中華民國八十四年全國計算機會議論文集
Proceedings of National Computer Symposium 1995

邏輯敘述程式的視覺化
Visualization of Well Engineered Logic Specification Programs*

施國琛　郭經華　余文祥
Timothy K. Shih, Chin-Hwa Kuo, and Wen-Shan Yu
淡江大學資訊工程系
Department of Computer Science and Information Engineering
Tamkang University
Tamsui, Taipei Hsien, Taiwan
R.O.C.
email: tshih@cs.tku.edu.tw
fax: Intl. (02) 623-8212

摘要

*Prolog*是邏輯敘述程式語言中最受歡迎的一種.但是當一個程式設計者在發展一大型軟體系統時,一個良好的程式發展環境是必需的. *SPEC*計畫是作者們早期開發出的產品.但是, *SPEC*計畫在支援需求分析及系統設計上,仍不夠完美.在這論文中,我們提出一個新的系統- *VSPEC*,其中包括一邏輯敘述程式流覽編輯工具,以及一宣告式邏輯敘述程式的語意分析器,利用一 *"and-or"*的樹狀圖,來表示一個邏輯程式的語意. *VSPEC*與一簡化了的*SPEC*系統結合後,現已可在微軟公司的視窗系統下執行.在此論文中,我們提及新的*SPEC*語言,以及程式轉換法,將敘述程式轉成邏輯程式.除此之外,我們更設計了一些高階程式指令,以便使用者撰寫更好的邏輯程式.

關鍵詞：敘述,軟體工程,視覺化,邏輯程式

Abstract

Prolog is one of the most popular languages in logic programming. However, when programmers deal with the task of developing large systems, A well-designed programming environment is necessary. The Specification Processing Environment with Controls (SPEC) project was earlier developed by the author. However, it is still lcaking supporting tools for the analysis and design of logic programs. In this paper, we propose a visualization environment (VSPEC) that facilitates a hyper-text like navigation of large logic specification programs.. In addition to the hyper-text exitor, a declarative specification browser utilizing an "and-or" tree showing the semantics of the specification program is also addressed. The visualization tool is integrated with a simplified version of SPEC running on the MS Windows. The revised SPEC language is addressed, followed by a discussion of the program transformation algorithm that generates Prolog programs from their specifications. A number of language constructs are also discussed.

Key words: Specification, Software Engineering, Visualization, Logic Programming

1 Introduction

Prolog is one of the most popular languages in logic programming. As Prolog became widely used for research in Artificial Intelligence as well as for commercial and industrial work, its drawbacks, such as the unsoundness of implementations of negation and the lack of control facilities, was realized by researchers [8, 9, 1]. Several extensions to and revisions of Prolog have been defined (e.g., IC-Prolog [2], Epilog [11], Parlog [3], Concurrent Prolog [12, 13], MU-Prolog [9]), some with the intent of providing the programmer more control over the execution of a program and others aimed specifically at parallel programming applications.

However, when programmers deal with the task of developing large systems, a programming language is not enough. A well-designed programming environment is necessary [6, 7, 4, 16, 15]. A well-designed programming environment not only supports the efficient implementation of programs, but also encourages a good style of analysis, robustness and validation of implementation, and ease of maintenance. A good approach is to design a system to support the entire software development life cycle including requirements analysis, specification, design, implementation, testing, verification and maintenance.

The SPEC project is a logic programming environment developed at Santa Clara University[1]. The project is designed to support the software development life cycle from requirements analysis, specification, design, implementation, and testing through maintenance. A specification language with high level declarative constructs as well as execution control facilities was developed. The system supports the separation of declarative and control specifications, enabling one to generate different implementations of the declarative specification by changing the control strategy. The users are asked to provide formal documentation and performance constraints as part of the specification of their programs. The system also provides a debugger and other tools, including a test generator, a verification assistant, and statistical analysis tools. Details of the SPEC project are discussed in [18, 19, 20].

However, in the development of SPEC system, it is still lacking supporting tools for data and control flow analysis. Even though there are a number of well-developed structured analysis/design and object-oriented analysis/design tools available, they are not quite suitable for logic programs. Structured analysis/design tools are suitable for procedural languages and the methodology focus on describing *how* data or controls are passed in between modules. Object-oriented analysis/design tools, with a different focus, pay more attention on *how* data are shared and *how* methods are invoked via message passing. These methodologies are relatively procedural in that their main strengths are to describe *how* system should be built. Logic programming, on the other hand, focuses on the declarative aspect of a system. Declarative programming is one of the most important goals of logic programming researches. A declarative logic program describes *what* system the program is trying to model instead of *how* the program is executed. Current paradigms of requirement analysis, however, do not provide such a declarative methodology for logic programmers. The need of a good analysis tool for declarative programming is thus worthwhile be investigated. Due to this reason, VSPEC aims to provide such a methodology and tool for the logic program developers.

Figure 1 gives the system architecture of the VSPEC project. Under the integrated graphical user interface, the specification hyper-text editor allows the user to navigate through pieces of a mini-specification. The declarative specification browser shows the declarative semantics of the design in an "and-or" tree. The translator takes as input the user's specification, and generates a Prolog program which can be run under our interpreter/debugger. The debugger also comes with a SLD tree viewer that allows a real time snap shot of a program execution. The VSPEC project also has an automatic testing tool for system verification. Due to the limitation of space, in this paper, we focus on the discussion of the hyper-editor, the declarative specification browser, and the translator.

[1] The SPEC project was originally proposed by Dr. Ruth E. Davis.

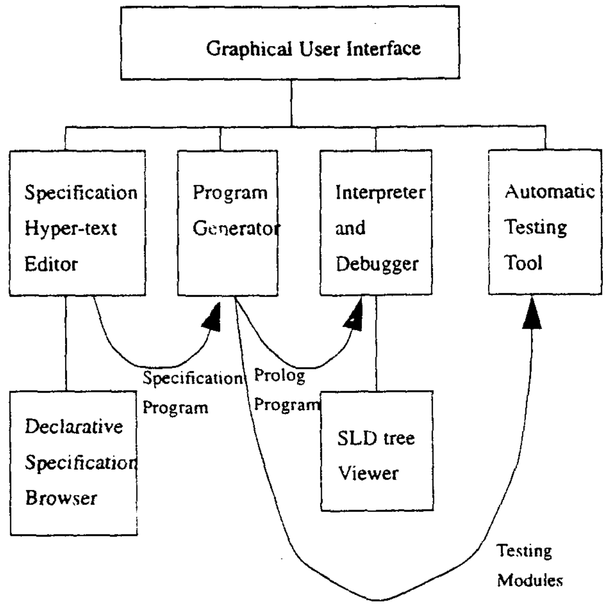

Figure 1: System overview of the VSPEC project

This paper is organized as the following. Section 2 introduces a hyper-text like navigation methodology along with a declarative program browser for logic program/specifications. Section 3 discusses a revised version of the SPEC language earlier developed by the authors. A short conclusion showing our contributions is given in section 5.

2 Hyper-text Navigation of Logic Specifications

When an individual constructs a logic specification program, one does not think about the declarative semantics of the program linearly. In spite of different software architecture approaches, such as a top-down, or a bottom-up strategy, a programmer usually cross references different pieces of a specification while constructing a new piece. In our VSPEC environment, we provide a hyper-text like specification editor which allows a user to click on a highlighted Prolog predicate in the specification in order to perform a cross reference among different parts of a specification. Each specification part is displayed on a separate window. The user is given an option to close a specification window or leave it on the screen after changing the window focus to another specification. This hyper-text specification editor, enabling a convenient navigation, allows the user to stepwise refine their specification programs.

During the construction of a specification program, the user can use a declarative specification browser to review the current status of his/her specification program. The browser displays an "and-or" tree like structure representing the semantics of the program. The browser, while activated, will ask the user to give a predicate which serves as the root node of the semantics tree. Starting from the given root predicate, the browser searches for predicates used by the root predicate. If the root predicate is defined as a disjunction of more than one clause, an "or" subtree is expanded. For instance, in figure 2, predicate `solution/1` is defined in two clauses and an "or" subtree of two branches is expanded. It is possible for an "or" subtree to consist of only one branch (e.g., the subtree of predicate **equeen**). The declarative specification browser then looks at the body definition of each clause of the expanded predicate and construct an "and" subtree if the body definition of a clause is a conjunction. For example, the second clause of the `solution/1` predicate has a body definition which is a conjunction of predicates `solution/1`, `member/2`, and `noattack/2`. Thus an "and" subtree of three branches is expanded. Disjunctions are treated as separate clauses and "or" subtrees are used. The expansion continues until all predicates are expanded. Some exceptions that terminate a branch of the expansion are the *expanded predicates*, the *system predicates*, and the *undefined predicates*. An expanded predicate is a predicate that exists in an upper level of the tree. A predicate will not be expanded more than once in the tree. A system predicate is one provided by the Prolog interpreter (e.g., write/1). An undefined predicate has not yet been declared in the specification program when the browser is invoked. An empty black box is used in the tree to indicate an undefined predicate. Figure 2 shows the "and-or" tree of the eight queen's problem solving specification.

In addition to the "and-or" tree expansion, there are three types of annotations used in a semantics tree. A "down" arrow annotation indicates that data is provided by the parent predicate. That is, the parameter in a parent predicate should be instantiated (or bound) in order to provide information for its child predicates in the tree. An "up" arrow, on the other hand, indicates the information is collected by the child predicates and passed to their parent predicate. An "bidirectional" arrow indicates that the information is passed in both ways, allowing the invertibility of logic programming. An edge in the tree can have multiple arrows if the predicate has more than a parameter.

3 The revised SPEC Language

The revised SPEC language enables a specification program to be compiled to a standrd Prolog program. A specification program contains one or more specifications. Each specification is a combination of the following objects:

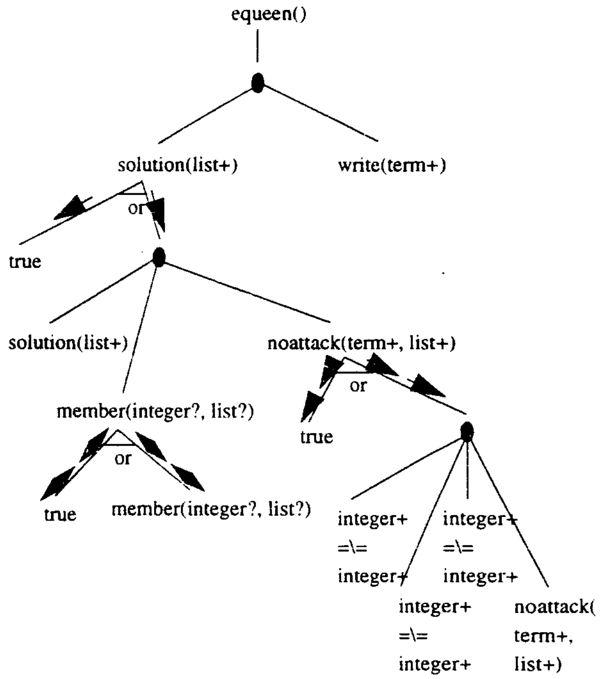

Figure 2: A "and-or" tree used in the declarative program browser

- Signature
- Preconditions
- Specification Body
- Postconditions
- Test Cases (optional)
- Comments

A signature consists of the name of the specification plus types and modes information of the specification. Types are treated as unary predicates in Prolog. For instance, `atom(X)` holds if object `X` is an atom. Modes can be input, output, or bidirectional, indicated by +, -, or ?, respectively. Modes are postfix declarations attached to type tags (e.g., `atom`, `integer`). The signature of a specification is used in type checking, and in the generation of annotation arrows while a declarative specification semantics tree is expanded. Preconditions are a conjunction consisting of predicates which must hold in order for the specification to produce useful results. The default precondition is `true` which always holds. The specification body is a number of Prolog clauses. Special language constructs in SPEC [18, 19, 20] are also allowed. Postconditions of a specification hold in the time point after a successful execution of its corresponding specification. The default postcondition is `none` that indicates no assertion is necessary. Postconditions are also conjunctions of Prolog predicates. Test cases are optional. If provided, the automatic testing tool will test the specification according to the test cases on the demand of a user. Comments are standard ASCII code that is not analysed by the system.

A specification program is compiled into a standard Prolog program with some renaming. Each specification, while translated to a Prolog program, consists of two parts. The first part is a predicate of one clause with its name as the name of the specification. The body of this part of the predicate is a conjunction of the precondition, a call to the renamed specification body, and the assertion of the postconditions. The renamed specification body has its name attached with a special tag "_body" right after the name of the specification. For example, the following specification is translated to Prolog code:

Specification in VSPEC

```
specification noattack(X/Y:term+, L:list+).
precondition ::= X >= 1, X =< 8,
                 Y >=1, Y =< 8.
bodyspec ::=
  noattack(_, []).
  noattack(X/Y, [X1/Y1 | Others]) :-
    X =\= X1,
    Y =\= Y1,
    Y1-Y =\= X1-X,
    Y1-Y =\= X-X1,
    noattack(X/Y, Others).
postcondition ::= none.
testcase ::= noattack(1/1, [3/4]).
             noattack(1/1, [1/4]).
             noattack(1/2, [2/4,3/1,4/3]).
comments ::= 'condition must hold for the
              eight queen problem'.
```

Prolog program generated

```
noattack(X/Y, L) :-
  X >= 1, X =< 8, Y >=1, Y =< 8,
  noattack_body(X/Y, L).
noattack_body(_, []).
noattack_body(X/Y, [X1/Y1 | Others]) :-
  X =\= X1,
  Y =\= Y1,
  Y1-Y =\= X1-X,
  Y1-Y =\= X-X1,
  noattack(X/Y, Others).
```

Note that if the postcondition is **none**, no assertion is made. Similarly, if the precondition is by default, no precondiction call is generated.

4 VSPEC Language Constructs

In this section we discuss some of the language constructs developed in our VSPEC project.

4.1 Conditional Constructs

Alternative solution paths for a given procedure may be mutually exclusive so that only one branch of the search space needs to be investigated for a given call. Moreover, for procedures involving a large case analysis, a general conditional structure is helpful. Instead of using disjunctions or several clauses with many redundant subgoals, two high level language constructs are provided to enhance readability and efficiency. Before discussing the syntax and semantics of the "`if` construct" and the "`cond` construct", some terminology is defined.

Definition: A *subgoal* is a component of a specification that is an atomic subgoal (i.e., a predicate applied to terms), a construct, a conjunction of subgoals, or a disjunction of subgoals. ■

Definition: The *lexical boundary* of a clause is the head and body of the clause. The *lexical boundary* of a construct includes the construct key words (e.g., `if`, `then`, `else`, `cond`) as well as all of its test subgoals and branches. ■

Definition: A clause, an `if` construct, or a `cond` construct containing no constructs forms a single *region*. If a clause or construct contains another construct, the *region* of the outer construct or clause is its *lexical boundary* excluding the *lexical boundary* of the inner construct. *Regions* are disjoint and separated by constructs. The parent *region* of a *region* R is the enclosing *region* of *region* R. ■

Definition: The *region name* of the *region* of a clause is the functor of the clause's head followed by a unique clause number. The *region name* of the *region* of a construct is a concatenation `RN_CN#` where `RN` is the *region name* of the enclosing *region*, `CN` is `if` or `cond` (as the construct is an `if` construct or `cond` construct respectively), and `#` is a unique number for each *region name*. ■

Definition: The *variable set* of a *region* or a *lexical boundary* is the set of variables (Prolog variables) that occur in the *region* or the *lexical boundary*. ■

Definition: The *communication variable list* I(R) of a *region* R is a lexically sorted list of the variables obtained by the I function:

$I(R) = \phi$, if R has no parent (i.e., R is the region of a clause).

$I(R) = S(R) \cap (var\_set(parent\_region(R)) \cup I(parent\_region(R)))$

where $\cap$ and $\cup$ are the usual set operations, ϕ is the empty set, $var\_set(R)$ is the *variable set* of *region* R, $parent\_region(R)$ is the parent *region* of *region* R, and

$S(R)$ is the *variable set* of the lexical boundary corresponding to *region* R (i.e., the union of variables in R and all its descendent *regions*). Intuitively, the communication variable list of a region contains all variables that are referenced both inside and outside the region. ■

A body specification is a logic procedure written in a super set of Prolog. Two constructs are provided to enhance the readability of a body specification. An **if** construct has the following syntax and semantics:

```
Syntax:
  (if Test_subgoal(VAR1, ..., VARn, TV) then
      True_branch
  else
      False_branch)
```

where $n \geq 0$, TV is a logical variable whose domain of ground values is {**true, false**}; **Test_subgoal** is a literal with parameters **VAR1, ..., VARn**, and **TV**; and **True_branch** and **False_branch** are subgoals.

Declarative Semantics:

(Test_subgoal(VAR1, ..., VARn, true) ∧ True_branch)
∨ (Test_subgoal(VAR1, ..., VARn, false) ∧ False_branch)

Operational Semantics:

Call the **Test_subgoal**. If the variable **TV** matches **true**, then proceed with the **True_branch**; if **TV** matches **false**, then proceed with the **False_branch**. If the **Test_subgoal** fails or binds **TV** to something that will not match either **true** or **false**, then the entire **if** construct fails.

This is **not** the if-then-else provided by Prolog, which relies on negation as failure by proceeding with the **False_branch** if the **Test_subgoal** fails. If the **True_branch** is attempted and fails, the **False_branch** may still be attempted (if **TV** matches **false**). It is also possible to backtrack over solutions to **Test_subgoal**.

Note that it is possible that **TV** is left unbound, in which case either branch may be taken, as **TV** will successfully match either **true** or **false**, though the operational semantics specifies that the **True_branch** will be attempted first. Note also that the **Test_subgoal** is attempted first before either of the two branches are taken. When the **if** construct is the only component of a clause body, its surrounding parentheses can be omitted. If the **if** construct is used in a conjunction with other literals or constructs, the parentheses are necessary since the comma has higher priority than **then** and **else**.

A **cond** construct has the following syntax and semantics:

```
Syntax:
  (cond (Test_subgoals_1) => Branch_1 $
        (Test_subgoals_2) => Branch_2 $
        ...
        (Test_subgoals_n) => Branch_n)
```

where **Test_subgoals_i**, $1 \leq i \leq n$, are subgoals (for atomic subgoals, parentheses are not needed), **Branch_i**, $1 \leq i \leq n$, are subgoals.

Declarative Semantics:

(Test_subgoals_1 ∧ Branch_1)
∨ (Test_subgoals_2 ∧ Branch_2)
...
∨ (Test_subgoals_n ∧ Branch_n)

Operational Semantics:

Test_subgoals_i's are attempted in the order given in a **cond** language construct. If one of the **Test_subgoals_i** succeeds, the corresponding **Branch_i** is attempted. The **cond** construct fails only if every **Test_subgoals_i** – **Branch_i** combination fails.

When the **cond** construct is the only component of a clause body, its surrounding parentheses can be omitted. Backtracking is allowed to traverse different branches to find alternative solutions. Note that constructs can be nested to any finite level.

The program generation (or translation) module of the VSPEC project takes as input a specification and generates an internal representation of logic program (i.e., ILP). It is also possible to execute the specification with the interpreter accessing control information at runtime. The system translates a logic clause, which may or may not contain **if** and **cond** constructs, into a Horn clause or clauses without the constructs. Note that control information is also incorporated into the translation.

In the following example, A definition of **merge(L1, L2, ML)** and the clauses that result from the transformation of the **cond** and **if** constructs it contains are given. **merge(L1, L2, ML)** holds if **L1** and **L2** are sorted lists and **ML** is the sorted list containing all elements of **L1** and **L2**. The clause "**merge**" consists of three *regions*. Note that *regions* do not overlap each other so that **region1** contains only one literal. The *variable sets* of the *regions* in the example are:

```
region1: {L1, L2, ML}
region2: {L1, L2, ML, N, L1tail, M, L2tail}
region3: {N, M, Is_lt, L1tail, L2, MLtail,
          ML, L1, L2tail}
```

The *communication variable lists* of the *regions* in the example are:

```
region1: {}
region2: {L1, L2, ML}
region3: {L1, L1tail, L2,
          L2tail, M, ML, N}
```

The subgoal generated for the cond construct is

`merge_cond0(L1,L2,ML)`

which is built from the predicate name "merge", the construct name _cond, a unique number 0, and the *communication variable list* L1, L2, ML of region 2 (the region of the cond construct). Similarly, the if construct is replaced by a new subgoal

`merge_cond0_if1(L1,L2,L1tail,L2tail,M,ML,N).`

The example showing the body specification of a merge relation is

```
---------------------------------------------region1
|merge(L1, L2, ML) :-                                |
|-------------------------------------------region2|
||( cond L1 = [] => L2 = ML $                       ||
||       L2 = [] => L1 = ML $                       ||
||       (L1 = [N | L1tail], L2 = [M | L2tail]) =>||
||       -----------------------------------region3||
||       | (  if lt(N, M, Is_lt) then              |||
||       |       ( merge(L1tail, L2, MLtail),      |||
||       |         ML = [N | MLtail]  )            |||
||       |    else                                 |||
||       |       ( merge(L1, L2tail, MLtail),      |||
||       |         ML = [M | MLtail]  )  )  ).     |||
||       ------------------------------------------||
|--------------------------------------------------|
----------------------------------------------------
```

By applying a program generation algorithm, the following Prolog program is generated.

```
merge(L1,L2,ML) :- merge_cond0(L1,L2,ML).

merge_cond0(L1,L2,ML) :- L1=[], L2=ML.
merge_cond0(L1,L2,ML) :- L2=[], L1=ML.
merge_cond0(L1,L2,ML) :-
    (L1=[N|L1tail],L2=[M|L2tail]),
    merge_cond0_if1(L1,L2,L1tail,L2tail,M,ML,N).

merge_cond0_if1(L1,L2,L1tail,L2tail,M,ML,N) :-
    lt(N,M,true),
    merge(L1tail,L2,MLtail),
    ML=[N|MLtail].
merge_cond0_if1(L1,L2,L1tail,L2tail,M,ML,N) :-
    lt(N,M,false),
    merge(L1,L2tail,MLtail),
    ML=[M|MLtail].
```

Note that clauses of the same predicate name are placed together in the generated ILP. After the ILP is generated, optimization can be made to improve the runtime performance of the program. For example, the three clauses of the merge_cond0 predicate can be optimized to:

```
merge_cond0([], ML, ML).
merge_cond0(ML, [], ML).
merge_cond0([N|L1tail],[M|L2tail],ML) :-
    merge_cond0_if1([N|L1tail],[M|L2tail],
        L1tail,L2tail,M,ML,N).
```

4.2 Iterative Constructs

Failure-driven loops are claimed to be a bad programming style. One way of defining a failure-driven loop is to use a repeat/0 predicate with a Prolog cut and a fail. For instance, the repeat/0 predicate can be defined as:

```
repeat.
repeat :- repeat.
```

Subgoals in each iteration of a repeat loop are visited again at a new level of a search tree. The search tree becomes deeper and deeper as the iteration proceeds.

The continuation mechanism can be used in the implementation of structured iterative constructs. Since a continuation call performs a jump to a previously defined node in the search space, the depth of the search will not be increased.

Structured iterative constructs in logic programming can be designed using the continuation predicates. The syntax and semantics of a *while construct* are:

```
Syntax:
  while Test_predicates,
  Loop_goals,
  end_while
```

where Test_predicates is a subgoal or a conjunction, Loop_goals consists of Prolog predicate(s) or logic construct(s), and while and end_while are reserved words. Loop_goals is optional.

Operational Semantics:

The Test_predicates are called. If they succeed, the Loop_goals are solved and a jump is made to the Test_predicates. If the call to Test_predicates or Loop_goals fails, the iteration ends, and the computation continues from the continuation after the *while construct*.

Declarative Semantics:

Since a while construct cannot fail, and one can assume nothing about the success of the goals in its body, its declarative semantics is simply: *true*. This, less than satisfying, semantics highlights the fact that the while loop is a non-logical construct invented for procedural convenience.

For the implementation, a *while construct* can be translated statically. For instance,

```
predicate :-
  Goals,
  while Test_predicates,
  Loop_goals,
  end_while,
  Other_goals.
```

can be translated to

```
predicate :-
  Goals,
  assert_continuation(while#),
  ( Test_predicates,
    Loop_goals
  ; retract_continuation(while#)
  ), !,
  ( call_continuation(while#)
  ; true
  ), !,
  Other_goals.
```

A continuation associated with the while construct (where # is a unique number for the while construct) is asserted before the loop starts. If the `Test_predicates` and `Loop_goals` succeed, a continuation call is made to the beginning of the loop. If the call to `Test_predicates` fails or `Loop_goals` fails, the unique continuation associated with the while construct is retracted, and the computation proceeds from `Other_goals`.

One may write a while loop in ordinary Prolog as follows:

```
while(T, L) :-
  call(T),
  call(L),
  while(T, L).
while(_, _).
```

Most Prolog interpreters support tail recursion elimination. However, not all of them use a fixed amount of stack space for executing a tail recursive predicate. In that case, the while loop implemented above will cost lots of memory due to the recursive call to `while`. Using a continuation jump, one can save memory. Note that both the while loop implemented above and the one introduced in this thesis need to rely on using assert/1 and retract/1 Prolog predicates (or other non-logical goals, such as `read/1`) in `T` or `L` in order to terminate the loop.

Similarly, a *do-until construct* can be designed. A *do-until construct* has the following syntax:

```
Syntax:
  do,
  Loop_goals,
  until Test_predicates
```

where `Test_predicates` is a subgoal or a conjunction, `Loop_goals` is a Prolog predicate(s) or logic construct(s), and `do` and `until` are reserved words. `Loop_goals` is optional.

The operational and declarative semantics of a *do-until construct* are equivalent to the following *while construct*:

```
while
  Loop_goals,
  not(Test_predicates),
  end_while
```

For the implementation, a *do-until construct* can be translated. For instance,

```
predicate :-
  Goals,
  do,
  Loop_goals,
  until Test_predicates,
  Other_goals.
```

can be translated to

```
predicate :-
  Goals,
  assert_continuation(do#),
  ( Loop_goals,
    \+ Test_predicates,
    call_continuation(do#)
  ; retract_continuation(do#)
  ), !,
  Other_goals.
```

A continuation associated with a *do-until construct* is asserted before the loop starts. The `Loop_goals` are solved and the `Test_predicates` are called. If the `Test_predicates` fail, the loop is started again via a continuation jump. If the `Test_predicates` succeed, the computation proceeds from `Other_goals`.

One way to implement a do loop in Prolog is

```
do(T, L) :-
  call(L),
  call(not(T)),
  do(T, L).
do(_, _).
```

The drawback of using this do loop is similar to the one of an ordinary while loop written in Prolog as discussed earlier.

5 Conclusions

In this paper, we presented a visualization environment that allows logic programming engineers to design their logic specification programs. The system (i.e., VSPEC) is a continuous work based on the SPEC project earlier developed by the authors. The visualization tools support a hyper-text like editor allows the user to navigate different pieces of a specification program via hyper-links. A declarative specification browser is also introduced, which uses an "and-or" tree structure to display the semantics of a logic specification program. These visualization tools, integrated with a revised version of SPEC running under the MS Windows, help logic programming engineers in designing their specifications. We believe that, the proposed visualization tools will make a contribution to the study of Software Engineering in Logic Programming.

References

[1] M. Bruynooghe, D. De Schreye, and B. Krekels. "Compiling Control". In *Proceedings 1986 Symposium on Logic Programming*, pages 70–77, 1986.

[2] K. L. Clark, F. G. McCabe, and S. Gregory. "IC-PROLOG Language Features". In K. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 253–266. Academic Press, 1982.

[3] K. L. Clark and S. Gregory. "Notes on Systems Programming in Parlog". In *Fifth Generation Computer Systems 1984*. North Holland, 1984.

[4] Ruth E. Davis and Timothy K. Shih. "A CASE for Logic Programming". In *Proceedings of The Tenth International Conference of The Chilean Computer Science Society*, pages 73–84, 1990.

[5] H. Gallaire and C. Lasserre. "Metalevel Control for Logic Programs". In K. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 173–185. Academic Press, 1982.

[6] M. Meier and H. Grant. "SEPIA Programming Environment". In *Proceedings of the NACLP '89 Workshop on Logic Programming Environments: The Next Generation*, pages 103–114, 1989.

[7] P. Mello, A. Natali, and C. Ruggieri. "Logic Programming in a Software Engineering Perspective". In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference 1989*, pages 441–458. MIT Press, 1989.

[8] L. Naish. "Automating Control for Logic Programs". *The Journal of Logic Programming*, 2(3):167–183, 1985.

[9] L. Naish. *Negation and Control in Prolog*, volume 238 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

[10] L. M. Pereira. "Logic Control With Logic". In *Proceedings of the First International Logic Programming Conference*, pages 9–18, 1982.

[11] A. Porto. "Epilog: A Language for Extended Programming in Logic". In *Proceedings of the First International Logic Programming Conference*, pages 31–37, 1982.

[12] E. Y. Shapiro. "A Subset of Concurrent Prolog and Its Interpreter". Technical report, Weizmann Institute of Science, 1983.

[13] E. Shapiro, editor. *Concurrent Prolog*. MIT Press, 1987.

[14] Timothy K. Shih and Ruth Davis. "Intelligent Backtracking and Control Based on a Deduction Status Representation in Logic Programming". In *Proceedings of the Second Golden West International Conference*, 1992.

[15] Timothy K. Shih and Ruth Davis. "Program Generation and Controls in a Specification Processing Environment". In *Proceedings of the 1992 International Computer Symposium*, 1992.

[16] Timothy K. Shih, Ruth Davis, and Rob Langsner. "A Specification Processing Environment for Making Well Engineered Logic Programs". In *Proceedings of the Second Golden West International Conference*, 1992.

[17] Timothy K. Shih, Ruth Davis, and Fuyau Lin. "Coping with Failure: Disciplined Exceptions in Logic Programming". In *Proceedings of The 1992 ALP UK Logic Programming Conference*, 1992.

[18] Timothy K. Shih. "Continuation Semantics of Runnable Specifications in Logic Programming". Ph.D. Thesis, Department of Computer Engineering, Santa Clara University, 1993.

[19] Timothy K. Shih, and Ruth Davis. "SPEC: Specification Processing Environment with Controls". to be published in *Journal of Information Science and Engineering*, 1995.

[20] Timothy K. Shih, and Fuyau Lin. "Continuation Semantics of Logic Programs with Exception Handling". to be published in the *Computer and Artificial Intelligence Journal*, 1995.