# Multilevel Logic Programming for Software Engineering: Modelling Version Selection in Software System Configuration Management

**Mária Bieliková and Pavol Návrat**

Slovak Technical University, Dept. of Computer Science and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia
Tel.: (+ 42 7) 791 395, Fax: (+ 42 7) 720 415
E-mail: {bielikova,navrat}@elf.stuba.sk

This research has been motivated by the need to provide more lucid and effective means for describing and structuring all the various knowledge related to certain software engineering tasks, such as a version selection. Software configuration management (SCM) is one of the areas closely related to supporting engineering approaches to software development. Our objective was to achieve an effective automated version selection - which is an important task in SCM - by means of incorporating heuristic filters in that process. Our approach offers means of abstraction for expressing various kinds of knowledge involved in the related process. It also offers techniques for structuring them according to both generality levels and to knowledge content, i.e. meta-levels. To illustrate these ideas, we show how multilevel programming can be used to model a method for version selection.

*Keywords:* software configuration management, version control, heuristic selection, module, multilevel logic programming, Prolog

## 1 Introduction

The problem of version selection is a topic of much interest in the recent research within the area of software configuration management. Solving it is important for achieving quality of the configuration being built, but it also influences efficiency of the process of building a software system configuration.

An approach often used to identify versions of software components is the use of attribution models where a version is described implicitly by stating properties which it should have (Cohen et al, 1988; Estublier, 1992; Leblang, 1994; Zeller and Snelting, 1995). In a change oriented version model (Lie et al, 1989) the options play a role similar to the attributes.

The component's attributes can be e.g. date of formation, author, status. Using such attributes, we can formulate the following requirement:

> *select a version which is being elaborated (status = check_out) by the author, who is the current user; if there is not such a one then select the most recent completed (status = check_in) version which has been already tested.*

Frequently used approach to version selection is to use conditions restricting properties of versions (Tichy, 1988; Cohen et al, 1988; Estublier, 1992). The conditions are often represented by a logic expression. For example, an expression

$(operating\_system = DOS \ \wedge \ communication\_language = Slovak)$

identifies all such versions which can run under DOS operating system and the communication with the user is in Slovak language. The ADELE system (Estublier, 1992) is an example illustrating this view of version selection.

The language of logic expressions is sometimes enhanced by allowing defaults, conditional selections, and by introducing three-valued logic (Nicklin, 1991) (i.e. *true, false, undefined*).

For modelling version sets, Zeller and Snelting (Zeller and Snelting, 1995) propose a unified approach based on feature logic. Version sets are identified by their features, that is, a boolean expression over (*name* : *value*) attributes. This approach subsumes all the above mentioned approaches to identify versions of components.

In another work (Bernard et al, 1987), version selection is based on logical conditions referring to values of attributes, too. Here, preferences can be specified as well. Preferences are in fact logical conditions which act as filters.

The system SHAPE (Mahler, 1988) uses a similar way of selecting versions. It is based on using a selection rule. The selection rule is a named sequence of alternatives. Predicates in alternatives allow expression of requirements with respect to the object attributes. If no selection rule is specified, the fixed default rule is active which is the same as in Make (*"select the busy object in the current directory"*).

System GYPSY (Cohen et al, 1988) uses predicate to obtain a subset of the component versions. In case the resulting set has more than one element, the final selection is the result of applying a single default rule, i.e. *"select the most recent version"*. In GYPSY, no precaution was made to handle the case when some or all of the filters failed to produce a nonempty set.

Another frequently used approach is to use rules. For example in the system DSEE (Leblang and Chase, 1987) there is defined a set of rules which are interpreted sequentially until the component being sought is selected. The language for writing rules allows defining default rules, dynamic rules (e.g. select the most recent version), and conditional rules (if-then). Despite the fact that such kinds of rules allow powerful means of selection, their power is limited in the DSEE system by the fact that the set of attributes is set in advance.

We have proposed a method for version selection which offers a framework for specifying various heuristics describing which versions are to be preferred. Our goal in this paper is not to present the method for version selection (Návrat and Bieliková, 1996), but rather to describe a representation framework for such a kind of engineering technique in general, and the method of version selection in particular. Our approach aims to achieve following properties regarding to version selection: (1) recognition of hierarchical levels within the relevant knowledge, (2) structuralisation of knowledge which enables better understanding, readability and maintenance, (3) extensibility (possibly to improve version selection as the field evolves), (4) easy combining of different techniques by specifying alternative modules and meta-modules which control their use in the selection process, and (5) development and experimental testing of new approaches (e.g. new heuristic functions which are domain dependent).

The paper is organized as follows. In Section 2, we give an overview of a multilevel logic programming. In Section 3, we describe our approach to version selection. We describe the main components in the version selection process in terms of modules at multiple levels in Section 4. Section 5 presents an application of the multilevel programming technique to representation of the program for version selection using modules. We show how modules can be combined together. The paper closes with our conclusions which also summarize the results.

## 2 Multilevel logic programming

As we mentioned already above, our research has been motivated by the need to extend and conveniently represent a particular software engineering technique. We concentrate on the way how to represent different approaches in a uniform (logic) formalism and how to combine them. A logic formalism, like any declarative formalism in general, is an excellent tool to express algorithmic knowledge at a very high level of abstraction. It is convenient for the developer because it aids to reducing the descriptive complexity, it supports rapid prototyping, it makes future modifications easier etc. We have chosen Prolog as a particular logic language to provide an evidence of suitability in this kind of tasks because it is widely known and used (Rosenman and Gero, 1994). The choice of Prolog is further supported by the possibility of metaprogramming in Prolog.

Despite several widely recognized advantages of logic programming such as declarative semantics, or mechanisms of unification and deduction, the extent of its suitability for development of large software systems is relatively limited. One of the crucial problems is the lack of concepts, and concequently of language constructs, for structuring of programs, modularity, sharing and hiding, etc. All of them are important means to manage complexity of a particular domain.

The desire to have in Prolog means for dividing a software system into smaller relatively separated and independent units with transparent minimal interfaces was responded by several authors. Separated logic databases are called modules (Giordano et al, 1992; Sannella and Wallen, 1992; Kwon et al, 1993), theories (McCabe, 1992), units (Lamma et al, 1991). Several authors (Kowalski, 1979; Zaniolo, 1984; Gallaire, 1986; Lamma et al, 1991; McCabe, 1992; Xu and Zheng, 1995) applied concepts of the object oriented programming to achieve structuring of logic programs.

Our approach is based on the reflection technique as elaborated by Lamma, Mello, and Natali (Lamma et al, 1991) who used reflection for combining Prolog databases through contexts and inheritance.

### 2.1 Reflection

Rather than (meta-) interpreting the overall behaviour of an abstract machine, some parts of the machine's state are made available to be accessed and manipulated directly through reflection mechanism. The reflection mechanism switches the computation from the (object-) level to the introspective (meta-) level domain (upward reflection) and vice versa (downward reflection) (Lamma et al, 1991).

The object level machine's visible state ought to be chosen to suit needs of the problem domain. Let us assume the visible state is the triplet $(CM, CG, AUX)$, where $CM$ is the current module, $CG$ is the current (sub-)goal, and $AUX$ is a term representing auxiliary information. This is one particular choice of the level of abstraction for the reflective operations.

Both the levels of a program are represented in the same way - as modules. In fact, this allows us to apply the reflection concept to a program designed into many levels, as we shall show later. Connection between an object level module $M$ and a meta-level module $Meta$ is defined by the relation *connect*. If *connect(Meta)* can be proved in module $M$ then module $Meta$ is a meta-module of $M$.

Now, let us assume the computation takes place at the object level in a module $M$. When a module $Meta$ becomes a meta-level module of $M$ by proving the goal $connect(Meta)$, there is provided an implicit upward reflection. An attempt to prove $reflect\_up([CM, CG, AUX])$ shifts the computation to the meta-level where the visible state of the abstract machine is explicitly available through the triplet. The attempt can either succeed or fail. If it fails, the failure is reported in the object level in the usual way.

When the computation takes place at the meta-level, an explicit downward reflection is attempted by the goal $reflect\_down([CM1, CG1, AUX1])$ (see Figure 1). This causes an object level computation to start in the module $CM1$ aiming to prove the goal $CG1$. Again, the attempt can either succeed or fail, similarly to the above case. If it succeeds, new visible state is reflected up by an implicit upward reflection. In such a way, results of the object level computation become available at the meta-level. If it fails, the failure is reported to $reflect\_up$ goal at the meta-level which fails, too.

We have deliberately not explained the role of the third part $AUX$ of the visible state.

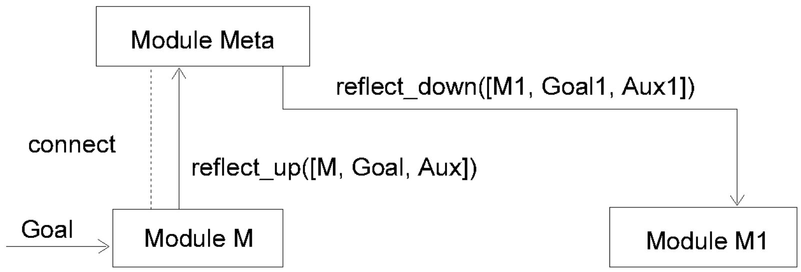


Figure 1: Proof of a goal in the module $M$.

Syntactically, it is a term to be processed at a meta-level. Semantically, the choice is left open to be determined in accordance with the application domain of problems being solved.

The reflection technique involving the visible state of the abstract machine allows to tackle various problems, such as (i) describing various ways of combining of modules, (ii) modifying dynamically the strategy of selecting modules for interpretation by changing the connection to the meta-module.

### 2.2 Design of a Multilevel Logic Program

Strictly speaking, we have discussed only two level logic programs so far. However, it is apparent that the concept of introducing a meta-level to a given program level can be applied to the meta-level as well, yielding meta-meta-level, etc. While conceptually this appealing idea is quite simple, there are certain more practical issues which require careful consideration. In this section, we shall present our approach to defining such multilevel logic programs.

We wish to underline that what we are facing at this stage is in some sense a design problem. It requires design decisions, based on considerations of various options. The problem may not have a unique solution.

A multilevel logic (Prolog) program is a modular logic (Prolog) program in which modules can be mutually interconnected by defining the relation *connect*.

The relation *connect* is used to establish program levels. At the lowest (i.e., object, or program) level, program modules are defined. At higher levels, modules are defined which determine the way goal is proved in program modules. Both program and meta-level modules are represented in the same way, and therefore further meta-levels are naturally possible.

In an appendix we present inference rules which are used by the abstract machine to process a multilevel logic program. They serve as a basis for implementing a multilevel Prolog program interpreter. The proof at the meta-level has the same procedural semantics as at the program level. If a module $M$ is *connect*ed to some other module, upward reflection occurs to next higher level. In particular, upward reflection can occur during an attempt to satisfy a goal *reflect_down*, too. However, if there is not *connect*ed any module to a given module during an attempt to satisfy a goal *reflect_down*, reflection occurs towards a level determined by a parameter of the term *reflect_down*.

## 3 An outline of our approach to version selection

When either more than one version complies with the requirements, or none of the versions does, difficulties in the process of configuration building can arise. When analyzing the problem of version selection, there are many similarities to be observed to problems approached by artificial intelligence techniques. Specifically, to evaluate the alternatives a heuristic information can be employed.

We express requirements for version selection as a sequence of heuristic functions which reduce the set of suitable versions. A relative importance of a given evaluating criterion can be expressed by modifying an order in which the heuristic functions are applied.

We have divided the requirements for version selection into two parts:

- *A necessary selection condition* must be satisfied by every version selected as a potential candidate. The condition can be expressed by a heuristic function which maps a set of all versions into a set of admissible versions,

- *A suitability selection condition* is used in step by step reduction of the set of admissible versions aiming to select a single version. The condition is represented by a sequence of heuristic functions.

One example of requirements for version selection may be as follows:

- *The necessary selection condition*
  $h_0$ : operating_system = DOS $\wedge$ communication_language = Slovak
- *The suitability selection condition*
  $h_1$ : problem_type = design $\vee$ algorithm = simple
  $h_2$ : programming_language = Prolog
  $h_3$ : prefer version with a smaller number of defined architectural relations with other components

The heuristic functions represent knowledge about the degree of suitability of the respective versions. They refer to properties of versions as defined by their attributes. A heuristic function can often be evaluated separately for each version.
It can have the form

$h : V \rightarrow \{satisfies, does\_not\_satisfy\}$,

where $V$ is a set of versions and $h$ is a heuristic function. Requirements for version selection formulated in a similar fashion can be found in many existing systems, e.g. (Tichy, 1988; Cohen et al, 1988; Estublier, 1992).

Heuristic functions can also express a knowledge captured during the software system development. Examples of such knowledge are

- Prefer a version with the greatest number of defined attributes,
- Prefer a version included in the greatest number of formed configurations,
- Prefer a version which is involved in the least number of architectural relations with other components (in the context of the whole configuration),
- Prefer a version which is involved in the least number of architectural relations with components which have not been included in the configuration being built.

The heuristics for version selection described above, and other similar ones cannot be described by a function of the form $h : V \rightarrow \{satisfies, does\_not\_satisfy\}$ (which can be expressed by a logic expression, with atoms representing relations over attributes of components). These are the properties of versions which can only be investigated on sets of versions as a whole. Therefore we define heuristic functions to be of the form $h : 2^V \rightarrow 2^V$, where $V$ is a set of versions and $h$ is a heuristic function.

Let us sketch the approach to version selection (Bieliková and Návrat, 1996) which makes use of such heuristic functions. First, the set of all versions will be reduced by applying the necessary selection condition into a set of admissible versions. Next, version selection continues as a successive application of the heuristic functions from the suitability selection condition, serving as filters until all the functions in the sequence are exhausted or until by applying one of them yields a one element set, the element being the desired version.

If, after applying all the filters we get a set of more then one version, the final choice must be made in a different way. Some default procedure must be applied e.g., (i) selection based on an order of occurence, (ii) random selection, (iii) selection based on a decision of the software engineer.

Even in this case the set may indeed become reduced in that the number of elements is smaller than in the original set of all versions. Of course, it depends on the heuristic functions and actual properties of versions.

### The method for version selection

**Input** to the method is:

- a set $M$ of all available versions
- requirements on version selection
  - a necessary selection condition represented by a heuristic function $h_0$,
  - a suitability selection condition represented by a sequence of heuristic functions $[h_1, h_2, \ldots, h_n]$, where $h_j : 2^M \rightarrow 2^M,\ 0 \leq j \leq n$

**Output** from the method is "the most suitable" version $v$, $(v \in M)$, or failure.

The method can be described by the following steps:

1. Apply the necessary selection condition to reduce the set $M$ of all available versions into a set of admissible versions: $suit_0 = h_0(M)$.
   If $suit_0 = \emptyset$ then **halt**, the method has not been successful.
   If $suit_0 = \{v\}$, i.e. the set of admissible versions has exactly one element then **halt**, the method has been successful and the output is the version $v$.
   Otherwise, continue.
2. Apply the heuristic functions $[h_1, h_2, \ldots, h_n]$ in the order of their appearence to the actual set of admissible versions:
   (a) $j := 1$
   (b) apply the heuristic function (filter) $h_j$ to the set $suit_{j-1}$ forming a set
   $$suit_j = \begin{cases} h_j(suit_{j-1}), & \text{if } h_j(suit_{j-1}) \neq \emptyset \\ suit_{j-1}, & \text{otherwise} \end{cases}$$
   (c) If $suit_j = \{v\}$, i.e. the actual set of admissible versions has exactly one element then **halt**, the method has been successful and the output is the version $v$.
   (d) If $j = n$ then **halt**, the method has been only partly successful so far. To determine its output, a version $v \in suit_n$ shall be found using some default way.
   (e) $j := j + 1$ and continue with 2b.

## 4 Modularization of knowledge related to version selection

Multilevel Prolog programming can be used with advantage whenever problem domain knowledge is available. After careful analysis, several layers of knowledge can usually be recognized. There is knowledge of the problem itself. There is also another kind of knowledge which describes structure and properties of objects, relations, including ways of solving problems. This is meta-knowledge. It can and, in fact, it should be structured just as any other knowledge should. Knowledge is better captured, understood, manipulated, and applied if structured into interconnected units. But supposing e.g. there are several problem solving methods defined at the meta-level, it is very likely that there is also another kind of knowledge available, viz. the one evaluating their respective suitability, applicability, etc. This is already a meta-meta-level knowledge. It can be extremely useful in deciding which method to apply to a particular problem instance. For example, the Specification-Conceptualization-Operationalization method (Akkermans et al, 1993) is the method for constructing problem solving methods, thus being a meta-meta-knowledge. Control knowledge which is used to determine how inferences are sequenced is also to be regarded as one level higher than meta-knowledge. As another example (Ohsuga, 1993) modelling is presented as a basis for problem solving, placing it on the meta-meta-level. This knowledge can be used to build problem solving methods (meta-level) for a particular problem (object level).

It is quite clear that structuring of knowledge according to such content, i.e. semantically related hierarchies can be potentially at least as fruitful as those more synctically oriented approaches.

Here, we try to identify the main layers (modules) of the knowledge hierarchy of version selection. We follow the method for version selection described above.

In describing the particular layers, we start with the software components (versions) which are to be considered the lowest level, i.e. **level 1**.

At the **level 2** there are strategies of retrieving attribute's value along with determining what kind of attribute it is. The way attribute's value is retrieved depends on its nature. To retrieve, we can make use of an inheritance of values in a hierarchy of elements of a software system. A reasonable hierarchy of elements of a software system could be based on notions of *family–variant–revision*. Another hierarchy is defined by such architectural relations as *includes, depends_on*. Depending on the nature of a particular attribute, various ways of inheritance could be used. For example, attributes *date, time_of_creation* should be defined for each software component. Therefore, inheriting such attributes does not make sense. Other attributes, such as *author, operating_system, depends_on, includes* can be inherited through unification. Value of an attribute of an element above in the hierarchy can be inherited when the present component does not have a value defined for that attribute. Still another way is an unrestricted inheritance.

Strategy of retrieving attribute's value comprises also the way undefined software component attributes are interpreted. One possibility is to assume that the component does not depend on attribute's value when it is not defined. Such component must satisfy any elementary condition referring to the attribute with the undefined value.

At the level 2, there are also the heuristic functions. Heuristic functions represent our knowledge of versions' properties. Moreover, the properties can be investigated for a set of versions as a whole. They serve as a selection condition. Expressing the condition in form of a logic expression is just one possible alternative.

Next level, i.e. **level 3** comprises definitions how each heuristic function is to be evaluated. Actual evaluation depends on a 'type' of the heuristic function. For example, when the heuristic function is represented by a logic expression where atoms denote relations over attributes of particular versions it can be applied separately to each element of the actual set of versions.

Method for version selection is represented at the **level 4**. At this level the way of heuristic function application is defined. Here alternative methods for version selection can be represented as well.

The version selection knowledge hierarchy with its modules outlined above can be depicted as in Figure 2.

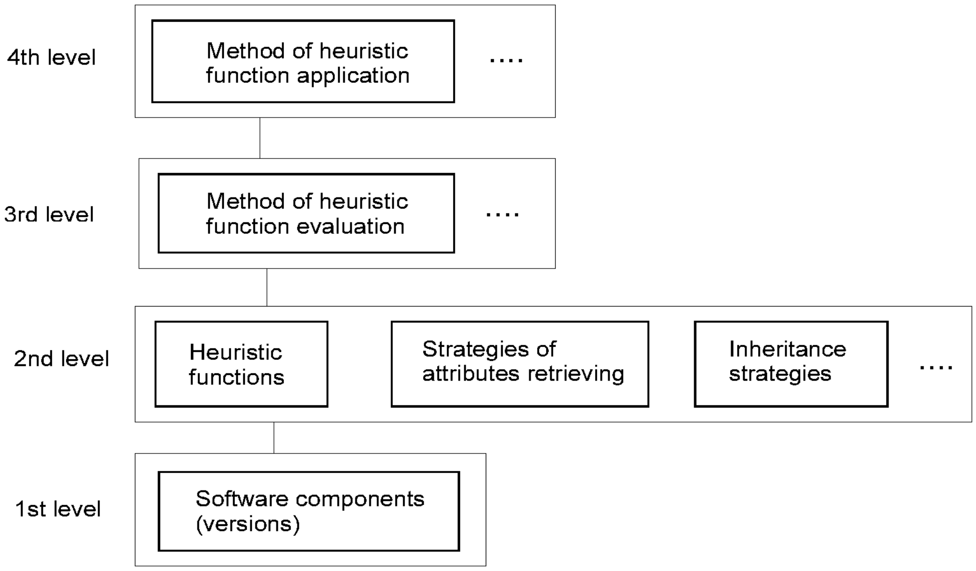


Figure 2: Version selection knowledge hierarchy.

Our approach which is based on using multilevel logic programs allows to represent modules as well as relations between them by means of a uniform formalism (i.e., logic). In the next section, we present an application of multilevel logic (Prolog) programming technique for version selection representation.

## 5 Applying the multilevel programming: an example of the method for version selection

In order to present the way the system of version selection is represented, modules belonging to the knowledge hierarchy defined above are briefly described. Their representation in multilevel Prolog is illustrated.

The above described method for version selection consists basically of two steps: application of a necessary selection condition, and application of a suitability selection condition. The method for version selection (level 4) can be expresed in Prolog as follows. Version selection is attempted by writing a goal

```
select_best([Necessary_cond, Suitability_cond], Versions, Best).
```

An attempt to satisfy such a goal takes place within clauses defined in a module em version_selection. Input to the method is represented by the first and second parameters. Third (output) parameter is the selected version.

```
version_selection ismod
 { select_best([Necessary_cond, Suitability_cond], Versions, Best) :-
          % the necessary condition application
      test_cond(Necessary_cond, Versions, OK_Versions),
          % the suitability selection condition application
      select_best_el(Suitability_cond, OK_Versions, OK_Versions, Best).

  % empty actual set of possible versions after application of the last filter,
  % i.e. by application of the actual filter all versions have been deleted and
  % the filter is the last one in the suitability selection condition
  % —> return the previous set of possible versions - HALT
  select_best_el([], Prev_set, [], Best) :-
      default_selection(Prev_set, Best).

  % empty actual set of possible versions
  % —> return to the previous set of possible versions and
  %     apply the next filter
  select_best_el([Filter1 | Rest], Prev_set, [], Best) :-
      test_cond(Filter1, Prev_set, Actual_set),
      select_best_el(Rest, Prev_set, Actual_set, Best).

  % one version in actual set of possible versions
  % —> this is the best version - HALT
  select_best_el(_, _, [One], One).

  % recently used filter is the last and
  % the actual set of possible versions contains more than one element
  % —> successively return elements from actual set of possible versions
  %     by delault selection - HALT
  select_best_el([], _, [Version1, Version2 | Rest],  Best) :-
      default_selection([Version1, Version2 | Rest], Best).

  % else apply the next filter
  select_best_el([Filter1 | Rest], _, Actual_set, Best) :-
      test_cond(Filter1, Actual_set, New_set),
      select_best_el(Rest, Actual_set, New_set, Best).
```

```
% if the previous set of possible versions is not equal actual set
% —> return difference as an alternative solution
select_best_el(_, Prev_set, Actual_set, Best) :-
      diff(Prev_set, Actual_set, List_of_best),
      default_selection(List_of_best, Best)   }.
```

The actual selection of the "best" version is within the reponsibility of the predicate *select_best_el/4*. It uses an auxiliary parameter (the second one) that represents a set of versions after the last but one reduction (i.e., an application of a filter). This is required in case the last applied filter rejects all the versions.

The predicate *select_best_el/4* can be satisfied repeatedly. When backtracking, it returns subsequently less suitable versions, as long as they exist. As a consequence, the predicate *select_best/3* is repeatedly satisified, too, as long as there exist versions not selected so far that satisfy the necessary selection condition.

In case the application of all the filters results in a set of more than one admissible versions, some default method is applied. The simplest one would be selecting according the order in which they are written in the list that represents the set. The method would be in that case implemented by a predicate *member/2*. Other methods are also possible, such as random selection, or selection in interaction with software engineer.

The method for default version selection is defined at a meta-level. In such a way, different methods can be applied according to an actual requirement simply by switching the *connect*ion between the level of version selection and the meta-level of its defaults. The latter level can be defined as follows:

```
meta_order_selection ismod
 { reflect_up([_, default_selection(List, Element), AUX]) :-
         reflect_down([methods_of_selection,
                                 member(Element,List),AUX])      }.

meta_question_selection ismod
 { reflect_up([_, default_selection(List, Element), AUX]) :-
         reflect_down([methods_of_selection,
                                 question(List, Element), AUX])  }.

methods_of_selection ismod
 { member(X, [X|_]).                    %order selection
   member(X, [_|R]) :- member(X, R).

   question(List, Element) :- .....     %selection by a question }.
```

We present the module for selection according to the order (module *meta_order_selection*) and the module for selection in interaction with software engineer (module *meta_question_selection*). Both the modules use predicates defined in the module *methods_of_selection*.

In this case, the relation between the module *meta_order_selection*, and the module *meta_question_selection* represents simple procedure call. The *version_selection* module is connected with one of these modules by the relation *connect*. When attempting to satisfy a goal, there occurs a reflection and a goal with the functor, *reflect_up* is proved as described in Figure 3. In the above case, there occurs a reflection upwards to the level where knowledge on how to select is written, i.e. to the *meta_random_selection* module.

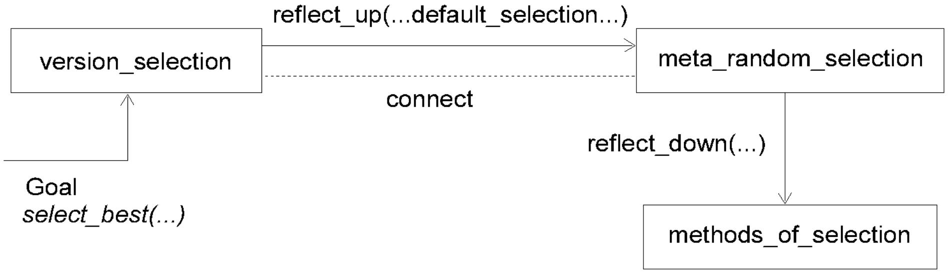


Figure 3: Proof of the goal *select_best*.

Now let us turn our attention to implementing heuristic functions evaluation (level 3). A heuristic function is applied by writing a goal *test_cond/3* (cf. module *version_selection* above). Definition of the predicate *test_cond* should be separated, however, mainly because there is a possibility that heuristic functions are of various kinds requiring different ways of processing. An attempt to satisfy a goal

```
test_cond(Cond, Versions, OK_Versions).
```

causes always an upward reflection to a meta-level represented by a module *meta_test_cond*. Here a decision is made about the way a filter is applied according to what kind of filter it is. All of this requires, however, that a connection has been established between the program level represented by the module *version_selection* and the above mentioned meta-level.

```
meta_test_cond ismod
 {     % heuristic function represented by logic expression;
       % evaluation is defined at object level (module test_elem_cond)
   reflect_up([_, test_cond(Cond, N, OK_N), AUX]) :-
       is_logic_expression(Cond), !,
       findall(E, (member(E, N), reflect_down([test_elem_cond,
                                  test_cond(Cond, E), AUX])), OK_N).

       % in this case a heuristic function is defined only at meta-level,
       % i.e. at object level no computation is needed
       % downward reflection is done implicitly after success of reflect_up
   reflect_up([_, test_cond(max_def_attr, N, OK_N), _]) :-
       obtain_num_of_def_attr(N, List_of_num),
       max(List_of_num, Max),
       find(N, Max, OK_N).
             ......
       % similarly for other heuristic functions       }.
```

This approach has an advantage in separating definitions of heuristic functions from their applications. They can easily be accessed, modified, or enhanced.

An evaluation of a heuristic function written as a logic expression is implemented by a simple interpreter of logic expressions:

```
test_elem_cond ismod
 { test_cond(true, _).
   test_cond(false, _) :- !, fail.
   test_cond((C1 and C2), N) :- test_cond(C1, N) , test_cond(C2, N).
   test_cond((C1 or C2), N) :-  test_cond(C1, N) ; test_cond(C2, N).
   test_cond(not(C), N) :- test_cond(C, N), !, fail.
   test_cond(not(C), _).
   test_cond(Elem_C, N) :- Elem_C =.. [Rel, Attr, Value],
                           obtain_value_of_attr(N, Attr, Value1),
                           compare(Rel, Value1, Value)                }.
```

The way of retrieving attribute's value is defined in the knowledge hierarchy at the level 2 along with determining what kind of attribute it is. It is supposed that each entity of a software system, be it a family, a variant, or a revision, is represented in a separate module (level 1). Attribute named *attr* with value *val* is written as a fact with a functor *attr/1* and a single parameter *val*. In a similar way, relations in a software system are represented by facts with relation names as functors, and with single parameters denoting entities being related to.

```
meta_inheritance ismod
 { reflect_up([_, obtain_value_of_attr(N, Attr, Val), AUX]) :-
         attr_type -> non_inherit(Attr), !,
         Atom =.. [Attr, Val],
         reflect_down([N, Atom, AUX]).
   reflect_up([_, obtain_value_of_attr(N, Attr, Val), AUX]) :-
         attr_type -> overwrite_inherit(Attr), !,
         ancestor(N, Ancestor),
         Str_template =.. [Attr, _],
         is_clause(Ancestor, Str_template), !,
         Atom =.. [Attr, Val],
         reflect_down([Ancestor, Atom, AUX]).
   reflect_up([_, obtain_value_of_attr(N, Attr, Val), AUX]) :-
         attr_type -> full_inherit(Attr), !,
         ancestor(N, Ancestor),
         Atom =.. [Attr, Val],
         reflect_down([Ancestor, Atom, AUX]).

   ancestor(N, N).
   ancestor(N, Ancestor) :-
         ancestor_rel(N, Version),
         ancestor(Version, Ancestor)           }.
```

```
attr_type ismod
 { non_inherit(date).              non_inherit(time).
       ......
   overwrite_inherit(author).      overwrite_inherit(op_syst).
   overwrite_inherit(depends_on).  overwrite_inherit(consist_of).
       ......
   full_inherit(project).          full_inherit(created_from).
       ......              }.
```

In the example above, standard Prolog mechanism is used to retrieve values of attributes. When there occurs reflection to a module which defines properties of the module, it may very well happen that an attempt to retrieve a value of an attribute with an undefined value fails.

Our solution is the following: for all the modules defining properties of elements of a software system, we define a connection to a meta-level *meta_solve* which determines the way values of attributes are retrieved.

```
meta_solve ismod
 {   reflect_up([Module, Goal, AUX]) :-
           reflect_down([Module, Goal, AUX]).
     reflect_up([Module, Goal, AUX]) :-
           Goal =.. [Attr, undefined]            }.
```

It is supposed as we stated above that attribute named *attr* with value *val* is written as a fact with a functor $attr/1$ and a single parameter $val$. The first clause attempts to retrieve the value in the defined $Module$. When it fails, the attribute value is not defined (the value $undefined$ is returned).

Furthermore, it is necessary to include into evaluation the specific knowledge that an undefined attribute satisfies any condition (cf. meta-module *meta_compare*, predicate *compare/3*).

```
meta_compare ismod
 {   reflect_up([_, compare(_, undefined, _), _]) :- !.
     reflect_up([M, compare(Rel, Val1, Val), AUX]) :-
           reflect_down([M, compare(Rel, Val1, Val), AUX])    }.
```

Predicate $compare/3$ is supposed to be defined in the program module which is *connect*ed to the module *meta_compare* (e.g. in the module *test_elem_cond* defined above).

## 6 Conclusions

We have presented our proposal of a multilevel logic programming technique. Our approach is based on the reflection technique (Lamma et al, 1991). The technique has been implemented in Prolog. The prototype is meta-interpreted.

The proposal of a multilevel logic programming technique falls into the area of the methods of structuring of logic programs (Bugliesi et al, 1994). Program can be divided into modules. Moreover, it can be organized into levels. Dividing programs into modules is the well known technique which helps cope with the complexity of the problem. Frequently, development of modules refers to levels of abstraction.

Organizing programs into (meta-)levels refers to knowledge content. A need to structure programming knowledge according not only to abstraction and generality levels, but to meta-levels as well has been stressed by Návrat (Návrat, 1996). The fact that also meta-knowledge can be written in modules aids to modifiability and reuse. Meta-level can be used to write a module defining various ways of processing goals from the object level. Meta-meta-level would be suitable to write a module defining method of selecting the proper way of processing.

From a more general perspective, the role of meta-knowledge in the context of knowledge structuring was recognized relatively long time ago. For example, Coyne and Gero (Coyne and Gero, 1986) proposed to describe domain knowledge by grammars and noted that control knowledge could be made explicit in the form of meta-grammars.

Our approach allows also to connect an object level program to several modules at the same meta-level. Similar question was tackled by Sterling (Sterling and Beer, 1989) who proposed two strategies of combining meta-interpreters.

In the paper, we have described application of multilevel programming to the problem of version selection. Another application that we attempted was for representation of a rule-based query optimizer for object-oriented databases (Bieliková et al, 1995). Multilevel logic programming is used to model both query rewriting and planning, as well as search strategies.

## References

H. AKKERMANS, B. WIELINGA, G. SCHREIBER. Steps in constructing problem solving methods. In N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, and Y. Kodratoff, editors, *Knowledge Acquisition for Knowledge-Based Systems - Proc. 7th European Workshop EKAW'93*, pages 45–65. Springer-Verlag, 1993. LNCS No. 723.

M. BIELIKOVÁ, B. FINANCE, L. GARDARIN G., MOLNÁR, P. NÁVRAT, M. SMOLÁROVÁ, ZHAO-HUI TANG. Modeling a query optimizer with multi-level logic programming. Final report of Copernicus project No. CP 93:6638, STU Bratislava, CNRS-PRiSM Versailles, 1995.

M. BIELIKOVÁ, P. NÁVRAT. A knowledge based method for bulding a software system configuration. *Knowledge Based Systems*, 9(1):61–65, 1996.

Y. BERNARD, M. LACROIX, P. LAVENCY, M. VANHOEDENAGHE. Configuration management in an open environment. In *Proc. 2nd European Software Engineering Conference*, pages 35–43. Springer-Verlag, 1987.

M. BUGLIESI, E. LAMMA, P. MELLO. Modularity in logic programming. *Journal of Logic Programming*, 19(20):443–502, 1994.

R.D. COYNE, J.S. GERO. Semantics and the organization of knowledge in design. *Design Computing*, 1:68–89, 1986.

E.S. COHEN, D.A. SONI, R. GLUECKER, W.M. HASLING, R.W. SCHWANKE, M.E. WAGNER. Version management in Gypsy. In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, pages 201–215, Boston, 1988. ACM Press.

J. ESTUBLIER. The Adele configuration manager. Technical report, L.G.I., Grenoble, 1992.

H. GALLAIRE. Merging objects and logic programming: Relational semantics. In *Proc. of AAAI Conference*, 1986.

L. GIORDANO, A. MARTELLI, G.F. ROSSI. Extending Horn clause logic with modules constructs. *Theoretical Computer Science*, 1992.

K. KWON, G. NADATHUR, D.S. WILSON. Implementing a notion of modules in the logic programming language λProlog. In Lamma and Mello (Lamma and Mello, 1993), pages 359–393.

R.A KOWALSKI. *Logic for problem solving.* North-Holland, 1979.

D.B. LEBLANG, R.P. CHASE. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, 1987.

A. LIE, T. DIDRIKSEN, R. CONRADI, E.A. KARLSSON, S.O. HALLSTEINSEN, P. HOLAGER. Change oriented versioning. In C. Ghezzi and J.A. McDermid, editors, *Proc. of 2nd European SE Conference (ESEC'89)*, pages 191–202. Springer-Verlag, 1989.

D.B. LEBLANG. The CM challenge: Configuration management that works. In W.F. Tichy, editor, *Configuration management volume 2 of* Trends in Software, pages 1–37. John Wiley & Sons, 1994.

E. LAMMA, P. MELLO, editors. *Extension of Logic Programming ELP'92.* Springer-Verlag, 1993.

E. LAMMA, P. MELLO, A. NATALI. Reflection mechanisms for combining Prolog databases. *Software - Practice and Experience*, 21(6):603–624, 1991.

F.G McCABE. *Logic and Objects.* Prentice Hall, 1992.

A. MAHLER, A. LAMPEN. An integrated toolset for engineering software configurations. In P. Hederson, editor, *Proc. ACM SIGSOFT'88*, pages 191–200, Boston, 1988. ACM Press.

P. NÁVRAT. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38:37–46, 1996.

P. NÁVRAT, M. BIELIKOVÁ. Knowledge controlled version selection in software configuration management. *Software—Concepts and Tools*, 17:40–48, 1996.

P.J. NICKLIN. Managing multi-variant software configurations. In P.H. Feiler, editor, *Proc. of the 3rd Int. Workshop on Software Configuration Management*, pages 53–57. ACM SIGSOFT, 1991.

S. OHSUGA. How can knowledge-based systems solve large-scale problems?: Model-based decomposition and problem solving. *Knowledge-Based Systems*, 6(1):38 – 62, 1993.

R.A. ROSENMAN, J.S. GERO. The what, the how, and the why in design. *Applied Artificial Intelligence*, 8:199–218, 1994.

L. STERLING, R.D. BEER. Metainterpreters for expert system construction. *Journal of Logic Programming*, pages 163–178, 1989.

D.T SANNELLA, L.A WALLEN. A calculus for the construction of modular Prolog programs. *The Journal of logic programming*, (12):147–177, 1992.

W.F. TICHY. Tools for software configuration management. In *Proc. Int. Workshop on Software Version and Configuration Control*, pages 1–20, Stuttgart, 1988.

D. XU, G. ZHENG. Logical objects with constraints. *ACM SIGPLAN Notices*, 30(1):5–10, 1995.

C. ZANIOLO. Object-oriented programming in Prolog. In *Proc. Int. Symposium on Logic Programming*, Atlantic City, 1984.

A. ZELLER, G. SNELTING. Handling versions sets through feature logic. In W. Schafer and P. Botella, editors, *Proc. 5th European Softawre Engineering Conference*, pages 191–204. Springer-Verlag, 1995.

## Appendix

### Inference rules used by an abstract machine to process a multilevel logic program

Let $P$ be a multilevel logic program, $G$ be a conjunctive formula, $A, A'$ be atomic formulae, $\tau, \delta, \sigma$ be substitutions, $\epsilon$ be an empty substitution. Composition of two substitutions is denoted by concatenation. $G\tau$ denotes an application of the substitution $\tau$ to $G$. Let $mgu(A, A')$ denote the most general unifier of two atomic formulae $A$ and $A'$. We shall use following abbreviations:

$mod(P) = \{M \mid M \ ismod\ D \in P\}$

$|M| = \{c \mid M \ ismod\ D, c \in D\}$,

i.e. $mod(P)$ denotes set of names of modules of program $P$ and $|M|$ denotes set of clauses defined in module $M$.

A goal $G$ is provable in a multilevel logic program $P$ in a module named $M$ with substitution $\tau$ if there exists a proof of the formula $P \vdash \tau(M, G, [\,])$.

Proof of a formula $G$ in a module $M$ of a multilevel logic program $P$ can be written as a sequence of formulae $P \vdash \tau_i(M_i, G_i, AUX_i)$, where $M_i$ is name of a module in program $P$, $G_i$ is a goal, $AUX_i$ is a term, and $\tau_i$ is a substitution. Initially, we start from an empty auxiliary memory $AUX$, i.e. $AUX_1 = [\,]$. Next formula of a proof is obtained by applying a suitable inference rule. Goal is proved if a formula is inferred with $true$ in place of a goal after a finite number of steps.

The inference rules are written in form

$$\frac{premises}{conclusion}$$

Inference rules:

1. True I

$$\frac{}{P \vdash \epsilon(M, true, AUX)}$$

2. True II

$$\frac{}{P \vdash \epsilon(M, true)}$$

3. Conjunction I

$$\frac{P \vdash \tau(M, A, AUX), P \vdash \delta(M, G\tau, AUX)}{P \vdash \tau\delta(M, (A, G), AUX)}$$

4. Conjunction II

$$\frac{P \vdash \tau(M, A), P \vdash \delta(M, G\tau)}{P \vdash \tau\delta(M, (A, G))}$$

5. Atomic formula I (no reflection)

$$\frac{\begin{array}{c} M \in mod(P), A' :-G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau) \end{array}}{P \vdash \tau\delta(M, A)}$$

6. Atomic formula II (upward reflection)

$$\frac{\begin{array}{c} Meta \in mod(P), M \in mod(P), \\ P \vdash \sigma(M, connect(Meta)), \\ A' :-G \in |Meta|, \\ \tau \in mgu(reflect\_up([M, A, AUX]), A'), \\ P \vdash \delta(Meta\sigma, G\tau, AUX1) \end{array}}{P \vdash \tau\delta\sigma(M, A, AUX)}$$

7. Atomic formula III (connect not defined)

$$\frac{\begin{array}{c} M \in mod(P), \\ \neg(Meta \in mod(P) \wedge P \vdash \sigma(M, connect(Meta))), \\ A' :-G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau, AUX) \end{array}}{P \vdash \tau\delta(M, A, AUX)}$$

8. Atomic formula IV (downward reflection)

$$\frac{\begin{array}{c} Meta \in mod(P), \\ \neg(Meta1 \in mod(P) \wedge P \vdash \sigma(Meta, connect(Meta1))), \\ M \in mod(P), A' :-G \in |M|, \\ \tau = mgu(A, A'), \\ P \vdash \delta(M, G\tau, AUX) \end{array}}{P \vdash \tau\delta(Meta, reflect\_down([M, A, AUX]), AUX1)}$$

The rules 2, 4, 5 define procedural semantics of a modular logic program. These rules are necessary in order to determine which module is a meta-module with respect to a given program module (the relation *connect*).