Spider vs. Prolog: Simulating Prolog in Spider

Tzanko Golemanov, Kostadin Kratchanov, Emilia Golemanova

Abstract: Control Network Programming is a programming paradigm that integrates ideas from imperative programming, declarative programming, rule-based systems, nondeterministic programming and graphical programming. Its computation rule is based on an extended specific version of backtracking. The report presents a comparison of the Spider language for control network programming versus the logic programming language Prolog.

Key words: Control network programming, CNP, Spider, logic programming, Prolog, programming languages, programming paradigms, AI programming, nondeterministic programming, declarative programming, integration of programming paradigms

1. INTRODUCTION

This report is the first part in a series of two works (the second one being [10]) presented to this conference and sharing a common objective – comparison between the programming paradigms of Control Network Programming (CNP) and Logic Programming. These programming approaches, although very different, have certain common fundamental underlying ideas. The by far most well-known programming languages implementing these paradigms are Spider for CNP and Prolog for logic programming, respectively. More precisely, this report compares Spider and Prolog.

Logic programming and Prolog are now classics; the same cannot be said about CNP and Spider yet. Therefore, we start with an introduction to CNP and Spider.

2. CONTROL NETWORK PROGRAMMING

Control Network Programming is a style of high-level programming suggested and having been developed by the authors and other collaborators [16-19]. It has evolved from three fundamental, most widely used programming approaches – imperative programming, declarative programming, and problem solving with rule-based systems (RBS). CNP has combined features from its three roots, at the same time substantially extending their potential [16]. It can be considered as a new, powerful programming paradigm.

The principal part of a program in CNP is the so called Control Network (CN) which is a textual representation of a set of graphs called subnets. The arrows in the CN are labelled with sequences of elementary actions (procedures) called primitives.

1.1 Fundamental characteristic features of CNP

Some of the important characteristic features of the CNP approach include:

- A CN program is visualized as a set of graphs. Many problems can be naturally represented in a similar graph-like form and are, consequently, prime candidates for a CNP solution [16-19].
- The program may represent a nondeterministic algorithm where, from a given point, the control can follow any of a set of possible branches and, if necessary, backtrack and attempt other alternatives [16-19].
- The program can be of declarative nature, i.e., can be a simple representation of the problem itself, without specifying any (usually much more complex) algorithm for solving this problem. Alternatively, typical procedural solutions can also be easily programmed [17].
- There are powerful but easy to use means (control options and system nodes) that allow the programmer substantial control over the computation (inference) process. Using these means the programmer can improve the efficiency of the solution and easily implement various types of heuristics [8,9,19].
CNP allows a convenient combination of procedural and non-procedural features – while the structure of the CN is generally nonprocedural and nondeterministic, the primitives used in it are procedural [16-19].

Our CNP programming language uses an “underlying” language (such as, for example, Pascal, Delphi or C#), and embodies all the powerful features of this underlying language, including object orientation [16,18,19].

It is the greatest advantage of CNP that there is no need for the CNP programmer to try to translate a “natural”, inherently graph-like, possibly nondeterministic, possibly declarative specification of a problem into a much more complicated and difficult to understand sequential algorithmic model. As most programming paradigms CNP is universal; however, it is especially effective for the type of problems described above. CNP solutions to some typical, representative procedural and non-procedural problems are described in [17].

Early versions of our CNP approach and tools were successfully employed in various application areas, in particular in diagnostic and classification systems, expert systems, compilers, natural language processing systems, robot control, state space search applications, etc. In all these applications, the CNP technique proved to be a very efficient methodology, especially in terms of the development time. Fuzzy CNP systems were also developed.

1.2 Spider: a programming environment and a language for CNP

Our most widely used programming environment for CNP is called Spider and allows a programmer to develop and run CNP programs. A non-commercial free version of the Spider CNP development environment has been made available to the scientific community at http://controlnetworkprogramming.com and http://www.regiware.info:8888/. In the core of the Spider programming environment lies the Spider programming language. The Spider CNP development environment includes a compiler from the Spider programming language into an intermediate language [16,18], an editor for Spider programs, run and debug features, and other tools. Technically, Spider was created as a superstructure over Borland Pascal and uses Borland Pascal as its underlying language. It has been actually embodied into the Borland Pascal programming environment where it can be found in the Tools menu. All usual functions provided by the Borland Pascal programming environment are also available. Recently, a first version of a newer, more powerful and more modern CNP development environment (under the working name WinSpider) was created and successfully tested. It is based on the CodeGear Delphi software development shell and allows, in particular, the development of Windows and web-based applications. Plans are underway for developing Spider# which will be integrated into MS Visual Studio. Here, for consistency with previous publications, we will use exclusively Spider as an illustrative programming language for CNP. All examples have been tested in WinSpider as well.

3. OTHER APPROACHES SHARE SOME OF THE FUNDAMENTAL IDEAS UNDERLYING CNP

As discussed, CNP is effective for solving problems of nondeterministic and/or declarative nature, as well as such that can be conveniently visualized as graph-like structures. Many other approaches attempting at solving problems with some of the mentioned above traits, are known in the literature. For example, nondeterminism and declarativeness are major features of logic programming; a further discussion follows in the next section. The so called nondeterministic programming addresses nondeterminism explicitly [1,6,7,11,12,22,27,28]. Nondeterministic extensions of many languages were suggested, e.g., Haskell, Lisp, Perl, Pandora, Scheme, Python, While, REF-ARF. Many parts of AI programming are actually based on the idea of declarative problem
representation. Graph-based approaches to programming are also known, e.g., HOPS [14].

However, all these approaches and programming paradigms aim at replacing the more traditional imperative methodologies with something completely, or at least very different. In contrast, our approach is based on complementing and integrating different techniques. Indeed, CNP is based on combining a possibly nondeterministic and possibly declarative graph-like problem specification (the control network) with imperative (including object-oriented) primitives. Currently, we work on a further level of integration – integrating CNP into a most advanced (and popular) integrated programming environment (such as CodeGear Delphi or MS Visual Studio). Thus, a program developer will have the opportunity of using CNP where it is most appropriate, while using a more classical means where CNP is not needed and will likely be less efficient. In this manner, integration is an underlying idea for our approach. We believe this gives a principal advantage in terms of popularity of techniques and tools, usability, user-friendliness, efficiency, etc. Instead of suggesting a new and unknown world, we are simply proposing an extension to the current-day most advanced and well-known world of professional programming.

4. WHY COMPARE CNP WITH LOGIC PROGRAMMING

CNP and logic programming are fundamentally different but at the same time they have interesting similarities. It is worth exploring the differences and comparing the features the two approaches have to offer. It is also useful to identify the similarities and try to project an effective solution designed within one of the paradigms, into the other paradigm in order to stimulate the development and improvement in the other area.

Firstly, logic programming shares two of the fundamental CNP objectives: nondeterminism and declarativeness. The nondeterminism in logic programming has various aspects. The behaviour of a logic program with a goal clause is in general a relation [15, para. 8.3, 8.6.B]. There are three sources of nondeterminism in the computation process [15, para.8.3, 21, para. 3.1]. As put in [24, para. 9.4], “Logic programming is a technique that comes fairly close to embodying the declarative ideal”. A very strong emphasis on the “declarative meaning” or the “declarative semantics” of logic programs is very characteristic for the logic programming approach to problem solving [2, para. 1.5, 2.3, 2.6, 5.1; 4, para. 2.1; 26, para 16.4].

Secondly, there are certain similarities in the inference mechanisms of the two styles. Although quite different, both mechanisms are based on search with backtracking. Logic programming is actually a resolution-based inference system in the Horn clauses subset of the first-order predicate calculus [15, para. 8.3, 8.6.B; 25, Ch. 11]. The logic program is a set of axioms – clauses (facts or implication rules); the program user states a theorem – goal, and the language implementation attempts to find a sequence of inference steps proving this theorem - in the process instantiating through unification the free variables which can be actually considered to be the result of the computation. As nondeterminism is involved, the inference is in fact a search process. It is widely stated that Prolog uses backward chaining (also called top-down resolution) in the original axiom set [26, para. 16.6.5]; in fact, the actual inference is equivalent to backward chaining but it is actually forward chaining in the “refuted” axiom set [15, para 8.3]. On its part, Spider uses an appropriately extended, with certain unusual features version of forward execution with backtracking [19]. The search here is not a traditional search on a “plain” but it rather propagates through the various subnets comprising the control network. When backtracking, the state of the data is restored through a reverse calculation.

![Figure 1 Example Prolog program](image)
Ideally, the computation of the behaviour is secondary [18], i.e., the results of a program are, in principle, determined by its declarative semantics and, as much as possible, the programmer should not be concerned with procedural details. In practice, however, in many cases procedural aspects are significant with ramification for efficiency, termination and choice among alternative solutions. Therefore, both Spider and Prolog provide means for control of the inference process [2, ch. 5; 3, ch. 4; 4, Ch.4; 19]. In addition, computation control in Spider is a powerful tool for an “automatic” implementation of heuristics [9]. Comparing the means for control provided by the two languages is a major objective of this comparative study.

Finally, it would be of great practical interest to compare the effectiveness of solving certain representative problems within the framework of each of the two approaches. This aspect of the comparison is, however, left for subsequent publications.

5. SIMULATING CORE PROLOG IN SPIDER

In many resources, a universal Turing machine is emulated in Prolog thus proving the “Turing-completeness” of Prolog [5,23]. As stated, “not only does a Prolog specification [of an abstract machine] provide a precise definition of an automaton's behaviour, but it also gives an immediately-executable simulator”. Below, we describe a universal way to model in Spider a program written in Prolog. Similarly to the above considerations, such a simulation shows by transitivity the Turing-computability of Spider, and can be also used as an illustration of how Prolog works.

Figure 1 shows an example Prolog program from [20]. Figure 2 gives a corresponding Spider CN that simulates the Prolog program.

The following rules specify how a CN simulating any given Prolog program can be created. A subnet corresponds to each Prolog predicate. This subnet consists of an initial node and a RETURN node, and an arrow between these two nodes for every clause of the predicate. The predicate arguments are converted into input/output parameters of the subnet; because of unification the parameters need to be \texttt{var} parameters. Prolog unification – matching with constants only – is simulated by subnet Unification which uses three simple auxiliary primitives. Matching with variables is done automatically by the mechanism for parameter passing between subnets. To cover the usage of compound
terms, subnet *Unification* can be augmented to allow matching with structures. Satisfying a goal corresponds to calling a subnet. The main subnet determines the global goal; the corresponding to this goal subnet is called. A conjunction goal in Prolog should be first replaced by a corresponding new predicate which will then play the role of a simple goal. In the case of a goal with a free variable, the variable should be initialized, e.g. to an empty string for a string-type variable.

The above procedure is universal and can be used for any core Prolog program. In some cases the corresponding CN can be simplified – a predicate can be converted into a state or even a primitive, e.g. when the primitive actually determines a function. We will not dive into more detail here, as our purpose is not to effectively model Prolog programs in Spider but to only show that a simulation of the core Prolog within Spider is possible. CNP is an efficient programming paradigm of its own, with its own programming methodology, and translation of any Prolog program into Spider is not what we are interested in.

6. **CONCLUSIONS**

The principal characteristic features of CNP as a new programming paradigm were summarized, and the importance of its comparison with logic programming was outlined. It was shown how a program in the basic subset of Prolog can be modelled in Spider. In particular, this proves the Turing-completeness of CNP.

Major points in the comparison Spider vs. Prolog are summarized below.

- A Prolog knowledge base can be simulated in Spider by a collection of trivial subnets consisting of an initial and a RETURN state and arrows between them. A Spider CN can represent a much more complex structure of subnets of arbitrary form.

- Most of the primitives in a Spider model of a Prolog program are rather trivial. The arrows in a general CNP can be labelled with primitives of any complexity (thus representing various levels and domains of thinking). The primitives have procedural nature – CNP is an integration of declarative and procedural approaches. The primitives can inherit powerful features from the underlying language. (In particular, the primitives can employ object orientation. This is not possible in Spider but has been implemented in WinSpider. Object-oriented extensions of Prolog also exist. [20])

- CNP is meant to be used in integration with (as a comparatively simple add-on of) other more traditional and most powerful approaches. Such ideas pertinent to logic programming also exist, for example for bridging Prolog and Java [13]. However, the achievement of this goal for CNP should be much more feasible as it uses an underlying language, and because programming in logic is conceptually much more distant from procedural programming.

- Structuring the knowledge base into a network is not only more convenient but also more efficient as far as the computation (search) process is concerned. In Prolog the knowledge base is amorphous – all the rules and facts are active at any one moment. Matching to satisfy a goal starts from the beginning of the knowledge base.

- Prolog, in contrast to Spider, does not support global data. Only the mechanism for parameter passing is used.

- Prolog is internally goal-oriented. It is very easy in CNP to implement both forward or backward execution.

- Spider is inherently well suited for solving problems with natural graph-like representation. Logic programs do not have such a direct resemblance.

The comparison Spider vs. Prolog continues in [10] where the means for search control in the two languages are related.
REFERENCES


ABOUT THE AUTHORS

Tzanko Golemanov (TGolemanov@ecs.ru.acad.bg) and Emilia Golemanova (EGolemanova@ecs.ru.acad.bg) are principal assistant professors in the Department of Computer Systems, University of Rousse, Bulgaria. Dr. Kostadin Kratchanov (kostadin.kratchanov@yasar.edu.tr) is associate professor in the Department of Computer Engineering, Yasar University, Turkey.
PROCEEDINGS
of the
International Conference
on Computer Systems and Technologies
and Workshop for PhD Students in Computing

CompSysTech’09

Ruse, Bulgaria, 18-19 June
PROCEEDINGS
of the
International Conference
on Computer Systems and Technologies
and Workshop for PhD Students in Computing

Edited by
B. Rachev
Technical University of Varna, Bulgaria
A. Smrikarov
University of Ruse, Bulgaria

CompSysTech’09

Ruse, Bulgaria, 18-19 June