Comparative Study of the Inference Mechanisms in PROLOG and SPIDER

Emilia Golemanova [1], Tzanko Golemanov [1], Kostadin Kratchanov [2]

[1]*Department of Computer Systems and Technologies, University of Ruse, 8 Studentska str., Ruse, Bulgaria*
[2]*Independent Scholar, P.O. Box 15, Sofia 1766, Bulgaria*

Abstract **– Control Network Programming (CNP) is a graphical nonprocedural programming style whose built-in inference engine (interpreter) is based on search in a recursive network. This paper is the third in a series of reports that share a common objective – comparison between the CNP language SPIDER and the logic programming language PROLOG. The focus here is on the comparative investigation of their interpreters, presented in a generic formal frame – reduction of goals. As a result of juxtaposing their pseudo-codes the advantages of SPIDER are outlined.**

Keywords **– Control Network Programming, programming paradigms, programming languages, inference mechanism, computation control.**

1. Introduction

Logic Programming (LP) is an inference system, based on the resolution refutation theorem proving in a subset of Fist-Order Predicate Calculus, called Horn clauses [1]. In mathematical terms, the two types of clauses – rules and facts - are axioms and the third clause type – the question - is the theorem, which the abstract (i.e. nondeterministic) interpreter of logic programs tries to prove. As the inference mechanism (computation) is with elements of nondeterminism, it is actually a search process.

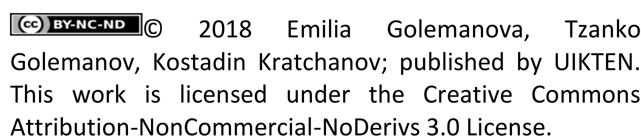
The nondeterminism is a major feature of Control Network Programming (CNP), too. CNP is a style of high-level programming and will be discussed in the next section. Computation in CNP is also a search process. Main languages, representative for LP and CNP, respectively, are PROLOG and SPIDER. The search strategy in both languages can be described as a generalized version of depth-first search with backtracking. The two programming styles, although very different, have other similarities as well – they are declarative in nature and possess various tools for programmer's control of the search.

It is worthwhile to compare these similarities seeking for correspondences and possibilities to simulate the search in LP by the search in CNP. It would be also useful to identify differences, marking the strengths of each of the paradigms and to design effective solutions that can be 'induced' from one of the languages into the other in order to improve the latter.

DOI: 10.18421/TEM74-30
https://dx.doi.org/10.18421/TEM74-30

Corresponding author: Emilia Golemanova,
Department of Computer Systems and Technologies, University of Ruse, 8 Studentska str., Ruse, Bulgaria
Email: egolemanova@uni-ruse.bg

Received: 04 October 2018.
Accepted: 12 November 2018.
Published: 26 November 2018.


© 2018 Emilia Golemanova, Tzanko Golemanov, Kostadin Kratchanov; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License.

The article is published with Open Access at www.temjournal.com

2. Control Network Programming

The philosophy and implementation of CNP, as well as its relationship to other programming paradigms were concisely described in [2]. Description in more detail including the theoretical model of computation (under the name *Cinnamon Programming*) and solutions to exemplary problems using CNP are presented in [3][4][5]. This section is a very short introduction to the CNP main features and terminology.

CNP has been created in response to the requirement the programming language to be natural, i.e. similar to the manner in which people think of problems and the style in which they specify them informally.

In many cases, the innate structure of a problem to handle and consequently the human form for

describing it, is not linear, but is a tree, a graph (network), or a recursive set of networks. It would be a great advantage if there was no need for the programmer to try to translate this inherently graph-like description into a much more complicated and difficult to understand sequential algorithmic model of this description. With this observation in mind, the program in CNP is a graphical specification of that declarative representation, which is more natural and intuitive than the declarative representation implemented by a programming text, used in logic programming languages like PROLOG and constraint programming languages like COMET, for example. Being a universal programming paradigm, CNP can be used for implementing not just declarative but procedural problem solutions as well. Presenting a procedural solution in a graphical form (similar to an activity diagram or a flowchart) makes the program control explicit and promotes easier understanding, creating, modifying, or verifying the algorithm. Last but not least, as the CNP involves elements of the procedural programming paradigm, it is convenient for the imperative language programmers who have to implement algorithms which involve nondeterminism or randomness, or are based on search, even heuristic search.

Programming through control networks, or **Control Network Programming**, or just **CNP,** is a style of programming in which the program can be visualized as a set of graphs, called a **Control Network (CN)**. Such a set of graphs is also known as a recursive network because the subnets can call each other or themselves. The individual graphs are called *subnets*. The CNP language we use is named **SPIDER**. The CN of a technical example is depicted in Figure 1.a and Figure 1.b. The screenshots are from **SpiderCNP** – the best currently used integrated programming environment for CNP, powerful and fully fledged with an embedded advanced graphic editor and debugger.

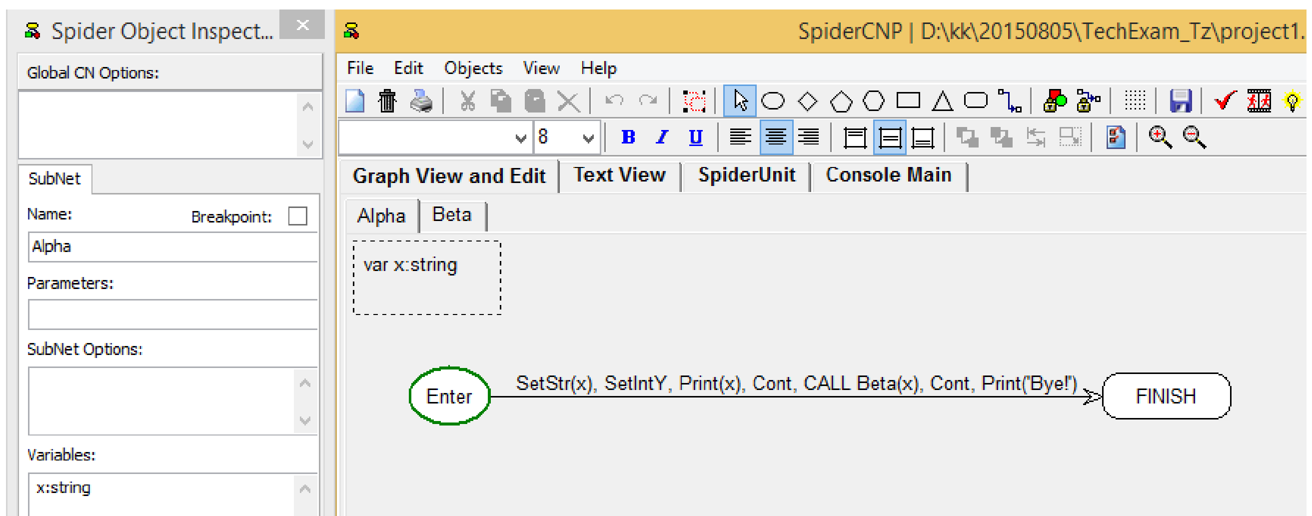


Figure 1a: A CN program – Technical example – the 'Alpha' main subnet

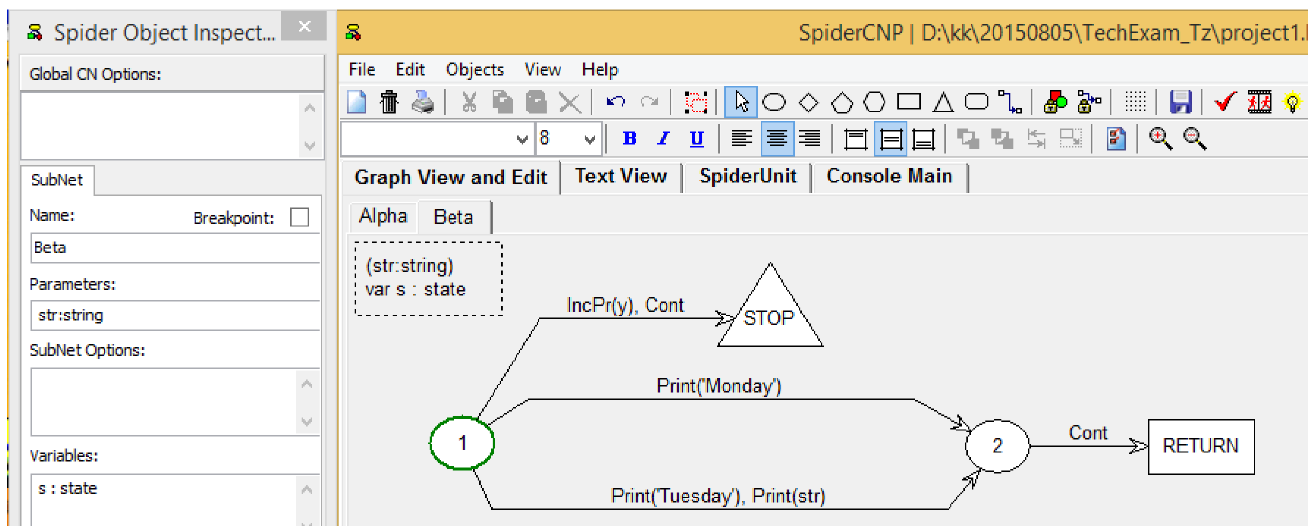


Figure 1.b: A CN program – Technical example – the 'Beta' subnet

Each subnet, one of which is the main subnet, consists of nodes (called also *states*), and arrows between nodes. The arrow is labelled with a sequence of *primitives*, which are elementary actions, implemented as user-defined functions in some procedural programming language. Both subnets and primitives can have parameters and local variables. The CN may be nondeterministic - from a given node, there is in general a set of various paths that may be explored while searching for a solution. The inference engine (interpreter) "executes" the CN by implementing a backtracking-like search strategy for traversing the CN, starting from the initial node of the main subnet and executing the primitives along the way. This process will successfully finish when the control reaches a system node FINISH.

Each primitive can be successfully or unsuccessfully executed. In the latter case the control will "backtrack". During this process the primitives are executed backwards. The system state STOP denotes "local failure" and also triggers backward execution. Subnet calls (invocations) are indicated by the system primitive CALL. Successful completion of a subnet is represented by the system state RETURN in which case the control is passed back to the point where the subnet was called.

The computation/search strategy of SPIDER, explained in detail in [6], is an extended version of backtracking; one of the major enhancements is the possibility to backtrack through a subnet, the other is introducing forward and backward execution of a primitive. The latter one is conducted during backward execution in order to restore the state of the data. Figure 2.a and Figure 2.b depict two possible executions (successful and unsuccessful) of the technical example in Figures 1.a and 1.b.

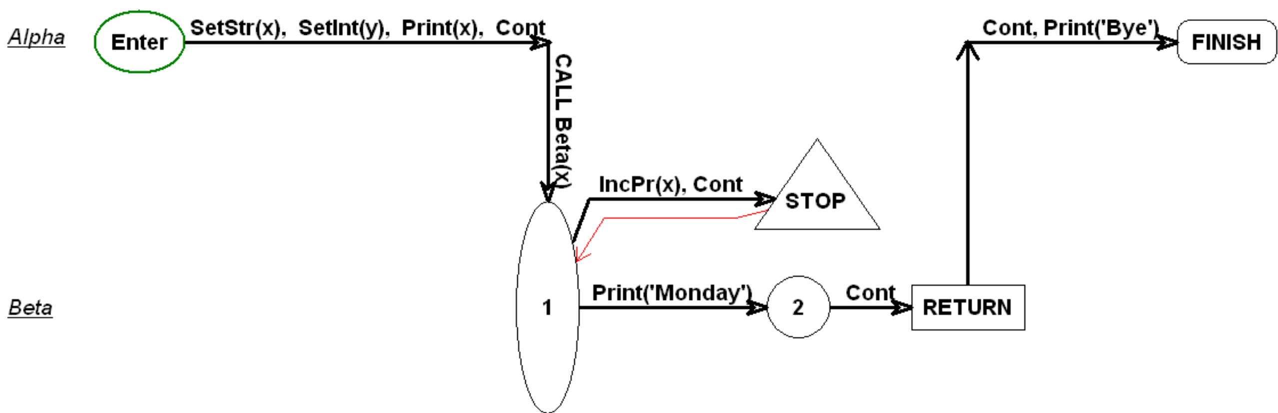


Figure 2a: Successful execution of the Technical example

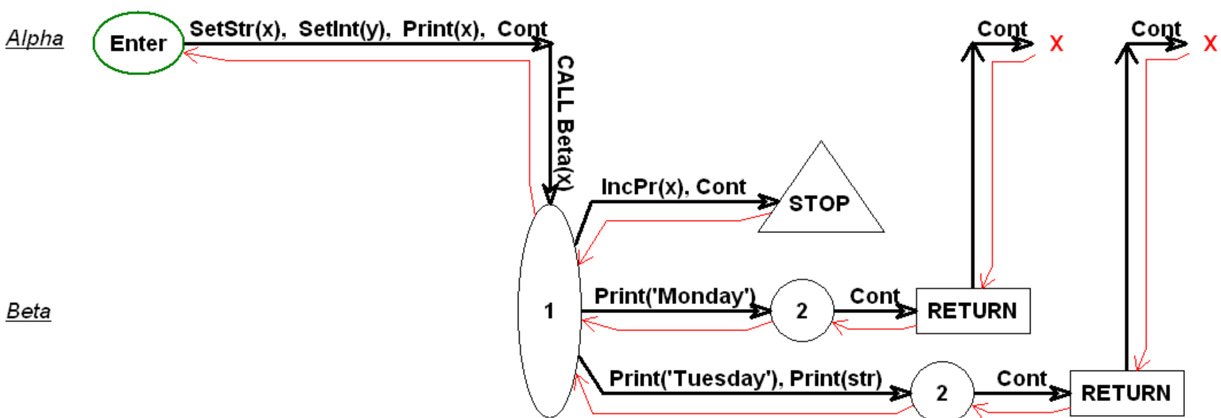


Figure 2b: Unsuccessful execution of the Technical example

As a matter of fact, in the *SpiderCNP* no interpreter actually exists. The compiler uses an approach similar to the recursive decent technique well known in compilation. The interpreter produces an "intermediate program" (in the imperative language of the primitives) which corresponds to the CN, at the same time also embodying the algorithm for interpreting it. The advantages of this approach are described in [2][7][8].

In addition to the primary search mechanism, SPIDER supports powerful built-in tools (named *control states* and *system options*) for user control of the computation [9][10].

3. Interpreter for Logic programs

The computation of logic programs performed by an abstract, i.e. nondeterministic, interpreter is carried out via a goal reduction. The inference mechanism of PROLOG is derived from the abstract interpreter of logic programs by selecting the leftmost goal from the resolvent, sequential search for a unifiable clause and depth-first search with backtracking [11]. Thus, the interpreter of PROLOG programs is a concrete realization of the abstract interpreter, and can be represented by the Boolean function ***Search*** (Figure 3.) for execution of a list of goals (a modified version of the procedure *execute* from Figure 2.11, [12]). The mechanism of unification and search for more solutions are not explicitly described in the presented pseudo-code.

Manipulation of the list *GoalList*, representing the resolvent, is accomplished by the standard list operations *empty*, *head*, *tail* and *append*. When a recursive call to ***Search*** ends with a failure, the execution continues with a backtracking to the previous recursion level and testing the next unifiable clause. Thereby PROLOG systematically examines all possible alternative paths in a search tree.

```
Search(GoalList):boolean;
Var     Goal: goal;
        OtherGoals, SubGoals, NewGoals: list of
        goals;
begin
 if empty(GoalList) then return true
 else
begin
   Goal:=head(GoalList);
   OtherGoals:=tail(GoalList);
   while Goal has unifiable clauses
    begin
     SubGoals:=body of unifying clause;
     NewGoals:=append(SubGoals, OtherGoals);
     if Search(NewGoals) then return true;
    end;
   return false;
end;
end;
```

Figure 3: An interpreter for PROLOG programs

4. Interpreter for CN programs

The formal semantics for *Core CNP* (named *Cinnamon Programming)*, was presented in [3], together with the activity diagram of an 'interpreter'. Here, a different approach is taken where the 'computation' in CNP is specified through an interpreter similar to the one for logic programming described in the previous section. This way, presenting both interpreters in a generic frame – reduction of goals, allows their more formal juxtaposing and finding the correspondences.

The computation in CNP can also be presented as a goal reduction. The initial goal is the "computation / execution" of the initial state of the main subnet. It is reduced to a sequence of subgoals, which consists of the primitives of the outgoing arrow and the state to which it points. It follows that a "goal" in CNP is either a primitive (user-defined or the system primitive *CALL*), or a state (ordinary or a system one). Actually, both kinds of primitives can be considered as special cases of state, i.e.:

- The *user-defined primitive,* which, as mentioned before, can have arguments, could be considered as a "state with parameters" and with just one outgoing arrow describing a deterministic computation.
- The *system primitive CALL* for a subnet call could be interpreted as a "state with parameters", too, because:
  - CALL may be considered as a transition to a state (default or explicitly specified) of the invoked subnet;
  - subnets can have parameters and local variables.

The basic behavior of the CNP interpreter (i.e., without the computation control) is given in Figure 4.

Each iteration of the while-loop at the reduction of a *goal-state* corresponds to the execution of one outgoing arrow. In general, the arrows are tried in the order of their definition. The computation continues with a new sequence of goals (*NewGoals*), obtained by replacing the *goal-state* with the subgoals of the selected arrow.

The reduction of a *goal-CALL* is the replacement of *CALL* by the state, corresponding to the entry point of the invoked subnet.

Reducing the *goal-primitive* means its execution, which can be successful or unsuccessful. If it is successful, the computation continues with the next goals on the current arrow.

RETURN is always a successful goal and the execution continues with *OtherGoals*, unlike the system state *STOP* which is always an unsuccessful goal (local failure). The computation ends when the system state *FINISH* is reached, which is interpreted

as a global success. The return of value "true" on the current recursion level leads to a successful termination to all the above recursive levels of ***Search***. Unsuccessful execution of the CNP-program, meaning there is no solution to the problem, is indicated by the value "false" of ***Search***.

```
Search(GoalList):boolean;
Var      Goal: goal
         OtherGoals, SubGoals, NewGoals: list of goals

begin
 Goal:=head(GoalList);
 OtherGoals:=tail(GoalList);
 case type of Goal
         state:
                  begin
                   while Goal has unexplored arrows
                    begin
                     SubGoals:=next arrow;
                     NewGoals:=append(SubGoals, OtherGoals);
                     if Search(NewGoals) then return true;
                    end;
                   return false;
                  end;
         CALL v:
                  begin
                   NewGoals:=append(v, OtherGoals);
                   return Search(NewGoals);
                  end;
         primitive:       return (execute(Goal) and Search(OtherGoals));
         RETURN:          return Search(OtherGoals);
         STOP:            return false;
         FINISH:          return true;
 end;             // case
end;              // Search
```

Figure 4: An interpreter for CN programs

5. Interpreter for PROLOG programs vs. Interpreter for SPIDER programs

5.1. Similarities

Juxtaposing the pseudo-codes in Figure 3. and Figure 4., i.e., comparing the inference mechanisms of PROLOG and SPIDER, leads to identifying the following **correspondences**:

- The code for *goal* reduction in PROLOG matches with the code, reducing a *state* in SPIDER;
- A *clause* in PROLOG corresponds to an *arrow* in SPIDER.

However, given that the goals in PROLOG are calls to predicates, which are productions with arguments [13], a *goal* in PROLOG corresponds not to an ordinary state but to a "*state with parameters*" in SPIDER. As already mentioned above, both the user-defined primitives and the system primitive *CALL* could be interpreted as "states with parameters". But as the goals in PROLOG can be re-satisfied, i.e. the predicates specify rather nondeterministic than deterministic calculations, a *goal* in PROLOG corresponds not to a user-defined primitive but to the *system primitive CALL* in SPIDER, indicating the invocation of a subnet with possibly more than one solution.

Thus, **a predicate in PROLOG could be simulated in SPIDER by a subnet consisting of just two states - an initial state and a RETURN state, and an arrow between these two nodes for every clause of the predicate. The arrow is labeled with the system primitive CALL for each goal in the body of the corresponding clause.**

As an illustration of this rule, Figure 5. shows the CNP model of the predicate *C* of the following form: (a modified version from [12], Sec. 5.1):

C:-P, Q, R, S, T, U.
C:-V.

where *P, Q, ..., V* are denotations of goals.

SUB **C**;

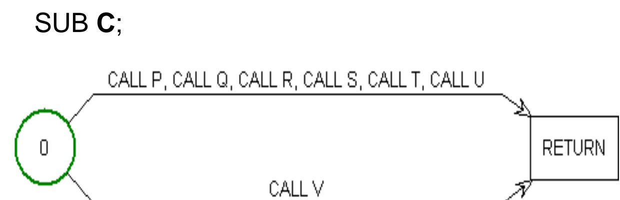


Figure 5: A CNP model of Prolog predicate C

To clarify this rule let us apply it to a more meaningful predicate [13], Sec.2.6:

lucky(Y):-generous(X), likes(X, Y).
lucky(Y):-content(Y).

Figure 6. gives the corresponding SPIDER program (The primitive *Free* is an auxiliary one and it is used for simulating of a free variable. In the presented example the variable X is instantiated with an empty string):

SUB Lucky

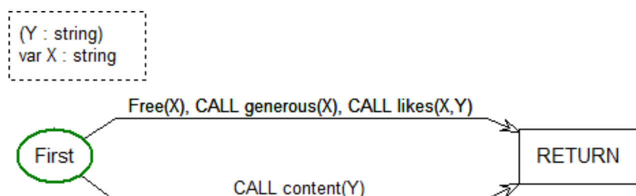


Figure 6: A CNP model of a predicate ***lucky***

In addition to identifying SPIDER correspondences of PROLOG fundamental concepts (predicate - subnet, clause - arrow, goal - CALL), this rule describes a universal way to model in SPIDER a program written in core PROLOG. A PROLOG knowledge base can be simulated in SPIDER by a collection of rather trivial subnets consisting of just an initial and a RETURN state and arrows between them. A SPIDER control network, in the general case, can have a much more complex structure of subnets.

In addition, such a simulation shows by transitivity the Turing-completeness of SPIDER (PROLOG is a Turing complete language). Other proofs for SPIDER Turing-completeness are made in [3][14][15].

As a side effect of this simulation (more precisely, the simulation of the computation control in PROLOG [16]) we realized that a SPIDER system option *[BACKTRANKING=No]* has to be modified. It would not only allow us to more closely model the behavior of *cut* in PROLOG but, more importantly, it will be more natural and will have certain advantages.

5.2. Differences

In addition to the discussed above correspondences between the interpreters of PROLOG and SPIDER, the following differences can be observed:

- The indicator of a successful execution of a logic program is reaching an empty resolvent, while in SPIDER such an indicator is arriving at a system state *FINISH*, which could be used in any subnet. This allows a termination of the computation in a subnet, which is not the main one. The situation is equivalent to a nonempty resolvent in PROLOG. This leads to optimization of the computation process and flexibility in application development.
- In SPIDER there are several kinds of "goals for reduction" – a user-defined primitive, a system primitive *CALL*, an ordinary state and system states (*FINISH*, *RETURN* and *STOP*). This allows for a more detailed specification of the problem to be solved and, as already mentioned, for subnets with a more complex structure than that in Figure 5.
- In SPIDER there is a flexible data management at different program levels. SPIDER supports both global data and local data (subnet parameters and local variables). PROLOG does not support global data. Only the mechanism for parameter passing is used. The global structures are modelled by meta-predicates for database manipulation (assert, retract) [1].
- In SPIDER, "behind" the while-loop in Figure 4., a far more complex logic (in the case of processing a state) is actually "hidden". For example, SPIDER has a powerful collection of built-in tools (*control states* and *system options*) for:
  - preventing looping via setting a maximum depth of iteration or recursion;
  - switching off backtracking at various levels - the whole control network, a subnet, a state, or an arrow;
  - evaluation and reordering the alternatives (arrows);
  - limiting the number of attempted alternatives;
  - selection of an alternative or a group of alternatives;

  Such possibilities through which the user can control the computation/search process are discussed in [9][10][5].
  In PROLOG, the only control of the search mechanism is achieved through "emergency exit from the while-loop" in Figure 4. (when the primitive *cut* is found).

In our opinion, all the above differences are advantages of SPIDER over PROLOG.

For the sake of demonstration, we would like to sketch the solution of two simple problems and their solutions in both languages, which illustrate the advantage of SPIDER in preventing looping and in declarative implementation of heuristic and stochastic algorithms.

Preventing looping

Infinite loops are not unusual in PROLOG programming [12]. Consider the following very simple example – Map traversal problem (MTP) [17]. The exemplary map is shown in Figure 7. Suppose we just want to check if there is any acyclic path between a given city and the city *F*.

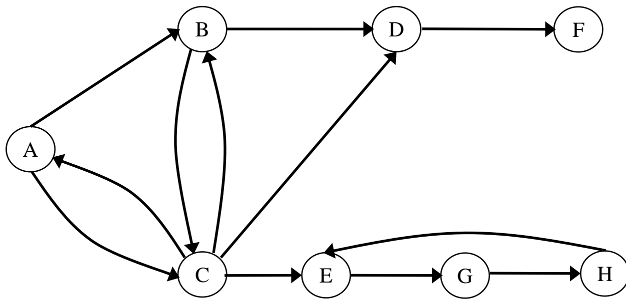


Figure 7: Exemplary map

As the PROLOG has a built-in search engine, the straight solution to the problem is just to define the map and leave the interpreter to find the path in the graph. The PROLOG program (Figure 8.) would be pure declarative:

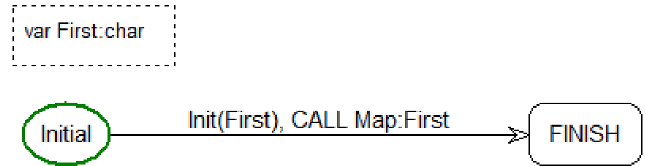
a :- b.	**a** :- c.	
b :- c.	**b** :- d.	
c :- a.	**c** :- b.	**c** :- e.
d :- f.		
e :- g.		
g :- h.		
h :- e.		
f.		

Figure 8: PROLOG solution for MTP

But the problem with this program is that being declaratively correct, it is procedurally incorrect because it is not able to produce an answer to question **?-a.**, for example. The reason is that PROLOG will enter in an infinite loop. The remedy for this problem is either the programmer to reorder the clauses, or to develop a loops detection mechanism [1][12][11].

In SPIDER such a loops detection mechanism is built-in and can be forced through three system options: LOOPS, ONEVIST, and RECURSION. The SPIDER solution to the presented MTP is shown in Figure 9., where the CN "copies" the map from Figure 7. The setting [LOOPS=0] prevents SPIDER from looping and ensures that only acyclic paths will be found.

MAIN MapTraversal;



SUB Map;

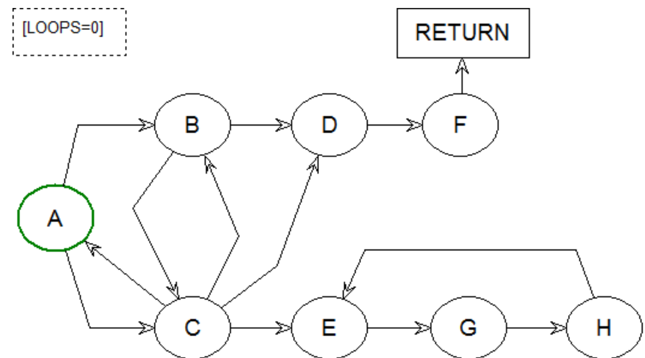
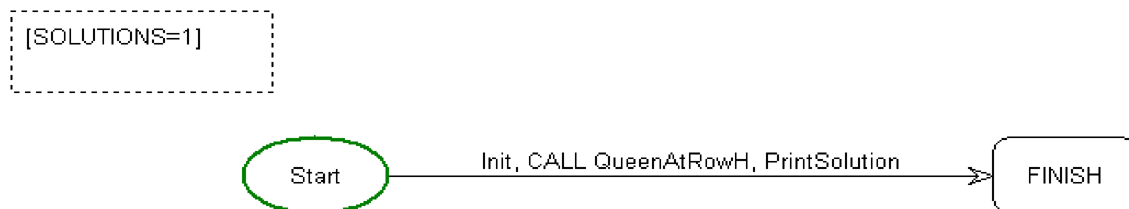


Figure 9: SPIDER solution for MTP

Implementation of heuristic and stochastic algorithms

Basically, the SPIDER programmer can do anything they want with the arrows of the current state, in contrast to the PROLOG programmer, who has no built-in means for evaluation and rearrangement of the clauses of the currently executed predicate. The built-in eleven system options and three control states in SPIDER [9][10] allow for a convenient user control of the inference strategy, or even for dynamic changes in the control that can be used for declarative implementation of advanced search strategies, including heuristics. As an illustration an example of their usage for implementing the Forward Checking algorithm (improved backtracking) with MRV, degree and LCV heuristics [18] on the 8-queens problem is given in Figure 10.

The 8-queens problem is a representative example of the Constraint Satisfaction Problems (CSPs), i.e. problems, formally described by a set of variables, a set of domains (values) and a set of constraints. The classic approach to solve CSPs is to use a backtracking search algorithm that picks one variable at a time and chooses a value for this variable. In the presented SPIDER implementation these two choice points with incorporated heuristics are modelled by system control states of type RANGE (namely *Rows*, *Cols*) in the recursive subnet *QueenAtRowH*. The implemented heuristics are the so called general-purpose heuristics - the two popular variable ordering heuristics - Minimum Remaining Values (MRV) and degree heuristic, and a value ordering heuristic -



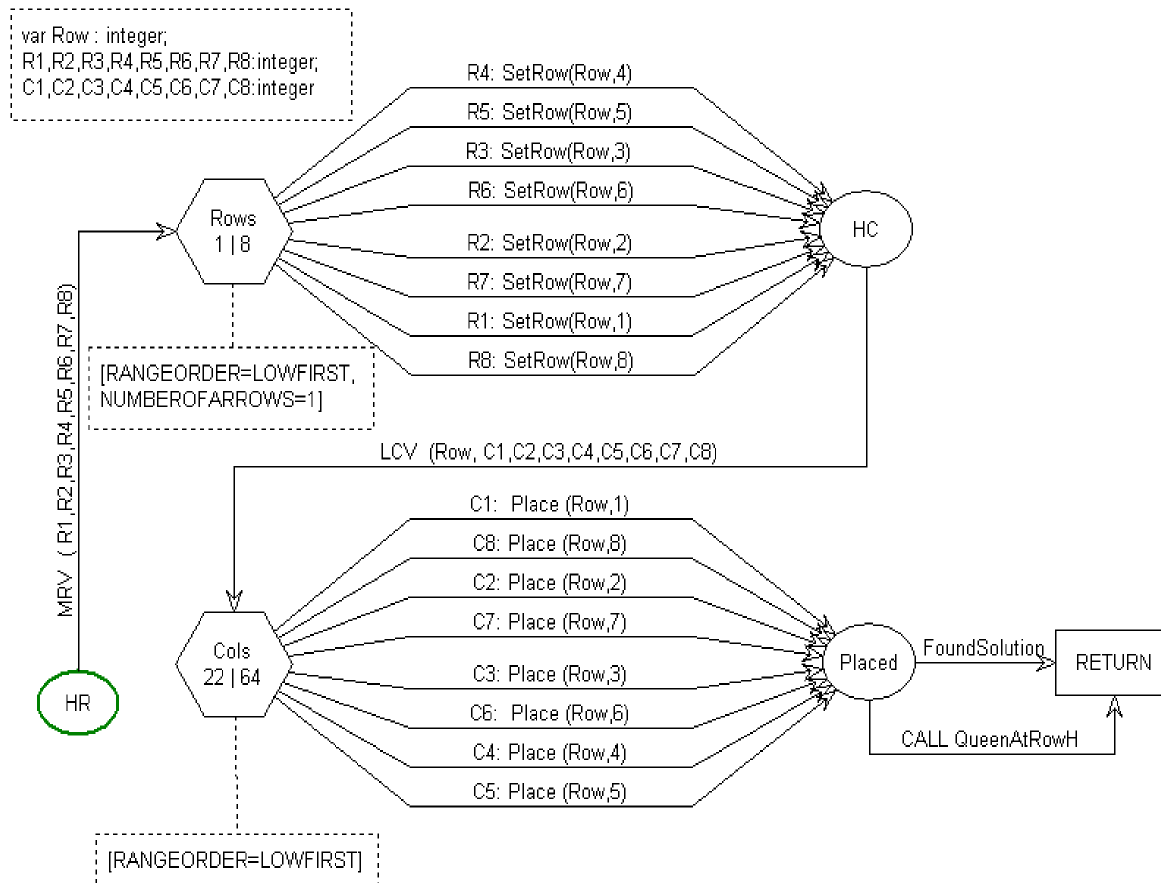


Figure 10: SPIDER implementation of the general-purpose heuristics for the 8-queens problem

Least-Constraining-Value (LCV). The primitive *MRV* and primitive *LCV* calculate the heuristic evaluations R_i and C_i, i∈{1, 2, …, 8} of all the 8 variables and all values of the already chosen variable, respectively, which are used as evaluations of the outgoing arrows from *Rows* and *Cols*. States *Rows* and *Cols* are RANGE type control states with low selector and high selector, determining which emanating arrows will be attempted - only those whose evaluations are in the specified range, others will be cut off. The “survived” emanating arrows will be attempted in ascending order of their evaluations due to the system option *[RANGEORDER=LOWFIRST]*. The other system option *[NUMBEROFARROWS=1]* is used because CSPs are commutative, i.e. it’s only needed to consider assignments to a single variable at each step [18].

Based on this solution it is easy to simulate other strategies (simple and stochastic variant of Forward Checking) which together with the above implementation are presented in more detail in [19]. Briefly, the SPIDER implementation of stochastic variable and value choice could be easily achieved if we replace the option *[RANGEORDER=LOWFIRST]* with *[RANGEORDER=RANDOM]* for both *Rows* and *Cols* control states. Correspondingly, a simple Forward Checking algorithm (backtracking with incorporated look-ahead technique) could be simulated in CNP using the solution from Figure 10. by deleting the rearranging arrows option RANGEORDER and by setting the outgoing arrows

from the control states *Rows* and *Cols* in a numerical order, i.e. from 1 to 8.

The corresponding PROLOG implementations of heuristic searches would be rather procedural than declarative, in contrast to the SPIDER implementation shown above, due to the fact that the PROLOG programmer has no built-in tools for dynamic evaluation and arrangement of the clauses of the currently executed predicate.

6. Conclusions

CNP addresses the long-standing "holy grail" of programming: removing the "representational gap", making a program look like a description by a domain expert. This programming paradigm is certainly not the first one that addresses this goal. Logic programming is one of the most famous declarative paradigms. The main novelty of the declarative approach of CNP/SPIDER is its emphasis on:

- bidirectional complementation and integration with the imperative programming technique. Most declarative programming paradigms, including Logic Programming, aim at replacing the more traditional imperative methodologies with something completely, or at least very different. Instead of suggesting a new and unknown world, we are simply proposing an extension to the current-day most well-known world of professional programming;
- graphical representation of the program – the program has explicit structure as a set of graphs;
- declarative user control – the programmer is provided with powerful set of built-in tools for computation control, which are also “visually visible”. They allow for the elimination of certain algorithmic issues (e.g., avoiding loops, switching off backtracking, restricting recursion), and, more importantly, easy and natural implementation of heuristic and randomized algorithms.

The strengths of the declarative approach of PROLOG, for its part, are:

- ability to manipulate symbolic data;
- it works well for problems where a knowledge base forms an important part of the solution, especially when it is suited to be encoded as logical statements;
- unification engine.

The interpreter (inference engine) in SPIDER implements an extended specific version of backtracking for search (inference / computation) in a recursive network. As the inference mechanism of PROLOG is based on search with backtracking, too, but in a set of clauses, it is worthwhile to explore the differences and to compare the features the two approaches offer. To do that in a formal way both algorithms were presented in a generic form – reduction of goals.

As a result of juxtaposing their pseudo-codes the advantages of SPIDER were outlined. They concern mainly the more flexible control mechanism of the interpreter.

The second main result of the comparison of the computation features of SPIDER and PROLOG is that the correspondences between the main concepts in both languages are outlined and the methodology for simulating PROLOG programs in SPIDER is developed [20][16]. The idea is to show that in principle this simulation is possible and to compare the resulting CNP model of a PROLOG program with a CN program in its general appearance. As a side effect of this simulation (more precisely, the simulation of the behavior of *cut* in PROLOG]), a modified version of the SPIDER system option *[BACKTRANKING=No]* was developed [16]. On the other hand, a good solution that can be ‘induced’ from SPIDER into PROLOG is enhancing the declarative nature of PROLOG with additional meta-programming features such as dynamic evaluation, reordering and selection of clauses. An additional result of that simulation was the demonstration by the transitivity of the Turing completeness of SPIDER.

Acknowledgements

The study was supported by a contract of Angel Kanchev University of Rousse, BG05M2OP001-2.009-0011-C01, Support for the Development of Human Resources for Research and Innovation at Ruse University Angel Kanchev with the support of the Operational Program Science and Intelligent Growth Education 2014-2020 funded by the European Social Fund of the European Union.

References

[1] Luger, G. F. (2005). *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education.

[2] Kratchanov, K., Golemanov, T., & Golemanova, E. (2007). Control Network Programming. *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*, Melbourne, Australia, 1012–1018.

[3] Kratchanov, K. (2017). Syntax and semantics for cinnamon programming. *Int. J. of Comp. Sci. and Information Technology*, *9*(5), 127-150.

[4] Kratchanov, K., Golemanova, E., & Golemanov, T. (2008, May). Control Network Programming Illustrated: Solving Problems with Inherent Graph-Like Representation. In *Seventh IEEE/ACIS International Conference on Computer and Information Science* (pp. 453-459). IEEE.

[5] Kratchanov, K., Golemanova, E., Golemanov, T., & Gökçen, Y. (2012). Implementing Search Strategies in Winspider II: Declarative, Procedural, and Hybrid Approaches. *Knowledge-Based Automated Software Engineering, Cambridge Scholars Publ*, 115-135.

[6] Kratchanov, K., Golemanova, E., & Golemanov, T. (2009, February). Control network programs and their execution. In *Proc. 8th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)* (pp. 417-422).

[7] Kratchanov, K., Golemanova E., Golemanov, T., & Gökçen, Y. (2012). Implementing Search Strategies in Winspider I: Introduction to Control Network Programming and Search. *Knowledge-Based Automated Software Engineering*, I. Stanev and G. K., Eds. Cambridge Scholars Publishing, 87–113.

[8] Kratchanov, K., Yüksel, B., Golemanov, T., & Golemanova E. (2014). Control Network Programming Development Environments, *WSEAS Transactions on Computers*, 13, 645–659.

[9] Kratchanov, K., Golemanov, T., & Golemanova, E. (2009). Control Network Programming: Static Search Control With System Options. *8th WSEAS International Conference on Artificial Intelligence, Knowledge Engineering and Data Bases (AIKED 2009)*, Cambridge, UK, 423–428.

[10] Kratchanov, K., Golemanov, T., Golemanova, E., & Ercan, T. (2010). Control Network Programming with SPIDER: Dynamic Search Control. *14th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2010)*, Cardiff, UK, 253–262.

[11] Sterling, L. & Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques (Logic Programming)*, Second Edi., Cambridge, Massachusetts: The MIT Press.

[12] Bratko, I. *Prolog Programming for Artificial Intelligence (*2011), 4th ed. Pearson Education.

[13] Moss, C. (1994). *Prolog++: The Power of Object-oriented and Pogic Programming*. Addison-Wesley.

[14] Golemanov, T. (2015). Declarative and Imperative Approaches for Proving Turing Completeness of SPIDER/CNP. *University of Ruse Annual*, 131–135. (in Bulgarian).

[15] Kratchanov, K. (2017). CINNAMONS: A Computation Model Underlayng Control Network Programming. *7th International Conference on Computer Science, Engineering & Applications (ICCSEA 2017)*, Copenhagen, Denmark, 1–20.

[16] Golemanova, E., Kratchanov. K., & Golemanov, T. (2009). Spider vs. Prolog: Computation Control. *10th International Conference on Computer Systems and Technologies (CompSysTech 2009)*, Ruse, Bulgaria, I.10-1-II.10-6.

[17] Winston, P., (1992). *Artificial Intelligence*, 3rd ed. Addison-Wesley.

[18] Russell, S. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach,* 3rd ed., Pearson.

[19] Golemanova, E. (2013) Declarative Implementations of Search Strategies for Solving CSPs in Control Network Programming. *WSEAS Transactions on Computers*, *12*(4), 176–182.

[20] Golemanov, T., Kratchanov, K., & Golemanova, E. (2009). Spider vs. Prolog: Simulating Prolog in Spider. *10th International Conference on Computer Systems and Technologies (CompSysTech 2009),* Russe, Bulgaria, II.9-1-II.9-7.