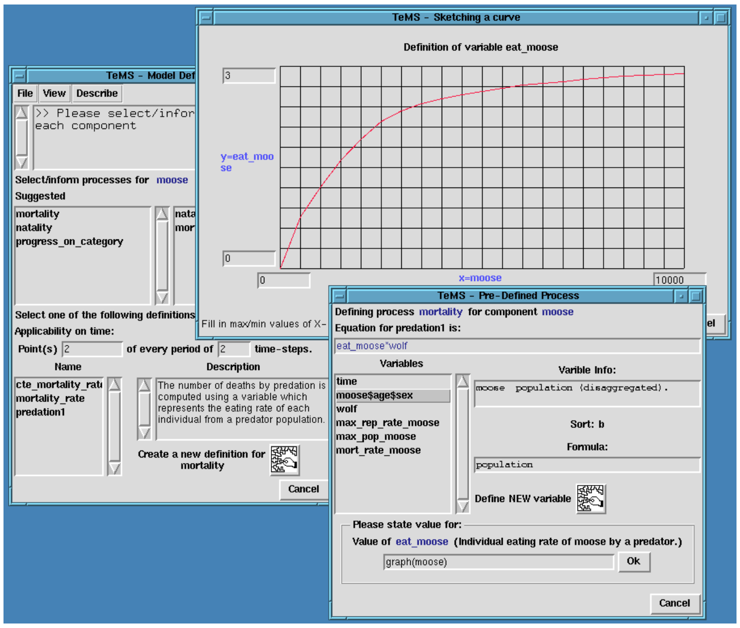


Figure 5: Process definition

the population will be calculated. However, in Section 3.2 we explained that some notion of hierarchical time is required in these sorts of model. To provide this we ask, when talking about applicability of processes, the user to give the number of time-steps defining the period T and the subsets of T in which the process applies. For example, to represent a process which applies in January and July of every year, we use 12 and $\{1\ 7\}$.

5.2.7 Setting up program construction

After each successful definition of a process, two kinds of data-structures are included in the knowledge base. The first, embodying the equations for variables and processes, will be used to run the model. The second are data-structures used during the generation stage, namely: *list of candidates*, defining which predicates must be generated and in which order; *code construction records (ccr)*, defining the sequence of techniques-editing operations needed to generate each predicate; and *binding records* controlling variable binding at each techniques-editing operation.

The following algorithm shows how this part of the model description is used to define the data-structures needed for program generation.

$$enter\_process(Comp, Proc, Cat, Eq)$$

where: $Comp$ is the component name; $Proc$ is the defining process; Cat is the category of the defining process and Eq is the process' equation.

IF *Comp* has a non-disaggregated population THEN

IF *Proc* is defined by user (new equation) THEN

- check the equation.
- get (from the knowledge base) the first part of techniques sequence and binding records, those corresponding to equation-defined and non-disaggregated kind of process definition. That is: `Sequence1 = [proc_head, def_comp, applicability, compute_eq]` and corresponding bindings.
- get second part of techniques and binding sequences, that is, the one assigning different behaviours for different categories of processes. For example, if Cat = natality, then `Behaviour = distribute_on_first_level`.
- build construction records (ccrs) for *Proc*.
  For the case above (Cat = natality), the technique-editing operations sequence would be:

```
TS = [unif_pred_name(proc_head,Proc,C1),
      add_technique(def_comp,C1,C2),
      add_technique(applicability,C2,C3),
      add_technique(compute_eq,C3,C4),
      add_technique(distribute_on_first_level,C4,Out)]
```

- include *Proc* in the list of candidates, assert corresponding ccr, binding records and equation.

ELSE IF *Proc* is selected from predefined set THEN

CASE *Proc* is defined by

equation THEN

proceed as for user-defined process;

schema THEN

- use arguments for instantiating the schema - no ccr/binding records are needed;
- include *Proc* in the list of candidates.

technique composition THEN

- define supplementary data-structures;
- get corresponding ccr and bindings for the composition. The user may be prompted for choosing amongst several choices.
- include *Proc* in the list of candidates and assert ccr/binding record.

END CASE

END IF

ELSE (population of *Comp* is disaggregated)

proceed as for non-disaggregated population, but with suitable selection from the knowledge base.

END IF

Any other process definition belonging to one of the categories previously defined may be added to the knowledge base without change in the algorithm. In the same way, some new categories can have their *modus operandi* embodied in schemata and inserted in the knowledge base. More complex changes would require new ways of linking process definition with program generation, and might require us to redefine the algorithm. Although we have impose some limitations to pre-defined processes and variables applied on disaggregated population, we believe the current implementation covers a sensible number of situations it might rise in population models as targeted by our framework.

5.3 Constructing the program

Once the structure of the model has been defined, program construction may start. This process consists of successive selection and application of techniques, for which it is necessary to know the sequence of *which* predicates need to be generated and *how* it will be done, that is, the sequence of techniques-editing operations needed. However, as pointed out on section 2.3, we cannot expect the user to choose which techniques are to be used or how to parameterise them.

In order to allow only meaningful predicates to be generated and to do that without user intervention, selection and application of techniques have to be restricted and some data-structures were defined to help in that task. Thus, the system knows which predicates will be generated by using a data-structure called *list of candidates*. Every "candidate" in the list must have associated to it, information on the techniques-editing operations which will actually generate the predicate, namely: *code construction record (ccr)* and *binding record*, mentioned earlier. A techniques-editing sequence within a *ccr* may only be triggered when a predicate from the list of candidates is in its turn for generation.

There are standard ways to add elements to the list of candidates, some predicates are included by default (by assertion in the knowledge base) or when some requirement may be proven on the knowledge base. Others are included after the definition of a process during model description, as described by algorithm *enter_process*.

While definition of techniques-editing sequences for each process is defined as in *enter_process*, an extra predicate with calls for all of them, must be done. It is worth to note that unusual placing of the arguments is needed. The following algorithm defines the sequence of techniques-editing operations necessary to build such predicate `update/4`, already mentioned in Section 4 (techniques *chained composition* and *partition*).

$$compose\_update(Comp, Procs)$$

$Comp$ is the component name; $Procs$ is the set of processes for that component.

IF there is no sequence started THEN

- start sequence: `TS = [unif_pred_name(proc_head,update,Code),`
  `add_technique(def_comp2,Code,C1)]`
- assert last binding record

END IF

IF $Procs$ is an empty set THEN

- add last goal to sequence:
  `TS = [ ...  , add_technique(adjust,LastCode,CodeOut) ]`
- assert binding record
- include `update/4` in the list of candidates

ELSE

- take the first process (P) from $Procs$, let T be the rest of that set
- include P in the sequence (process call):
  `TS = [ ...  , add_technique(`P`,PreviousCode,NewCode) ]`
- $compose\_update(Comp, T)$
- assert binding record

END IF

The core task of the generation stage is to search the knowledge base for new candidates, get their ccrs and binding records, and execute them, that is, to apply the techniques editing sequences appropriate to each candidate in the list of candidates as well as to each new candidate found to be needed during that process.

The following are some signposts from the construction of a typical model within the framework presented here.

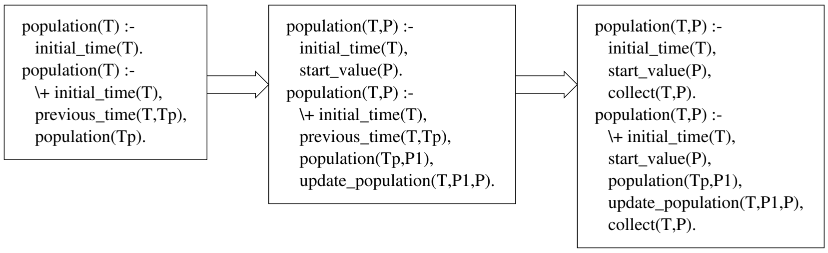
- The "entrance predicate" is by default, the first on the list of candidates. If `data_collection(yes)` can be proven, then ancillary procedures (see technique on Section 4) will be used to produce:

  ```
  model(T,P) :-
    open_files,
    population(T,P),
    close_files.
  ```

  Predicates `open_files/0` and `close_files/0` will be defined by schemata and predicate `population/2` is the core predicate for the simulation and will involve some techniques-editing operations.

- The standard way we adopted to build the core predicate (see Section 3.3) is: We start by taking the technique *recursion over time points* as the main flow of control, then apply technique *population updating* and if `data_collection(yes)` can be proven, technique *data_collection* will also be applied.

  Upon completion, this procedure asserts in the knowledge base the predicate's name and arity along with proper ccr and binding records. The following is a diagram showing the several stages in the execution of the techniques-editing sequences on this predicate's ccr.



- At each technique application when defining the predicate above, new elements are included in the list of candidates (CL) corresponding to those new goals added to the body of the defining predicate. These are `initial_time/1, start_value/1, collect/2` and `update_population/3`.

- Some of the new elements in the list of candidates are standard in the domain and can be added to the program without ccrs or binding records.

- Other elements have to be defined through new techniques application. Two examples of the latter were shown on Section 4: The first is the predicate `initial_value/2` (included in CL when defining `initial_time/1`), which uses techniques *traverse_pop* and *get_value*. The second is one of the definitions for predicate `update/4` (included in CL when defining `update_population/3`). Note that there must be one definition of `update/4` for each component in a model whereas other predicates must be defined only once.

5.4 Using the model

Finally, the user can run the model generated on the previous phase. The code generated was recorded and it is loaded into a current Prolog session. A data-file is generated from each simulation, with population values for each component being recorded at every time step. An external Unix package is used to generate a graph.

The Prolog code shown on Section 3.3 is an actual (yet not complete) listing produced by TeMS. Different choices during the description of the model would lead to the generation of different programs. Using again the example shown on Section 3.3, if the user decides that wolf population should not be disaggregated in the way described there, the next options presented by TeMS during the model description stage would suppress those which are typical of disaggregated populations (*e.g. ageing*) and the code generated would reflect that. Some predicates might be maintained despite the change in population disaggregation (*e.g.* `rep_rate/4`), that is so because pre-defined processes are supplied for both disaggregated and non-disaggregated populations and their respective formulas are dealt with by TeMS' Prolog meta-interpreter. The following is the version of `update/4` for the case we described here.

```
%
% update each component
%
update(A, B, C, D) :-
  C=[J,_,I],
  J=moose,
  rep_rate(A, B, C, I, H),
  predation1(A, B, C, H, G),
  hunting(A, B, C, G, F),
  ageing(A, B, C, F, E),
  adj_values(E, D).
update(A, B, C, D) :-
  C=[I,_,H],
  I=wolf,
  rep_rate(A, B, C, H, G),
  starvation(A, B, C, G, F),
  adj_values(E, D).
```

The runnable Prolog code for the model defined (except for some utilities) is viewable within TeMS and Figure 6 shows how the user see the results of a simulation plotted in a graph. The model represented there is the same described in Section 3 and, as the model generated by "human construction" on the first stage of our project, its population graph shows a behaviour similar to that generated by Crête's model.

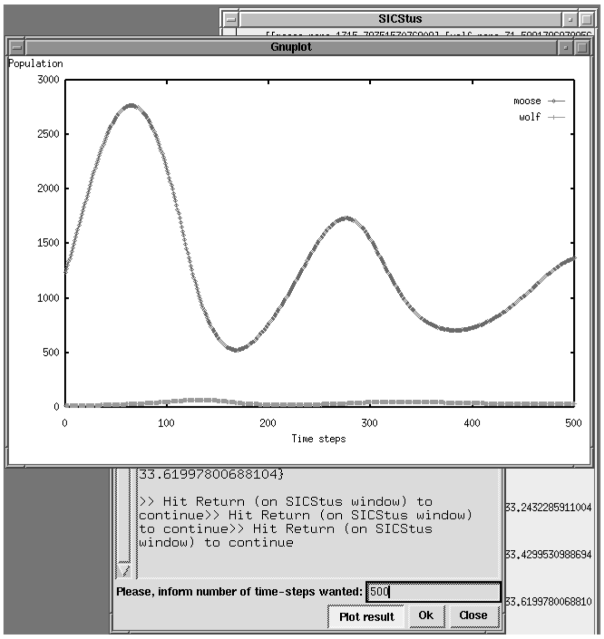


Figure 6: Running the model

6 Conclusion

The aim of this paper is to demonstrate what happens when we take a generic method for structuring formal specifications (in this case techniques editing) and attempt to tailor it to the design of a class of specifications in a target domain (in this case population dynamics modelling). A fundamental difficulty in our target domain is that the styles of description which population modellers understand are different from those needed to control specification synthesis using our chosen formal method. Consequently, to tackle the problem we:

- Identify domain-specific techniques which use the parameterisation method of generic techniques but which contain information specific to the population dynamics domain (Section 4).
- With these techniques in mind, provide a problem description language which uses concepts from population dynamics, and construct an interface which allows these concepts to be supplied (Section 5.2). This enables the style of problem description to be disconnected, initially, from the style of definition of the domain-specific techniques.
- Build an automated system which ensures appropriate parameterisation of the domain-specific techniques based on the population dynamics problem description (Section 5.3). This connects the problem description to the techniques needed for model generation. It can be automatic because we employ restricted languages for both problem description and techniques.
- Supply tools for executing the specification in styles familiar to those in

the domain (Section 5.4). This gives those in the domain an opportunity to check whether the model they have received is the one they expected.

The first step in the list above is perhaps the most difficult because it isn't always easy to identify appropriate domain-specific techniques. Our way of starting to acquire the necessary knowledge is to begin with a sample model (from the modelling literature) and to build the techniques needed to construct it. By making these techniques as general as possible (while still being recognisable within the domain) we generalise from a particular example to a class of models - in our case these are population dynamics models with any number of interacting populations; a hierarchically disaggregated population structure; and process definitions controlling the interaction between populations. Since the main boundaries of this class can be described in terms of the domain, rather than more abstract mathematical limitations, it is easier to make clear to modellers whether or not the system is appropriate to their needs.

References

[Bowles 94] A. Bowles. A techniques editor for prolog novices. Internal software report, Department of Artificial Intelligence, University of Edinburgh, 1994.

[Bowles *et al.* 94] A. Bowles, D. Robertson, W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying prolog programming techniques. *International Journal of Human-Computer Studies*, 41:329–350, 1994.

[Crête *et al.* 81] M. Crête, R.J. Taylor, and Jordan P.A. Simulating conditions for the regulation of a moose population by wolves. *Ecological Modelling*, 12:245–252, 1981.

[Hutchins *et al.* 86] E.L. Hutchins, J.D. Hollan, and D.A. Norman. Direct manipulation interfaces. In D. Norman and S. Draper, editors, *User-centered system design*, pages 87–124. Laurence Erlbaum Associates, 1986.

[Kirschenbaum *et al.* 89] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling. Skeletons and techniques for prolog programming. Technical Report TR–89–170, Case Western Reserve University, 1989.

[Kirschenbaum *et al.* 94] M. Kirschenbaum, S. Michaylov, and L.S. Sterling. Skeletons and techniques as a normative approach to program development in logic-based languages. Technical Report 25, Ohio State University – CISRC, May 1994.

[Muetzelfeldt 95] R. I. Muetzelfeldt. A framework for a modular modelling approach for agroforestry. *Agroforestry Systems*, 30:223–234, 1995.

[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. Springer-Verlag, Berlin, 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.

[Sterling & Kirschenbaum 93] L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, chapter 6, pages 127–140. John Wiley & Sons Ltd, 1993.

[Vargas-Vera 95] M. Vargas-Vera. *Using Prolog techniques to guide program composition.* Unpublished PhD thesis, University of Edinburgh, 1995.

[Vargas-Vera *et al.* 93] 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 93] W. W. Vasconcelos. Designing prolog programming techniques. In Y. Deville, editor, *Logic program synthesis and transformation (LOPSTR 93) – 3rd International Workshop, Louvain-la-Neuve*, pages 85–99. Springer-Verlag, 1993.

[Vasconcelos 94] W. W. Vasconcelos. Extracting prolog programming techniques. In T. Pequeno and F. Carvalho, editors, *Proceedings of the XI Brazilian Symposium on Artificial Intelligence*, pages 269–283. Brazilian Computer Society, 1994.

[Weiser 84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.