An Environment for Combining Prolog Programs

María Vargas-Vera[1], Dave Robertson and Robert Inder

Department of Artificial Intelligence
University of Edinburgh,
Edinburgh, Scotland, EH1 1HN.
email: mariav@aisb.ed.ac.uk

Abstract

The purpose of this paper is to describe an environment for the construction of complex Prolog programs by combining simpler Prolog programs. The technique consists of the development in parallel of pieces of software which are combined to produce the final program. For producing elegant and efficient programs we need to keep information such as: the kind of program according to a classification effected by considering features of the program and information about history development such as the initial skeleton and techniques that the user applied in the construction of the program. The problem of combining two programs for producing a new program while retaining correctness properties is not an easy problem. Therefore we design a system which provides assistance in deciding how to combine the two programs. The system is based on program transformation, a technique used for optimisation of programs [BD77], supplemented with knowledge of program development.

1 Introduction

Several techniques have been suggested as aids for producing software that can be easily updated, for instance stepwise refinement proposed by Wirth [Wir71] for the construction of programs in procedural languages. This technique involves the development of the system by means of a sequence of refinements of the specification for which each stage is slightly more refined (ie. concrete). At each stage one of several instructions of the given program is decomposed into more detailed instructions. This successive decomposition or refinement of the specification terminates when all the instructions are expressed in terms of the target language. The technique described

[1]Supported by the National University of Mexico

above is also known as iterative enhancement. Another variant of stepwise refinement is the creation of parallel enhancements which need to be combined to produce the final problem. This combined program incorporates the various changes performed on the skeleton program (provided by the techniques editor) in a consistent way. This construction of complex programs by dividing the problem into smaller subproblems until these small subproblems can be seen to form extensions of the same skeleton, is the idea of parallel enhancement. The latter suggest the idea of developing a program by finding the suitable basic control flow (called the 'skeleton'); once the skeleton has been determined, extra computations are included by applying appropriate *techniques* to produce an extension. Separate extensions can be combined to produce the desired program. The extensions can be regarded as another skeleton permitting us to repeat the process recursively until the final program has been developed. In this context, an extension is a program derived from a skeleton by applying consistently the following modifications: renaming the predicate, adding arguments, adding subgoals, reordering arguments and adding clauses which do not change the control flow.

The motivation for our research arose from the idea of having an automatic tool that helps in the construction of programs by assisting programmers in the task of integrating several pieces of software while retaining the correctness properties of the original programs. This tool will be part of an environment for the construction of Prolog programs which will consist of a *problem description module*, a *techniques editor* and a *composition system*. The *problem description module* will allow users to describe key aspects of the type of problem that the programmer wants to solve. The output from this module will be a plan for the construction of the program, with the restriction that the system will generate plans for a defined domain. The *techniques editor* for the construction of programs is based on the idea of skeletons and techniques proposed by Kirschenbaum et. al. [KLS89]. This techniques editor uses a library of skeletons and techniques from which the user can choose a suitable skeleton and set of techniques for developing the plan (which in a future stage will be proposed by the description and planning module) for the construction of his/her application. The *composition system* combines programs which have the same control flow or different control flows for a restricted classes of programs.

People in several areas of computer science have been interested in the problem of merging two or more programs as a major issue in research. In the program transformation field Burstall and Darlington built a system for transforming programs where the programs were expressed as recursion equations [BD77] and Tamaki and Sato developed unfold/fold transformations for logic programs proving that the unfold/fold transformations preserve equivalence of programs [TS83]. There are also solutions to the composition problem in procedural languages as described by Horwitz for merging different enhancements of the same skeleton [HPR88] and Berzis who gives a partial semantic rule for merging extensions for applicative programs [Ber86]. Other researchers who also worked in program composition were Sterling, Beer and Lakhotia. Their main interest was to compose meta-interpreters. Sterling and Lakhotia developed methods for combining programs such as procedural join, join 1-1 and meta-join [LS87].

The problem of combining programs together efficiently is extremely difficult in the general case. Problems arise when we try to combine programs with the same control flow and these become worse when we try to combine programs with different control flows. The partial solution which we have adopted for this is to classify Prolog programs into classes of programs and to propose a method for each class of program. Each method works properly for each particular class of programs. Special classes of programs, for instance, meta-interpreters, mutants and trav-strav (see section 2), within the composition system require extra information. This information is a history of program development which is constructed by a techniques editor. The kind of information that the combination system will require is a set of relations about how the subgoal in the clause of the initial skeleton was transformed in the program, plus information about extra clauses which do not appear in the initial skeleton. Our claim is that using knowledge about the program development, called the *history of the program*, we can produce, for some methods, efficient combined programs whilst in other methods without this information it is not possible to combine programs ensuring correctness. For instance by using the history of the program the meta-interpreters method produces efficient programs whilst in the mutant and the trav-strav method there is no way of combining programs belonging to these classes while preserving correctness properties.

Each method implemented in the composition system described in the next section can be seen as a sequence of transformation operations. Thus the composition problem can be seen as a novel application of program transformation as opposed to its orthodox use in program synthesis. These methods are given to the user in an interactive system to help in the construction of a program that solves his/her application. Some of these methods are completely automatic while others are semi-automatic.

This paper is structured as follows: section 2 gives the classification of pairs of programs which can be combined using our system, and describes how a composition method is chosen by examining features of the two programs. Section 3 introduces a set of methods for combining programs given restrictions on the application of each method. Section 4 draws our initial conclusions.

2 Classification of pairs of programs

There is not a general solution to the the problem of combining programs. For this reason we divide the set of programs that we can write in Prolog into classes of programs according to stipulated structural features and we find methods which can be used for combining programs in each class. Our classification covers a large set of useful Prolog programs and we also show classes of programs that cannot be combined by any of our methods. The classification uses the following skeletons: traverse, short_traverse, search and solve described in detail in [KLS89]. We classify the pair of programs that will be combined into two types: programs with the same control flow and programs with a different control flow. The programs with same control flow are programs which are enhancements from the same skeleton. The programs belonging to the second type (programs with a different control) can start by using the same skeleton but at some stage of the development extra clauses were added to the program. Alternatively they could be programs built by using different skeletons; for instance, a program constructed using the traverse skeleton and the second program using the short_traverse skeleton. The classification of programs is shown in figure 1.

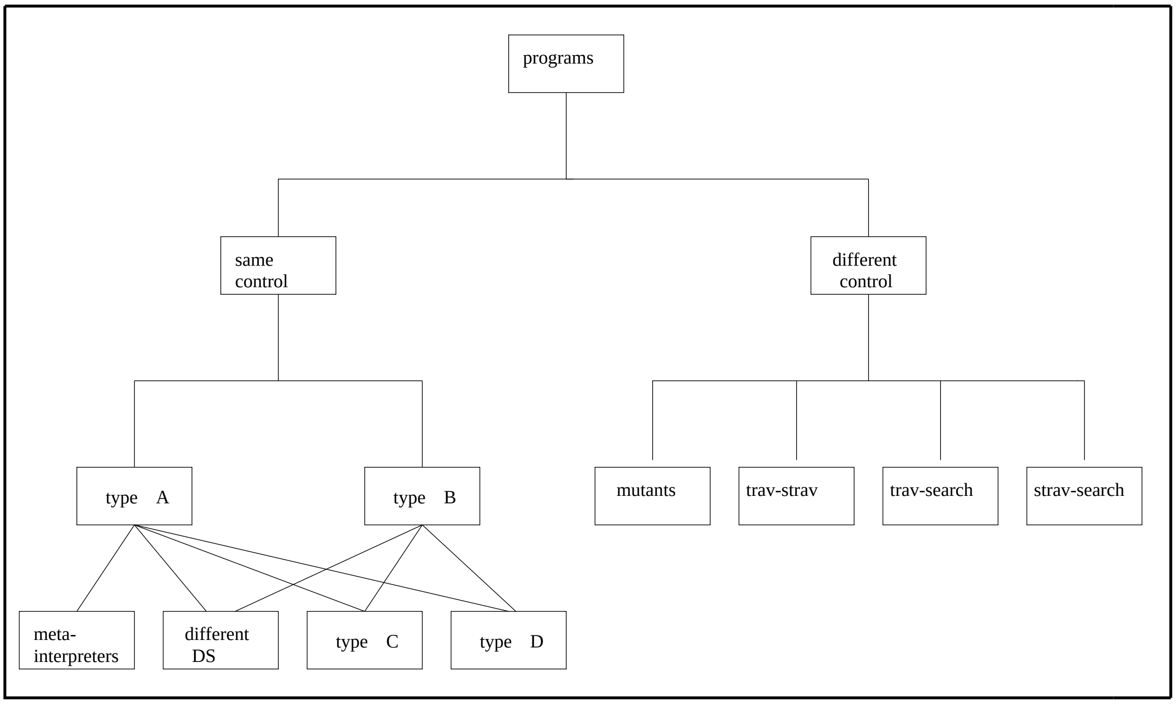

Figure 1: Classification of Prolog programs

Pairs of type A are programs which come from the same skeleton with the same number of clauses, which have one condition on the recursive cases and which traverse just one data structure (for instance lists, trees).

Pairs of type B are programs which come from the same skeleton with a different number of clauses, which have more than one condition on the recursive cases and which traverse data structures of a given type.

Pairs of type C are programs which are extensions of type A or B working in an arithmetic domain.

Pairs of type D are programs which are pairs of type A or B, where the input for one program P_2 is the output of the other member of the pair P_1.

The meta-interpreters programs are those of type A built using the skeleton solve/1 defined below.

```
solve(true).
solve((A,B)) :- solve(A), solve(B).
solve(A) :- sys(A), call(A).
solve(A) :- clause(A,B), solve(B).
```

Pairs of type A or B which traverse different data structures are called different DS.

Mutant pairs of programs are pairs $< AB >$ (derived from pairs $< CD >$) in which the control flow is slightly modified with respect to the control flow in pair $< CD >$.

The *trav-strav* pairs of programs are those where one is constructed by using the traverse skeleton and the other is built by using the short_traverse skeleton.

The *trav-search* pairs of programs are those where one is constructed by using the traverse skeleton and the other is built by using the search skeleton.

The *strav-search* pairs of programs are those where one is constructed by using the short_traverse skeleton and the other is built by using the search skeleton.

The composition system provides guidance during program composition by asking questions about the features of each program to be combined and by using information coming from the techniques editor. This information is used in the selection of the combination method. Our classification procedure is described using rules similar to the examples given below. In these rules the characteristics of the programs are the preconditions and the conclusion is the method which should be used for combining programs.

```
RULE 1
```

```
if
both programs are of type A
then
the method is: join 1-1.
RULE 2
if
both programs are of type B
then
the method is: procedural join.
RULE 3
if
both programs are of type DS
then
the method is: DS.
RULE 4
if
both programs are of type meta-interpreters
then
the method is: meta-interpreters.
```

3 Combination techniques

Given the above classification of each pair of programs we then can select the most appropriate combination strategy from a library of methods. For the sake of brevity we only present some of the combination methods in the current paper. A full set is given in [VV94].

The join 1-1 method is defined as a sequence of unfolding and folding operations which preserve the equivalence of programs as was proved in [TS83]. This method works only with programs that are extensions of the same skeleton [LS87]. Each stage in the combination process is done by combining the ith-clause from program A and the ith-clause from program B. Note that it is the fact that they are derived from the same skeleton which makes this possible. This information is given by the techniques editor which was used in the construction of the programs. The restrictions for join 1-1 are that it only combines programs that have the same number of clauses and also that the two programs must be generated using the same skeleton. The advantage of imposing this restriction is that the programs generated by 1-1 join are free of redundant clauses, because this case only allows joined clauses which operate on the same instance of the induction parameter.

The starting point is the join specification which gives information about how the two programs are to be joined. The meaning is conveyed by sharing variables between the arguments of the resulting program and the two initial programs. Therefore a *join specification* is a formal expression in which we define the characteristics of the new program which can be generated using two other programs. The join specification is defined as follows:

Let T be defined as $\mathcal{T} :- \mathcal{P}, \mathcal{Q}$ where $\mathcal{P}$ and $\mathcal{Q}$ are predicates to be joined; they are called join operands and $\mathcal{T}$ is the join target. This join specification is called (in the terminology of program transformation) the definition clause.

The unfolding operation is applied to the join specification, the first subgoal $\mathcal{P}$ is selected, and then a new clause T_i is constructed by selecting a clause

$P_i \ :- B_1, \ldots, B_n \ \ (n \geq 0)$

whose head unifies with $\mathcal{P}$ by using mgu Θ_P. Therefore $T_i \ \ :- (B_1, \ldots, B_n, \ \mathcal{Q} \ )\Theta_P$. Following this process the second subgoal in the join specification can be unfolded, obtaining $T_i \ :- (B_1, \ldots, B_n, C_1, \ldots, C_m)\Theta_P \Theta_Q$.

The folding operation can be defined as the inverse operation of the unfolding operation except that the folding process is done by just taking into consideration γ, which is the set of join specifications for the program.

$T_i :- (B_1, \ldots, B_n, C_1, \ldots, C_m)\Theta_P \Theta_Q$

If there exists some pair of subgoals B_i and C_i in the body of the clause T_i such that (B_i, C_i) and (P, Q) unifies with some mgu Θ

Then replace the two subgoals B_i and C_i with T_i.

For example let us consider the predicate *sum/2* which computes the sum of the elements of a list and the predicate *count/2* which computes the number of elements of a list. The notation which will be used in the following example is: $P_{i,body}$ is the body of the ith-clause in program P and $Q_{i,body}$ is the body of the ith-clause in program Q.

```
P1:  sum([],0).
P2:  sum([H|T],Sum) :- sum(T,Sum1),
          Sum is Sum1 + H.
Q1:  count([],0).
Q2:  count([H|T],Count) :- count(T,C1),
           Count is C1 + 1.
```

The application of join 1-1 on the two procedures *sum/2* and *count/2* with respect to the join specification shown below will be described as follows:

sum_count(List,Sum,Count) :- sum(List,Sum), count(List,Count).

The process consists of taking clause 1 from program *sum/2* and clause 1 from program *count/2* and creating an instance of the join specification (ie. $T_1 \ :- \ P_1, Q_1$). After unfolding P and Q with respect to P_1 and Q_1, P_1 unifies with P by using the mgu $\theta_P = \{[]/List, 0/Sum\}$ and Q_1 unifies with Q by using mgu $\theta_Q = \{[]/List, 0/Count\}$. Then by replacing $P_{1,body}$ and $Q_{1,body}$ in T_1, we obtain the clause shown below.

sum_count([],0,0).

Folding the previous clause we obtain the first combined clause.

Secondly clause 2 is taken from program *sum/2* and clause 2 from program *count/2* and an instance of the join specification (ie. $T_2 \ :- \ P_2, Q_2$) is created. Then P and Q in T_2 is unfolded with respect to P_2 and Q_2, after that we obtain the clause which appears below, where P_2 unifies with P by using the mgu $\theta_P = \{[H|T]/list, Sum/Sum\}$ and Q_2 unifies with Q by using mgu $\theta_Q = \{[H|T]/List, Count/Count\}$ therefore replacing $P_{2,body}$ and $Q_{2,body}$ in T_2 we obtain the following clause:

```
sum_count([H|T],Sum,Count) :-
        sum(T,Sum1), Sum is Sum1 + H,
        count(T,C1), Count is C1 + 1.
```

The combined program is shown below.

```
sum_count([],0,0).
sum_count([H|T],Sum,Count) :-
       sum_count(T,Sum1,C1),
       Sum is Sum1 + H,
       Count is C1 + 1.
```

The meta-interpreters method is based on a sequence of unfolding and meta-folding operations where the meta-folding is a variant of the folding operation defined above. The *meta-folding* operation verifies before folding if the subgoals (from program A and from program B) that are candidates for joining are enhancements of the same subgoal in the skeleton. If that is the case the subgoals are folded, otherwise any subgoal is folded.

The main problem in the combination of programs belonging to the meta-interpreters class is that the information that is conveyed in the join specification is insufficient for folding subgoals because after the unfolding operation is applied we have local variables which do not appear in the head of the join specification; thus we need another mechanism that allows us to apply the folding operation. We have called this extra information a *program history*. This history of the program provides knowledge about how the initial skeleton is transformed in the development of the program and by using this information it is possible to fold the subgoals that are enhancement of the same subgoal in the initial skeleton. This history of the program can be supplied by a techniques editor for Prolog. The current version of our editor for the construction of programs [Rob91] is based on the idea of skeletons and techniques defined by Kirschenbaum et. al. [KLS89]. The idea is that the user chooses a control flow or skeleton for solving his/her application, then a program is developed by adding computations around its skeleton without changing the control flow it gives. At a future stage the editor will hold a history of the development of the program. This history of the program is the knowledge that will be used in the composition process. This knowledge is provided as relations, which can be defined as follows:

$\mathcal{R}\ (\alpha, \beta, \gamma)$

where α is subgoal in the skeleton, β is the transformed subgoal in the skeleton and γ is the program name.

The problem in the combination of programs belonging to the class of *meta-interpreters* is that after applying the unfolding operation $\mathcal{P}$ and $\mathcal{Q}$ with respect to a join specification we can get clauses that are not an instance of the join specification, as in the following example in which we are combining the meta-interpreter *count/2* and the meta-interpreter *proof/2*. The meta-interpreter *count/2* executes a Prolog goal and computes the number of goal reductions required to prove it and the meta-interpreter *proof/2* returns the proof tree associated with the proof of a goal. These two meta-interpreters are shown below.

```
P1:  count(true,0).
P2:  count((A,B),C) :-
           count(A,D),
           count(B,E),
           C is D+E.
P3:  count(A,0) :-
           sys(A),
           call(A).
P4:  count(A,C):-
           clause(A,B),
           count(B,D),
           C is D + 1.
```

```
Q1:  proof(true,true).
Q2:  proof((A,B),(PA,PB)):-
          proof(A,PA),
          proof(B,PB).
Q3:  proof(A,sys) :-
          sys(A),
          call(A).
Q4:  proof(A,(A :- PB)) :-
          clause(A,B),
          proof(B,PB).
```

The two meta-interpreters *proof/2* and *count/2* defined above will be combined using the join specification shown below.

count_proof(Goal,Count,Proof) :- count(Goal,Count), proof(Goal,Proof).

The stages of the composition process can be described as follows: Firstly taking the first clause of *count/2* and the first clause of *proof/2* and creating an instance of the join specification (ie. $T_1 :- P_1, Q_1$). After unfolding P and Q with respect to P_1 and Q_1, we have the case that P and $P_{1,head}$ unifies with the substitution $\theta_P = \{true/Goal, 0/Count\}$ and Q and $Q_{1,head}$ unifies with mgu $\theta_Q = \{true/Goal, true/Proof\}$ and we obtain the clause shown below.

count_proof(true,0,true) :- count(true,0), proof(true,true).

Then by replacing $P_{1,body}$ and $Q_{1,body}$ in T_1 we obtain the following clause:

count_proof(true,0,true).

Folding the previous clause we obtain the first combined clause.

Secondly using the second clause of program *count/2* and the second clause of program $proof/2$ we construct an instance of the join specification in this case $T_2 \ :- \ P_2, Q_2$. After unfolding P and Q with respect to P_2 and Q_2 we find that P and $P_{2,head}$ unifies with the substitution $\theta_P = \{(A,B)/Goal, Count/Count\}$ and Q and $Q_{2,head}$ unifies with mgu $\theta_Q = \{(A,B)/Goal, (PA,PB)/Proof\}$ and we obtain the clause shown below.

```
count_proof((A,B),Count,(PA,PB)) :-
    count(A,D), count(B,E), Count is D + E,
    proof(A,PA), proof(B,PB).
```

Applying the folding operation to $count(A,D)$ with $proof(A,PA)$ we have that these two subgoals can be substituted by $count\_proof(A,D,PA)$ and in the same way $count(B,E)$ and $proof(B,PB)$ can replaced by $count\_proof(B,E,PB)$. Therefore the resulting combined clause is shown as follows:

```
count_proof((A,B),Count,(PA,PB)) :-
    count_proof(A,D,PA), count_proof(B,E,PB),
    Count is D + E.
```

Thirdly we take the third clause of each program and construct an instance of the join specification (denoted $T_3 \ :- \ P_3, Q_3$). After unfolding P and Q with respect to P_3 and Q_3 we have the case that P and $P_{3,head}$ unifies with the substitution $\theta_P = \{A/Goal, 0/Count\}$ and Q and $Q_{3,head}$ unifies with mgu
$\theta_Q = \{A/Goal, \ (A \ :- \ PA)/Proof\}$. This gives the following clause:

```
count_proof(A,0,(A :- PA)) :-
    sys(A), call(A),
    sys(A), call(A).
```

After applying a merge operation (which removes all the goals which are syntactically identical) and doing a folding operation we obtain the third combined clause.

```
count_proof(A,0,(A :- PA)) :-
    sys(A), call(A).
```

Fourthly we take the fourth clause of each program obtaining an instance of the join specification (T_4 :– P_4, Q_4). After unfolding P and Q with respect to P_4 and Q_4 we have the case that P and $P_{4,head}$ unifies with the substitution
$\theta_P = \{A/Goal, Count/Count\}$ and Q and $Q_{4,head}$ unifies with mgu
$\theta_Q = \{A/Goal, (A \text{ :– } PB)/Proof\}$ and we produce the following clause:

```
count_proof(A,Count,(A :- PB)) :-
    clause(A,C), count(C,D), Count is D + 1,
    clause(A,B), proof(B,PB).
```

In this unfolded clause it is not possible to apply folding to the subgoals *count(C,D)* and *proof(B,PB)* because they are not an instance of the join specification. The variable $Goal$ unifies with C in the subgoal $count(C, D)$ and $Goal$ unifies with B in the subgoal $proof(B, PB)$. Therefore we need to use the program history to inform us about which subgoals to combine during folding. This *program history* is knowledge about the program and is kept in relations as defined below. These relations provide information about the skeleton used in the construction of the program and how the subgoal $solve(Goal)$ in the initial skeleton was transformed in the program.

```
R(solve(Goal),count(Goal,D),count).
R(solve(Goal),proof(Goal,P),proof).
```

Using this information the subgoals $count(C, D)$ and $proof(B, PB)$ are folded because the relations define that both subgoals are enhancements of the subgoal $solve(Goal)$. Then we obtain the clause which is presented as follows:

```
count_proof(A,D,(A :- PB)) :-
    clause(A,B), clause(A,C),
    count_proof(B,D,PB), Count is D + 1.
```

In the previous clause the second subgoal can be deleted using the *merge* operation producing the final fourth combined clause.

The complete program *count_proof* is shown as follows:

```
T1:    count_proof(true,0,true).
T2:    count_proof((A,B),Count,(PA,PB)) :-
           count_proof(A,D,PA),
           count_proof(B,E,PB),
           Count is D + E.
T3:    count_proof(A,0,(A :- PA)) :-
          sys(A),
          call(A).
T4:    count_proof(A,Count,(A :- PB)) :-
           clause(A,B),
           count_proof(B,D,PB),
           Count is D + 1.
```

The 'particular' method is a method that combines programs defined in the same algebraic domain. The user specification is defined as: $\mathcal{T} :- \mathcal{P}, \mathcal{Q},\ F_1 \ldots F_n$ where $\mathcal{P}$ is identical to $\mathcal{Q}$ and F_1 to F_n are subgoals that perform computations in the same algebraic domain of $\mathcal{P}$ and $\mathcal{Q}$. This particular method works by performing the following sequence of transformation operations: unfolding; application of laws such as commutativity or associativity which are valid in each algebraic domain; and finally folding.

For example consider the problem of the construction of the program *len_two* which computes the length of two lists and returns as result the total length using the definition of naive length/2 and by using the following join specification:

```
len_two(L1,L2,Len) :- len(L1,Len1), len(L2,Len2), Len is Len1+Len2
```

In the example $\mathcal{P}$ and $\mathcal{Q}$ are identical and the program that computes the length of two lists at the same time is defined as follows:

```
len_two([],[],0).
len_two([],[H2|T2],Len) :-
     len(T2,Lenx),
     Len is Lenx+1.
len_two([H1|T1],[],Len) :-
      len(T1,Lenx),
      Len is Lenx+1.
len_two([H1|T1],[H2|T2],Len) :-
      len_two(T1,T2,Lenax),
      Len is Lenax+2.
```

The 'general' method is an automatic way of building programs that has a user specification formed from more than two subgoals without any restriction in the form of the subgoals. We make a distinction in this case between the join specification and the user specification. The *join specification* works over two subgoals as in the previous methods (such as join 1-1 and meta-interpreters method) but the *user specification* can be formed for m subgoals where m is an integer greater than 2. The *general* method works by selecting two subgoals from the *user specification* for which it is possible to make a composition. The second stage is to combine these two subgoals using a preloaded join specification or with a join specification that the user can define interactively into the system.

The algorithm can be described in terms of a user specification, join specification and procedural join. The algorithm shown below is an extension of procedural join but can be easily extended for the rest of the methods. The justification for the implementation of this method arises from the fact that sometimes the user needs to combine two programs as a first stage in the software development process and as a second stage just needs to add extra computations using the values computed in the first stage. This method gives this facility automatically.

1. Definition of a *user specification* defined as:
   $T \ :- \gamma_1, \gamma_2, \cdots, \gamma_n$
   where each γ_i is a subgoal
2. Choose two subgoals in the join specification. In this case γ_1 and γ_2.
3. Select a *join specification.* In this stage if the system has a join specification for combining γ_1 and γ_2 it will offer this to the user, otherwise the system provides an interactive interface for defining the new join specification.
4. Apply the procedural join method.

The same example *len_two* was also generated by using the 'general' method obtaining the code shown below but this is less efficient than the program obtained by using the 'particular method'.

```
len_len([],[],0,0).
len_len([],[H2|T2],0,Len2) :-
        len(T2,Lenx2),
        Len2 is Lenx2 + 1.
len_len([H1|T1],[],Len1,0) :-
        len(T1,Lenx1),
        Len1 is Lenx1 + 1.
len_len([H1|T1],[H2|T2],Len1,Len2) :-
        len(T1,T2,Lenx1,Lenx2),
        Len1 is Lenx1 + 1,
        Len2 is Lenx2 + 1.
len_two(List1,List2,Len) :-
        len_len(List1,List2,Len1,Len2),
        Len is Len1 + Len2.
```

The 'mutant' method is an extension of the meta-interpreters method described above. In the algorithm designed for meta-interpreters the composition process is guided by the subgoals which appear in the skeleton in each clause. We need to handle the problem of extra clauses that do not have a corresponding parent clause. The solution for these mutant clauses is described in the algorithm presented below. The algorithm creates an instance of the join specification T_k and the values of variables are passed through the instance T_k by unfolding the predicate $\mathcal{P}$ or predicate $\mathcal{Q}$. The selection of which predicate is unfolded is determined as follows: if the mutant clause belongs to program $\mathcal{P}$ then $\mathcal{P}$ is unfolded, otherwise predicate $\mathcal{Q}$ is unfolded. This process is repeated while there are mutant clauses in each program. The operation of unfolding is applied partially through the join specification, and the final mutant combined clause is redefined interactively by the user.

1. Create a instance T_i of the join specification $\mathcal{T} \ :- \ \mathcal{P}, \mathcal{Q}$
2. Unfold $\mathcal{P}$ in T_i with respect to P_i.

   - if P unifies with the head of clause P_i by using mgu Θ_P and P_i is defined: $P_i \ :- \ A_1, \ldots, A_n$

   Then in T_i replace P with the body of P_i with the mgu Θ_P .

   This process can be described in the same way for $\mathcal{Q}$ in the case that $\mathcal{Q}$ is the mutant clause. Note that depending on which is the mutant clause we have two cases: if the mutant clause belongs to program P the values of the variables in the head of the clause T_i are passed to Q. In the second case if the mutant clause belongs to program Q then the values of the variables in the head of the clause T_i are passed to P.

   Then

- T_i :– $(A_1, \ldots, An)$, $\mathcal{Q}$. Θ_P with $n > 0$ or
- T_i :– $\mathcal{P}$, $(C_1, \ldots, Cm)\Theta_Q$ with $m > 0$

3. Apply the fold operation using the set of join specifications (called γ).
4. Ask the user for redefinition of these clauses
5. Repeat the same process for each mutant clause that appears in each program.

The example for this method is constructed by using the example shown in the meta-interpreters method. A mutant program is obtained adding an extra clause to the program *count_proof/3.*

Let count_proof(A,D,overflow) :- $D > 0$ be the mutant clause.

The interpreter *how/3* (which explains how the goal was proved) will be constructed using the two meta-interpreters *count_proof/3* and *interpret/1* defined above and by using the following join specification:

how(Goal,Proof,Depth) :- count_proof(Goal,Depth,Proof), interpret(Proof).

```
Q1: interpret(true).
Q2: interpret((Proof1,Proof2)) :-
        interpret(Proof1), interpret(Proof2).
Q3: interpret(Proof) :-
        fact(Proof,Fact),
        nl, write([Fact,'is a fact in the database.']).
Q4: interpret(Proof) :-
        rule(Proof,Head,Body,Proof1),
        nl, write([Head,'is proved using the rule']),
        display_rule(rule(Head,Body)),
        interpret(Proof1).
```

The process consists of taking clause 1 from program *count_proof/3* and clause 1 from program *interpret/1* and creating an instance of the join specification (denoted $T_1 :- P_1, Q_1$). After unfolding P and Q with respect to P_1 and Q_1, P_1 unifies with P by using the mgu $\theta_P = \{true/Goal, true/Proof, 0/Depth\}$ and Q_1 unifies with Q by using mgu $\theta_Q = \{true/Goal\}$. Then by replacing $P_{1,body}$ and $Q_{1,body}$ in T_1, we have the following clause:

how(true,true,D).

Folding the previous clause we obtain the first combined clause.

Secondly we have a mutant clause in program *count_proof/3*. The way that we deal with the second clause of program *count_proof/3* is to create an instance $T_2 \; :- \; P2, Q_2$. After a partial unfolding (i.e. P with respect to P_2), P_2 unifies with the mgu $\theta_P = \{A/Goal, overflow/Proof, D/Depth\}$. Then by replacing $P_{2,body}$ in T_2 we have the following clause:

```
how(A,overflow,D) :- D>0.
```

This clause needs to be redefined by the user because the final functionality depends of the application that the user wants to have on his/her program. For instance, the previous clause can be redefined as follows:

how(A,overflow,D) :- !.

This redefinition of the clause forces the termination of the program after the depth bound is reached.

The following set of assumptions are required by the algorithm which tackles the mutants problem.

1. Program A and program B were created using the same skeleton S.
2. Termination of all new subgoals which we have included by using techniques is assumed.
3. Both programs work over the same type of data structure.

The mutant method requires extra knowledge about which the mutant clauses are. This information is provided by the techniques editor as relations, which can be defined as follows:

$\mathcal{R}\ (\alpha, \beta, \gamma, \delta)$.

where α is the number of the clause in the program, β is the name of the program, γ is the initial subgoal in the skeleton and δ is the transformed subgoal.

4 Conclusions

We have implemented a tool which incorporates a set of methods for combining programs. These methods are a sequence of transformation operations which guarantee correctness. Therefore the combined program retains the same meaning as the two initial programs. Our result is that using knowledge about the program, called *program history* we can produce efficient combined programs and also combine programs which cannot be combined without the use of this knowledge.

Acknowledgements

I would like to give acknowledgements to Ian Lewin, Siani Baker and Susan Bull for their comments on this paper.

References

[BD77] Rod Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal ACM*, 24(1):44–67, 1977.

[Ber86] Valdis Berzins. On Merging Software Extensions. *Acta Informatica*, 23:607–619, 1986.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[KLS89] Marc Kirschenbaum, Arun Lakhotia, and Leon S. Sterling. Skeletons and Techniques for Prolog Programming. Tr 89-170, Case Western Reserve University, 1989.

[LS87] Arun Lakhotia and Leon Sterling. Composing Logic Programs with Clausal Join. Tr 87-25, Computer Engineering and Science Department, Case Western Reserve University, 1987.

[Rob91] Dave 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, April 1991.

[TS83] Hisao Tamaki and Taisuke Sato. A Transformation System for Logic Programs which Preserves Equivalence. Tr 83-18, ICOT Research Center, 1983.

[VV94] Maria Vargas-Vera. *Guidance during Program Composition in a Prolog Techniques Editor*. PhD thesis, Department of Artificial Intelligence, Edinburgh University, 1994. In preparation.

[Wir71] N. Wirth. Program Development by Stepwise Refinement. *Communications of the Association for Computing Machinery*, 14(4):221–227, 1971.