Development of a Prolog Tracer by Stepwise Enhancement

Arun Lakhotia[1], Leon Sterling & Dimitar Bojantchev

Computer Engineering and Science Department
Case Western Reserve University
Cleveland, OH 44106, USA

Abstract

A PROLOG tracer is essentially a PROLOG interpreter extended to provide features, such as *retry, fail, leap, skip, and quasi-skip*, to trace the computational flow of a program. This paper describes how a PROLOG tracer may be built by *stepwise enhancement*. Using this method, first a collection of partial-tracers are developed, each partial-tracer providing only a part of the tracer's functionality. The partial-tracers are then *composed* to yield a single tracer with the composite functionality of the partial-tracers.

Stepwise enhancement provides an alternative to stepwise refinement and iterative enhancement in scenarios where the "natural" subproblems do not correspond to distinct subprograms. This typically happens, as in the case of a PROLOG tracer, when different subprograms require the same program flow due to the contraint that their computations be performed simultaneously.

Stepwise enhancement eases the development and extension of a tracer, and also makes it customizable. A user may customize her tracer by composing only the partial-tracers providing the desired functionality. New features may be added by developing a partial-tracer that provides the desired behavior and composing it with other partial-tracers.

Keywords PROLOG tracer, program composition, interpreters, program enhancement

[1]Current Address: The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

1 Introduction

A PROLOG tracer is in essence an interpreter of PROLOG with provisions to inspect or modify the program's state[2]. With the meta-programming support of PROLOG, a PROLOG tracer may be written in PROLOG itself. Such a tracer may rely on the underlying PROLOG engine for unification, backtracking, and clause selection, and worry about providing the actual trace features only.

Using a tracer as an example, it is our intention to demonstrate the method of developing PROLOG programs by *stepwise enhancement*. The method is useful when developing recursive, non-deterministic programs that perform several logically independent functions *simultaneously* around the same search tree. In such a situation, as elaborated upon below, the methods of modular decomposition [11] may not lead to a correct solution, and iterative enhancement [1] and stepwise refinement [20] may lead to programs that may, over time, lose their structure and clarity.

A typical PROLOG tracer has a long repertoire of commands to trace the computational flow of the program. One may consider developing a PROLOG tracer by decomposing it into several different partial-tracers (*a la* modules), each implementing a specific feature. The top level tracer goal may then be stated as a conjunction of goals of the partial-tracers. This decomposition, however, does not lead to correct tracer behavior because each of the partial-tracers, being extensions of a PROLOG interpreter, are recursive and non-deterministic. Hence, a simple conjunction of the goals of the partial-tracers will not ensure that the resulting answers correspond to isomorphic traversals of the corresponding search trees. Furthermore, a conjunction of several partial-tracer predicates, even if it gave the correct result, will cause the goal being traced to be interpreted as many times; clearly a behavior that is not desired.

A tracer may be developed using iterative enhancement, wherein starting with an initial tracer that provides just a few tracing features, the complete tracer is developed by adding one (or some) features in successive iterations. Iterative enhancements may be used either when all the features required of a tracer are not known ahead of time or when the programmer(s) find the task of developing the complete tracer equipped with all the tracer features very hard.

Alternatively, assuming all the tracing features required are known, a programmer experienced with meta-programming in Prolog and well versed with the subtlities of implementing each tracer feature, may develop a tracer using stepwise refinement. This may entail decomposing the tracer into a set of clauses, each responsible for tracing a particular type of goal - conjunction, system, or atomic. Each clause may then be refined to do the task necessary to perform each of the tracer functions.

Even if a PROLOG tracer is developed using stepwise refinement, sooner or later it will get modified to add new feature; because the designers or the users may think of new features

[2]Such a software tool is often referred to as a *debugger*. We prefer to call it a *tracer* because the tool itself only traces a program's execution. The phrase 'debugger' connotes the entity that actually finds and removes a bug, which happens to be the programmer and not the tool.

that may be helpful in debugging. Thus, developing a PROLOG tracer, as also other software, implicitly follows iterative enhancement.

Changing a PROLOG tracer gets increasingly cumbersome as new features are added because the state information required for each command of a tracer is often carried as additional arguments to a base PROLOG interpreter. As the list of commands grows, so does the list of arguments to the interpreter. Each feature may require adding some new arguments and/or restructuring of the program. This continuous tampering with the code leads one to the realm of Lehman's second law of program evolution [8] which states that:

> "As a program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."

If one is not cautious, the program very soon loses its structure and becomes incoherent. Caution is also required to maintain the status quo of the already implemented features. This need for extra care, compounded with the extensive editing required to add arguments, makes every iteration for adding a feature increasingly laborious.

Using stepwise enhancement, we decompose the problem of building a tracer into that of building several independent partial-tracers each providing a separate tracer feature. It differs from modular decomposition in that the partial-tracers are *mechanically* composed into a tracer that has their composite functionality.

The method has several advantages that stem from its support to build logically independent functions separately rather than in the same program. It makes the initial development of a tracer and also its extension to add new features easy. New features may be added by developing them separately and integrating them mechanically with the existing tracer. Building various features separately also opens up the possibility of customizable tracers where only the features that are required may be composed.

This document is not intended as a tutorial on PROLOG tracers. The reader is referred to the DEC-10 PROLOG manual [2] or manual of any other PROLOG implementation. The paper may, however, be used as a guide for developing a tracer.

Section 2 presents a decomposition of the tracer and reasons why its logical pieces do not add up to a whole. A sketch of the method of stepwise enhancement is presented in Section 3. A single-stepper is developed in Section 4 by stepwise enhancement. The stepper provides the skeleton to develop the various tracer features. The various partial-tracers obtained by enhancing the stepper are given in Section 5. The tracer resulting by composing these partial-tracers is presented in the Appendix. Our concluding remarks are in Section 6.

2 Decomposing the tracer

A PROLOG tracer provides the following features:

- *leap* and *quasi-skip*
- *retry* and *fail*
- *skip* and *true*
- show ancestor

In the above list, features requiring similar processing have been placed together. For example, both the leap and quasi-skip commands suppress any further interaction until a "spy point" is reached. Similarly, the retry and fail commands affect only goals that have already been invoked, whereas the skip and true commands affect only the current goal (current at the time the command is requested).

In the above list the commands listed in different groups are independent of each other. That is, computations performed for one affect the computations performed by another only to the extent that the commands trace the execution of the program. The logic for *retry*ing a goal is independent from that of *skip*ping a goal or performing a *leap* up to a spy point.

Given the similarity between features in the same group, and differences between features in different groups, the task of developing a tracer may be decomposed into the task of developing a collection of partial-tracers, each providing features in only one group. Each group of features may be implemented as a tracer (an extension of a PROLOG interpreter) because the features are only meaningful in the context of the execution of a PROLOG program.

A PROLOG tracer consists of one program that traces the execution of a PROLOG goal and simultaneously provides the features of all the partial-tracers. One may consider realizing the complete tracer by a conjunction of the top level goals of the partial-tracers. Doing so will lead to incorrect behavior because the tracer features will not be simultaneously available since each partial-tracer will trace a program independently. This may clearly not be desired.

Common to each partial-tracer is the problem of interpreting a program, stopping the execution at a "port" of certain goals, displaying status information, and querying the user for a command. Being common to all the partial-tracers, this problem may be solved independently, say by developing an interpreter that exhaustively traces through every port of each goal. We call such an interpreter a *single-stepper*.

We contend that in order to develop a PROLOG tracer using a single-stepper, one has to *modify* the single-stepper so as to introduce the other tracer features. The reason is that the only ways new features may be added to a PROLOG program are (a) by changing the existing predicates and (b) by adding new predicates and combining them (using conjunction or disjunction) with the existing program. A single-stepper cannot be extended to provide new features by simply adding new predicates for the same reason that a tracer cannot be created by a conjunction of partial-tracers goals.

3 Stepwise enhancement

From the above discussion, we infer that although a PROLOG tracer may be decomposed into several partial-tracers, each implementing only a subset of features, the resulting partial-tracers cannot be combined (using conjunction or disjunction) to create a complete tracer. This is because the logic for the various partial-tracers is intricately woven around the common problem of interpreting PROLOG programs, stopping execution of a goal at its various ports, and interacting with the user to accept commands.

The PROLOG tracer could alternately be decomposed into partial-tracers, each of which

implements all the features but evaluates only a subset of the PROLOG language. Assuming certain consistency in names of predicates and order of arguments across these partial-tracers, one may simply collect all the resulting clauses (i.e., combine them using disjunction) and create the complete tracer. Such an approach is analogous to stepwise refinement [20].

Since individual partial-tracers resulting from stepwise refinement only interpret a subset of PROLOG, to trace a PROLOG program one would need all the other partial-tracers. Thus, these partial-tracers cannot be developed (and tested) independently. We contend, therefore, that stepwise refinement is not effective in reducing the difficulty of the problem of developing a PROLOG tracer.

The method of stepwise enhancement [7] has been designed to develop programs for problems whose partial solutions based on decomposition along feature boundaries cannot be combined (using programming language primitives, such as sequencing and procedure calls in procedural language and conjunction and disjunction in logic languages) to solve the complete problem. In other words, stepwise enhancement is applicable in situations when logical decomposition of the problem does not lead to a physical decomposition of its code. In such situations, as is the case with developing a PROLOG tracer, this method advocates the development of partial programs by *enhancing* a common subprogram called a *skeleton* program and *composing* the partial programs to generate the final program.

Enhancements are modifications done to a program to add new functionality without losing its old behavior or structure. *Composition* is a ternary operation defined over logic programs that have 'similar structure' [6]. Assuming that two of its inputs are enhancements of the third, composition creates a new program in which 'similar' components of the enhanced programs are retained as is and the different components are merged.

Developing a program by stepwise enhancement takes four steps. In the first step the problem is decomposed into sets of related "features." An assessment is now done as to whether programs implementing these features can be combined to create the final program. If the final program can be so generated then one need not apply stepwise enhancement. Otherwise, in the next step a problem common to all the features is identified and a program to solve this problem developed. This program, termed a *skeleton* program, may be developed using any method, including stepwise enhancement. The third step involves making several independent enhancements to the skeleton program, each introducing a subset of the features. In the last step all the enhancements are composed to yield a single program with the composite behavior of all the enhancements.

The partial-tracers, stated in Section 2, can all be enhanced from a single-stepper, an interpreter with only the provision to interactively step through every port of every goal in the execution of a program. Therefore, a single-stepper is a good candidate for being the skeleton program. The steps in developing a PROLOG tracer by stepwise enhancement may be summarized as:

- Develop a single-stepper.
- *Enhance* the single-stepper into various partial-tracers
- *Compose* these partial-tracers to create the tracer.

4 Developing a single-stepper

In the previous section we observed that a single-stepper is a good starting point for developing a PROLOG tracer. A single-stepper, like a tracer, stops before entering each port of each goal and displays the status of the goal at that port. On receiving a *creep* command it continues execution until it enters another port. For ease of extensibility into a complete tracer, the status information displayed by a single-stepper ought to be the same as that displayed by a tracer. Thus, a single-stepper should display the depth of each goal in the search tree and the instantiation state of the goal. Additionally, it should display a unique number for each new invocation of a goal, i.e., the number should be the same at different ports of the same goal.

To develop a single-stepper we once again look at its various subproblems. In order to step through the execution of a PROLOG program the stepper must (a) interpret a PROLOG program, (b) detect which port of a goal is being entered, (c) compute depth, and (d) assign a unique number to each new goal. The latter three tasks, however, are done in the context of interpreting the first task. Furthermore, these tasks must be done simultaneously (for the same execution of the program).

The stepper should display the port, depth, and goal number during the same execution of the program. Having three different subprograms computing this information on different executions of the traced program would not be very useful. Once again, therefore, we have subproblems which are logically independent but cannot be mapped into separate physical subcomponents in the final program.

The above discussion suggests that the single-stepper itself may be developed by stepwise enhancement. To do so we first identify a program that may form the skeleton of the single-stepper. Since all the tasks mentioned earlier are defined around execution of a PROLOG program, a PROLOG interpreter is the obvious choice for a skeleton. The steps to develop a single-stepper by stepwise enhancement may be summarized as:

- Develop a PROLOG interpreter *solve*, Figure 1.
- *Enhance solve* into a meta-interpreter *port* that provides hooks to detect the ports of a goal, Figures 2 and 6.
- *Enhance solve* into a meta-interpreter *depth* that computes the depth of each goal (in the search tree), Figure 3.
- *Enhance solve* into a meta-interpreter *goalnum* that assigns a unique number to each goal, Figures 4 and 7.
- *Compose* the above interpreters into one interpreter *stepper0*, Figures 5, 6, and 7.

The program *stepper0* generated after composing only 'identifies' the ports and computes the required values. It neither displays the status, nor does it interact with the user. Therfore, it is not a single-stepper. Thus as a last step:

- *Enhance stepper0* to *stepper*, a program that interactively single steps through every port of every goal of the program.

PROLOG interpreters have been studied extensively in the literature [18]. Hence, we omit details describing the meta-interpreter *solve* and its enhancements. The *stepper0* program,

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

Figure 1: Vanilla

```
port(true).
port((A,B)) :-
      port(A),
      port(B).
port(Pred) :-
      call_and_fail_port(Pred),
      port_exec(Pred),
      exit_and_retry_port(Pred).

port_exec(Goal) :-
      sys(Goal),
      call(Goal).
port_exec(Head) :-
      clause(Head,Body),
      port(Body).
```

Figure 2: Detect 'ports'

```
depth(Goal) :- depth(Goal,0).

depth(true,Depth).
depth((A,B),Depth) :-
      depth(A,Depth),
      depth(B,Depth).
depth(Pred,Depth) :-
      Depth1 is Depth+1,
      depth_exec(Pred,Depth1).

depth_exec(Goal,Depth) :-
      sys(Goal),
      call(Goal).
depth_exec(Head,Depth) :-
      clause(Head,Body),
      depth(Body,Depth).
```

Figure 3: Compute Depth

```
goalnum(Goal) :-
      initialize_goalnum(0),
      goalnum(Goal).

goalnum(true).
goalnum((A,B)) :-
      goalnum(A),
      goalnum(B).
goalnum(Pred) :-
      gen_number(NewNum),
      goalnum_exec(Pred).

goalnum_exec(Goal) :-
      sys(Goal),
      call(Goal).
goalnum_exec(Head) :-
      clause(Head,Body),
      goalnum(Body).
```

Figure 4: Assign goal number

```
stepper0(Goal) :-
      initialize_goalnum(0),
      stepper0(Goal,0).

stepper0(true,Depth).
stepper0((A ',' B),Depth) :-
      stepper0(A,Depth),
      stepper0(B,Depth).
stepper0(Pred,Depth) :-
      Depth1 is Depth+1,
      gen_number(NewNum),
      call_and_fail_port(Pred),
      stepper0_exec(Pred,Depth1),
      exit_and_retry_port(Pred).

stepper0_exec(Goal,Depth) :-
      sys(Goal),
      call(Goal).
stepper0_exec(Head,Depth) :-
      clause(Head,Body),
      stepper0(Body,Depth).
```

Figure 5: Stepper0

```
call_and_fail_port(Goal) :-
      call_port(Goal).
call_and_fail_port(Goal) :-
      fail_port(Goal),fail.

exit_and_retry_port(Goal) :-
      exit_port(Goal).
exit_and_retry_port(Goal) :-
      retry_port(Goal),fail.

call_port(Goal).
fail_port(Goal).
exit_port(Goal).
retry_port(Goal).
```

Figure 6: Support for 'ports'

```
initialize_goalnum(Number) :-
      retractall(('$goalnum'(X)),
      asserta('$goalnum'(Number)).

gen_number(NewNumber) :-
      retract(('$goalnum'(LastNumber)),
      NewNumber is LastNumber +1,
      asserta('$goalnum'(NewNumber)).
```

Figure 7: Support for goalnum

Figures 5 Figures 5, 6, and 7, contains some predicates, such as *call_port/1* and *fail_port/1*, that are not very useful in *stepper0*. They have been put in place with the enhancement of *stepper0* to *stepper* in mind. Their utility will be explained as they are used during enhancement.

4.1 Enhance *stepper0* to *stepper*

The enhancement for *stepper* requires trapping a program's execution just before a port of each goal is entered. The procedure *call_port/1* is called before entering the call port of the object goal. (This procedure was developed in the *port* enhancement, Figures 2 and 6, as a hook to perform processing before entering the call port of a goal.) Similarly, the procedures *fail_port/1*, *exit_port/1*, and *retry_port/1* provide hooks for trapping execution before entering into the fail, exit, and retry ports respectively. Thus, the interaction needed to display status before entering each port may be performed by enhancing these predicates.

The status that a tracer displays at each port includes the goal number, depth, port, and the goal. However, only the the goal and the port to be entered are available to the procedures *call_port/1*, *fail_port/1*, *exit_port/1*, and *retry_port/1*. The goal number and depth information, computed by enhancements *depth* and *goalnum*, Figures 3, 4, and 7, should be propagated from the third clause of *stepper0/2* to the procedures trapping the entry into ports. The enhancement requires changing the third clause of *stepper0/2* to:

```
stepper(Pred, Depth) :-
    Depth1 is Depth+1,
    gen_number(GoalNum),
    % propagate Depth and GoalNumber to the port handlers
    call_and_fail_port_stepper(Pred, Depth1, GoalNum),
    stepper_exec(Pred, Depth1),
    % propagate Depth and GoalNumber to the port handlers
    exit_and_redo_port_stepper(Pred, Depth1, GoalNum).
```

It also requires enhancing the port handling procedures into those given in Figure 8. The enhancements include propagating the depth and goal number information and addition of the predicate *wait_at_port_stepper/4* that calls *process_command/4* to display the the status line and process commands. Even though the predicate *wait_at_port_stepper/4* has two clauses, it succeeds exactly once, therefore preserving the execution behavior of PROLOG.

Notice that when enhancing a predicate we also change its name. Thus *stepper0* has been renamed to *stepper* and *call_and_fail_port* has been renamed to *call_and_fail_port_stepper*. The renaming of predicates is assumed to be performed over the complete program. In the discussion we only present clauses that have been changed, beyond just renaming to maintain correct behavior.

The command interface provided by *process_command/4* is developed with future enhancements in mind.

```
call_and_fail_port_stepper(Goal, Depth, GoalNum) :-
        call_port_stepper(Goal, Depth, GoalNum).
call_and_fail_port_stepper(Goal, Depth, GoalNum) :-
        fail_port_stepper(Goal, Depth, GoalNum),
        fail.

exit_and_redo_port_stepper(Goal, Depth, GoalNum) :-
        exit_port_stepper(Goal, Depth, GoalNum).
exit_and_redo_port_stepper(Goal, Depth, GoalNum) :-
        redo_port_stepper(Goal, Depth, GoalNum),
        fail.

% Extended procedures for processing at the goal's ports
call_port_stepper(Goal, Depth, GoalNum) :-
        wait_at_port_stepper(call,Goal, Depth, GoalNum).
fail_port_stepper(Goal, Depth, GoalNum) :-
        wait_at_port_stepper(fail,Goal, Depth, GoalNum).
exit_port_stepper(Goal, Depth, GoalNum) :-
        wait_at_port_stepper(exit,Goal, Depth, GoalNum).
redo_port_stepper(Goal, Depth, GoalNum) :-
        wait_at_port_stepper(redo,Goal, Depth, GoalNum).

wait_at_port_stepper(Port, Goal, Depth, GoalNum) :-
        process_command(Port, Goal, Depth, GoalNum),
        !.
wait_at_port_stepper(_Port, _Goal, _Depth, _GoalNum).
```

Figure 8: Port handling

4.2 Generic User Interface

There are several actions that are performed for all the tracer commands, for instance parsing a command or displaying the status of a goal. There are also actions that are performed by only a specific command, such as printing a '**' in front of the status line of a goal that has been defined as a spy point. Based on this observation we enhance *stepper0* to have a generic user interface.

The interface consists of two parts. First, a collection of generic predicates, Figure 9, to perform the tasks common for all the commands, such as display the status, query command, and process a command. Second, calls from these generic predicates to software hooks, i.e., calls to as-yet-undefined procedures, to perform special processing needed for individual features.

The generic predicates call the following hooks.

- *print_goal_state_hook(+Port, +Goal, +Level, +NodeNum)*
  This hook is called before the status of a goal is printed. Special processing for printing of '**' when the goal is a spy point or '>' when the skip command is permitted at a goal may be performed at this hook.

- *map_char_to_command_hook(+Char, -Command)*
  This hooks maps a character input as a command to some internal name.

  ```
  map_char_to_command_hook(0'c, creep).
  map_char_to_command_hook(0'a, abort).
  map_char_to_command_hook(0'r, retry).
  ```

  Note $0'X$ is the numeric code for character X.

- *process_option_hook(+Command, +Argument, +Port, +Goal, +Level, +NodeNum)*
  This hook processes a tracer option. For the single stepper the following clauses do the processing needed.

  ```
  process_option_hook(next, _Argument, _Port, _Goal, _Depth, _GoalNum).
  process_option_hook(abort, _Argument, _Port, _Goal, _Depth, _GoalNum)
      :- abort.
  ```

Some tracer commands require additional information. For example, to *show ancestors* of a goal the list of ancestors is needed, or to *show the clause* selected for a goal clause selected should be known. To support these features the generic command interface would have to be enhanced to propagate the information needed from the interface to the hooks. This may be done by *propagating the context* via the port handler and the command interface.

The command requested by the user must be communicated by the user interface to the rest of the program. Some of these commands, such as *leap*, have effect even after backtracking from the current goal. Since instantiation of variables is lost on backtracking,

```
process_command(Port, Goal, Level, NodeNum) :-
    print_goal_state(Port, Goal, Level, NodeNum),
    query_command(Command, Argument, Port),
    process_option(Command, Argument, Port, Goal, Level, NodeNum).

print_goal_state(Port, Goal, Level, NodeNum) :-
    print_goal_state_hook(Port, Goal, Level, NodeNum),
    fail.
print_goal_state(Port, Goal, Level, NodeNum).

query_command(Command, [ ], Port) :-
    unleashed_port(Port), !,
    Command = next.
query_command(Command, Argument, Port) :-
    query_option(Char, Argument),
    map_char_to_command(Char, Command).

map_char_to_command(Char, Command) :-
    map_char_to_command_hook(Char, Command), !.
map_char_to_command(Char, Char).

process_option(Command, Argument, Port, Goal, Level, NodeNum):-
    process_option_hook(Command, Argument, Port, Goal, Level, NodeNum).
process_option(Command,_Argument, Port, Goal, Level, NodeNum) :-
    % Command not processed at any hook
    % Give an error message and get another command
    invalid_command_message(Command),
    process_command(Port, Goal, Level, NodeNum).
```

Figure 9: Generic Command Interface

propagation of information across "or" branches of a search tree may be done by "asserting" the information in the PROLOG database.

We assume that, if needed, the *process_option_hook* predicate for a command asserts a fact:

control_flag(Command, Information).

This flag is used by a partial-tracer to globally announce the selection of a *Command* and provide the *Information* needed for its processing.

In addition we assume the existence of a support predicate *remove_control_flag* that removes all the *control_flag/2* facts (or clauses) from the PROLOG database.

5 Developing the Tracer

As stated earlier, we build the tracer by first developing a set of partial-tracers, each implementing a set of features, and then composing these partial-tracers into a single program. The partial-tracers are developed by enhancing the single-stepper described in the previous section. This section describes the enhancements performed to develop each partial-tracer.

For the sake of brevity we do not present the complete code for each of the partial-tracers. Instead we present only those code segments that are modified or added. To get the complete picture we call upon the reader to do some mental manipulations, using the following points as a guide:

- The program *stepper* developed so far consists of:
  1. The procedures *stepper/1*, *stepper/2*, and *stepper_exec/2* derived by:
     (a) renaming the predicates *stepper0/1*, *stepper0/2*, and *stepper0_exec/2* of Figure 5 and
     (b) replacing the third clause of the resulting *stepper/2* by the clause given in Section 4.1.
  2. The code for assigning a unique number to each goal, Figure 7.
  3. The code for trapping entries into the various ports of a goal and interfacing with the user interface, Figure 8.
  4. The generic user interface, Figure 9.
  5. The implementation of hooks for the single stepper, Section 4.2.
- The enhancements performed in this section will create:
  1. *leap*, a partial-tracer providing the leap and quasi-skip commands,
  2. *skip*, a partial-tracer providing the skip and true commands, and
  3. *retry*, a partial-tracer providing the retry and fail commands.

- These enhancements are performed by:
  1. redefining predicates of the *stepper*,
  2. adding some new predicates, and
  3. adding hook predicates.
- When a predicate from *stepper* is modified it is also renamed. The new name is created by introducing, as a suffix, an affix, or a substring, the name of the partial-tracer, i.e., *leap*, *skip*, or *retry*, in the original name. Even though only modified clauses are sometimes shown, the renaming is assumed to be performed over the entire program.
- Renaming does not remove the previous predicates, it only adds new predicates. Thus, the enhancements performed to one partial-tracer do not interfere with those applied to the other.
- The newly added predicates are shown in *italics*. The behavior of these predicates is either described as pseudo-code or in PROLOG.
- The hook predicates retain the same name in all the partial-tracers. Their actions are either described or are assumed implicit.

5.1 Enhancements for Leap and Quasi-skip

When debugging a large program, instead of tracing the execution of every goal, one may wish to inspect a program's state on entry into the ports of some specific goals, called *spy points*. The *leap* and *quasi-skip* commands provide support for such trace. They execute the object program without interacting with the user or displaying the status until a spy point is reached. A goal is designated as a spy point by a separate command outside the tracer.

The quasi-skip command disables interaction with the user until either the execution enters another port of the current goal or if during the execution a goal marked as a spy point is reached. In contrast, a leap command disables interaction until the goal marked as a spy point is reached. Furthermore, though the leap command may be requested at any port of a goal, the quasi-skip command may be requested only at the call or redo ports.

The *stepper* can be enhanced to a partial tracer *leap* that provides these two commands. To disable interaction with the user the procedure *wait_at_port_stepper/4*, Figure 8, is (renamed and) modified just before control is passed to the generic interface.

```
wait_at_port_leap(Port,Goal,Depth,GoalNum) :-
    resume_trace_at_quasi_skip_goal(GoalNum),
    resume_trace_at_spy_point(Goal),
    process_command(Port,Goal,Depth,GoalNum),
    !.
wait_at_port_leap(_Port,_Goal,_Depth,_GoalNum).
```

The *process_option_hook* predicate for this partial tracer asserts a control flag that gives the number of the goal at which the leap or the quasi-skip command was requested.

The predicate *resume_trace_at_spy_point/1* guards the command interaction part. When either a leap or a quasi-skip command is being processed, this predicate checks if the goal whose port is being entered is a spy point. If so, it removes the control flag and enables normal interaction, otherwise it simply fails.

```
resume_trace_at_spy_point(Pred) :-
        leap_or_quasi_skip_in_progress, !,
        spy_point(Pred),
        remove_control_flag.
resume_trace_at_spy_point(Pred).
```

The predicate *resume_trace_at_quasi_skip_goal/1* provides enables interaction when executions reaches another port, typically exit or fail, of the goal at which the quasi-skip command was requested.

```
resume_trace_at_quasi_skip_goal(GoalNum) :-
        control_flag(quasi_skip, StartNum), !,
        StartNum = GoalNum,
        remove_control_flag.
resume_trace_at_quasi_skip_goal(GoalNum).
```

Note the use of the cut, '!', in the two procedures. It makes the procedure fail if a leap or a quasi-skip command is in progress and the conditions to enable interaction are not met. If neither the leap nor the quasi-skip command is in progress these predicates succeed, allowing the tracing continue as usual.

5.2 Enhancements for Skip, Unconditional True

Like the quasi-skip command, the skip command disables tracing until the current goal is completely executed. It differs with the quasi-skip command in that (a) the skip command can only be requested at a call port and (b) this command does not enable interaction when a spy point is reached. Interaction is resumed only after the goal requested to be skipped either succeeds or fails.

When the complete execution of a goal is not being traced, the goal need not be interpreted by the tracer. Instead it may be executed directly by the PROLOG engine, thereby avoiding the runtime and memory overheads associated with interpretation. The enhancements to build a partial tracer *skip* from *stepper* reflect this observation. The enhancement requires modifying how a goal is interpreted.

A true command forces a goal to succeed unconditionally. This command too requires changing the underlying PROLOG interpreter. Given this similarity between the true and skip commands their enhancements have been grouped together.

In the single stepper, an atomic goal is interpreted by the predicate *stepper_exec/2*. A change in the method of interpreting an atomic goal, as needed for the *skip* and *true* commands, requires changing this predicate. The enhancement may be achieved by adding at the beginning *stepper_exec/2* (and renaming the procedure to *skip_exec/2*) the following two clauses:

```
skip_exec(Pred, _Depth) :-
        is_true_on, !,
        reset_true.
skip_exec(Pred, _Depth) :-
        is_skip_on, !,
        reset_skip,
        call(Pred).
```

The predicates *is_true_on* and *is_skip_on* use the control flag asserted by the *process_option_hook* predicate of this partial tracer to detect whether a skip or a true command has been requested. The predicates *reset_true* and *reset_skip* remove the control flag.

5.3 Enhancements for Retry and Fail

The retry and fail commands are invoked with the number of a goal as an argument. The number refers to a goal that has already been invoked. A retry command causes the tracer to go back to the call port of the corresponding instance of the goal requested to be retried. A fail command forces the selected goal instance to fail.

The *retry* partial tracer implements the retry and fail commands. It too is developed by enhancing *stepper*. Since the commands require falling back to an earlier state in the computation, the *process_command_hook* predicate for this partial tracer asserts a control flag giving the command that was invoked and the number of the goal that is requested to be failed or retried.

An earlier state in the computation of a partial tracer may be reached by forcing a failure at all the choice points starting backwards from the selection of the retry or fail command to the instantiation of the goal selected. The failures, therefore, have to ripple backwards through the port handling procedures of Figure 8 into the procedures interpreting PROLOG clauses, the enhancements to *stepper* from *stepper0* of Figure 5.

The procedures in Figure 10 enhance the corresponding procedures in Figure 8 so as ripple the failures back through the port handling procedures. The behavior of the support procedures used in Figure 10 is described by the pseudo-code below.

going_back_for_retry ←
    if the control flag indicates that a retry of fail
    is selected then fail
    succeeds otherwise

```
call_and_fail_port_retry(Goal,Depth,GoalNum) :-
        call_port_retry(Goal,Depth,GoalNum).
call_and_fail_port_retry(Goal,Depth,GoalNum) :-
        fail_this_goal(GoalNum),
        going_back_for_retry,
        fail_port_retry(Goal,Depth,GoalNum),
        fail.
exit_and_redo_port_retry(Goal,Depth,GoalNum) :-
        exit_at_port_retry(Goal,Depth,GoalNum).
exit_and_redo_port_retry(Goal,Depth,GoalNum) :-
        going_back_for_retry,
        redo_at_port_retry(Goal,Depth,GoalNum),
        fail.
        ⋮
wait_at_port_retry(Port,Goal,Depth,GoalNum) :-
        process_command(Port,Goal,Depth,GoalNum),
        !,
        % trigger failure if a retry or fail
        going_back_for_retry.
wait_at_port_retry(_Port,_Goal,_Depth,_GoalNum).
```

Figure 10: Port handling enhanced to support retry and fail commands

fail_this_goal(GoalNum) ←
    if control flag indicates that GoalNum is marked as
    the goal to be failed then
        remove the control flag from the database

The above enhancement bypasses entry into the exit or fail ports of goals that are still active. This also implicitly inhibits the selection of unselected clauses in the PROLOG interpreter. Since the tracer does not trace the execution of system goals, the choice points for these goals should also be removed. The following modifications to the procedure *stepper_exec/2*, the one that processed atomic goals in a single stepper, cut the choice-points when failures are triggered due to a retry or a fail command. (The modified procedure has been renamed to *retry_exec/2*.

```
retry_exec(Goal,Depth) :-
    sys(Goal),
    !,
    call(Goal),
    (true ; retry_or_fail, !, fail).
retry_exec(Head,Depth) :-
    clause(Head,Body),
    retry(Body,Depth),
    (true ; retry_or_fail, !, fail).
```

The predicate *retry_or_fail/0* succeeds only if the control flag indicates that a retry or a fail command has been chosen.

The above changes force the state of a partial tracer back to where a given goal was first invoked. If the retry command is selected this backtracking must be stopped once the desired state is reached. This is done by adding the goal *check_to_stop_retrying/1* to the last clause of the procedure *stepper/2* to give (after renaming to *retry/2*):

```
retry(Pred,Depth) :-
    Depth1 is Depth+1,
    gen_number(GoalNum),
    check_to_stop_retrying(GoalNum),
    call_and_fail_port_retry(Pred,Depth1,GoalNum),
    retry_exec(Pred,Depth1),
    exit_and_redo_port_retry(Pred,Depth1,GoalNum).
```

The procedure that tests when the backward ripple of failures may be stopped is defined as follows:

```
check_to_stop_retrying(_GoalNum).
check_to_stop_retrying(GoalNum) :-
    retry_this_goal(GoalNum),
    reset_retry,
    initialize_goalnum(GoalNum),
    check_to_stop_retrying(GoalNum).
```

The first clause is invoked before entering the call port of an object goal; this clause always succeeds. The second clause is selected on backtracking due to the failure of a goal. The predicate *retry_this_goal/1* checks if the goal that failed has been requested to be retried. If it has been, the predicate *reset_retry* removes the control flag and the predicate *initialize_goalnum/1*, Figure 7, resets the current unique number to the number of this goal. The call port of the goal to be retried is then reentered via a recursive call to *check_to_stop_retrying/1* procedure. The recursive call is performed so that a subsequent retry command for the same goal behaves correctly.

6 Conclusions

Due to the nature of the problem, conventional program modularization techniques cannot be applied to the development of a Prolog tracer. Each new trace feature has to be built by modifying the existing code. This soon leads to a deterioration in the program structure.

We apply the program composition technique [6, 16] to avoid this problem. New features are built independently on a common base program, a single-stepper, and then merged together by composition. The single-stepper is itself developed using stepwise enhancement by composing three enhancements of a base PROLOG meta-interpter.

Our method relies on the knowledge that various trace features are independent of each other and have to be built on the same program. Its success does not depend on any other property specific to the tracer. This method can thus be used in the development of other programs with similar property.

The work is part of an effort to develop tools for Prolog programming, ongoing at Case Western Reserve University. In fact, the tracer as described here was originally written in 1988. Research at CWRU over the past seven years has focussed on formalizing the 'structure-preserving' operations, to use the terminology of this paper, that can be performed on programs. readers interested in seeing the ideas developed should look at [15, 5, 4].

Acknowledgements: This research was supported by NSF grant and REU supplement for the third author. The first author thanks Aditya Srivastava of Texas Instruments for discussions on problems and implementation of a PROLOG Tracer. We thank the Composers Group at the Computer Engineering and Science Department at CWRU for providing an environment conducive for research.

References

1. Basili, V. and A. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. of Soft Eng*, Vol. SE-1, No. 4 December 1975, pp. 390-396.

2. Bowen, D.L., L. Byrd, F. Periera, L Periera, D.H.D Warren, *DEC-10* PROLOG *Users Manual*. Occasional Paper 27, DAI, University of Edinburgh, November, 1982.

3. Byrd, L., "Understanding the Control flow of PROLOG programs," In Tarnlund, S (Ed), *Proceeding of the Logic Programming Workshop*, 1980, pp 127-138.

4. Jain, A., L. Sterling and M. Kirschenbaum, "Towards Reusability Based Upon Similar Computational Behavior," In *Proceedings of the* 7^{th} *International Conference on Software Engineering and Knowledge Engineering*, Rockville, Maryland, USA, June 1995. To appear.

5. Kirschenbaum, M., L. Sterling and A. Jain, "Relating Logic Program via Program Maps," *Annals of Mathematics and Artificial Intelligence*, 8(III-IV):229–246, 1993.

6. Lakhotia, A. and L. Sterling, "Composing Recursive Logic Programs with *Clausal Join*," *New Generation Computing*, **6**, 1988, pp. 211-255.

7. Lakhotia, A. and L. Sterling, "Program Development by Stepwise 'Enhancement'," In *Proceedings of the* 2^{nd} *International Conference on Software Engineering and Knowledge Engineering*, pages 78–83, Skokie, Illinois, June 1990.

8. Lehman, M.M. and L.A., Belady, "Program Evolution," *Academic Press*, 1985.

9. Levi, G. and G. Sardu, "Partial Evaluation of Metaprograms in a 'Multiple Worlds' Logic Language," Proceedings Workshop on Partial and Mixed Computation, Denmark, August 1987.

10. O'Keefe, R., "On the Treatment of Cuts in Prolog Source-Level Tools," Proceedings of the Symposium on Logic Programming, Boston, 68-72, 1985.

11. Parnas, D. L., "On the Design and Development of Program Families," *IEEE Trans on Software Engineering*, Vol SE-2, No. 1, March 1976.

12. Plummer, D., "Coda: An Extended Debugger for PROLOG," AI TR87-84, April 1987

13. Sergot, M.J., "A Query-the-User Facility for Logic Programming," *Integrated Interactive Computer Systems*, Ed. P. Degano and E. Sandewall, North Holland, 1983.

14. Shapiro E.Y., "Logic Programs with Uncertainties: A Tool for Implementing Rule-Based Systems," *Proceedings IJCAI 8* Karlsruhe, Germany, 1983, pp. 529-532

15. Sterling, L. and M. Kirschenbaum, "Applying Techniques to Skeletons," In J.M.J. Jacquet, editor, *Constructing Logic Programs*, Chapter 6, pages 127–140. John Wiley, 1993.

16. Sterling, L. and A. Lakhotia, "Composing PROLOG Meta-Interpreters," *Proceedings of the 5^{th} International Conference on Logic Programming*, 1987.

17. Sterling, L. and R.D. Beer, "Meta-Interpreters for Expert System Construction," *Journal of Logic Programming*, 6(1,2):163–178, 1989.

18. Sterling, L. and E. Shapiro, "The Art of Prolog," *MIT Press*, 1986.

19. Yalcinalp, L.Ü. and L. Sterling, "An Integrated Interpreter for Explaining PROLOG's Successes and Failures," Case Western Reserve University, CES TR-88-04, April 88.

20. Wirth, N., "Program Development by Stepwise Refinement," *Comm. ACM*, **14**, 4, 1971 pp. 221-227

A Tracer: Result of composition

```
trace(Goal) :-
        reset_control_flag,
        reset_leashing_mode,
        tracer(Goal,0).

tracer(true,_Depth).
tracer((GoalA,GoalB),Depth) :-
        tracer(GoalA,Depth),
        tracer(GoalB,Depth).
tracer(Pred,Depth) :-
        Depth1 is Depth+1,
        gen_number(GoalNum),
        check_to_stop_retrying(GoalNum),
        call_and_fail_port_tracer(Pred,Depth1,GoalNum),
        tracer_exec(Pred,Depth1),
        exit_and_redo_port_tracer(Pred,Depth1,GoalNum).

tracer_exec(_Pred,_Depth) :-
        is_true_on,
        !,
        reset_true.
tracer_exec(Pred,_Depth) :-
        test_and_reset_skip,
        !,
        call(Pred).
tracer_exec(Goal,_Depth) :-
        sys(Goal),
        !,
        call(Goal), (true;retry_or_fail,!,fail).
tracer_exec(Head,Depth) :-
        clause(Head,Body),
        tracer(Body,Depth), (true;retry_or_fail,!,fail).

call_and_fail_port_tracer(Goal,Depth,GoalNum) :-
        call_port_tracer(Goal,Depth,GoalNum).

call_and_fail_port_tracer(Goal,Depth,GoalNum) :-
        fail_this_node(GoalNum),
        going_back_for_retry_or_fail,
```

```
        fail_port_tracer(Goal,Depth,GoalNum),
        fail.

exit_and_redo_port_tracer(Goal,Depth,GoalNum) :-
        exit_port_tracer(Goal,Depth,GoalNum).

exit_and_redo_port_tracer(Goal,Depth,GoalNum) :-
        going_back_for_retry_or_fail,
        redo_port_tracer(Goal,Depth,GoalNum),
        fail.


call_port_tracer(Goal,Depth,GoalNum) :-
        wait_at_port_tracer(call,Goal,Depth,GoalNum).
fail_port_tracer(Goal,Depth,GoalNum) :-
        wait_at_port_tracer(fail,Goal,Depth,GoalNum).
exit_port_tracer(Goal,Depth,GoalNum) :-
        wait_at_port_tracer(exit,Goal,Depth,GoalNum).
redo_port_tracer(Goal,Depth,GoalNum) :-
        wait_at_port_tracer(redo,Goal,Depth,GoalNum).


wait_at_port_tracer(Port,Goal,Depth,GoalNum) :-
        resume_trace_at_spy_point(Goal),
        resume_trace_at_quasi_skip_node(GoalNum),
        process_command(Port,Goal,Depth,GoalNum),
        !,
        going_back_for_retry_or_fail.
wait_at_port_tracer(_Port,_Goal,_Depth,_GoalNum).
```