Abstract

Control Network Programming (CNP) is a style of high-level programming that is especially effective for solving problems that have natural graph-like representation of imperative, declarative, or mixed nature. The 'program' is often nondeterministic. The report is aimed as a concise, 'modern', and relatively self-contained introduction to the background and philosophy of CNP, its implementation fundamentals, as well as its relationship to other programming paradigms.

1. Introduction
1.1. Inherently graph-like problem representations and CNP

In many cases, the innate structure of a problem to handle is not linear. For example, the primary, natural description of a problem might take the form of a tree, a graph (network), a recursive set of networks. We will refer to this type of problem specifications as graph-like representations.

In fact, such a problem specification can be of either procedural or declarative nature. In the former case, the problem description actually reflects the algorithm of the problem solution - this is the typical case in procedural (imperative) programming. In the latter case - typical for nonprocedural (declarative, descriptive) programming - the problem specification is a pure description of the problem itself, with its relationships and constraints, and possibly without any hints to an algorithm for finding a solution.

Furthermore, in many cases, the 'natural' and simple problem description might be of nondeterministic nature.

Examples of procedural graph-like representations are the flowchart of an algorithm or program (correspondingly extended to adequately handle 'sub-program' invocations), and a UML activity diagram. A classification tree, the syntax diagrams of a language and many AI problems are typical examples of problems specified declaratively.

It would be a great advantage if there was no need for the 'programmer' to try to translate an inherently graph-like, possibly nondeterministic, possibly declarative description into a much more complicated and difficult to understand sequential algorithmic model of this description. In fact, because of its naturalness, the 'human' form will be most probably the most consistent and easily verifiable representation.

An answer to this goal is the Control Network Programming (CNP) paradigm introduced below. CNP allows for an easy, natural, and efficient 'programming' of problems with graph-like representation.

1.2. CNP as a merging point of three programming paradigms

Today, we consider CNP a new programming paradigm that has evolved from, further developed and become a merging point of three fundaments, as shown in Figure 1. The relationship between CNP and other programming paradigms will be discussed in more detail in Section 4 below.

1.3. The purpose of this report

Aspects of the theory, practice, and applications of CNP (as well as fuzzy CNP) have been discussed in earlier publications, e.g. [1-10]. However, these publications are somewhat scattered, not comprehensive, and not easily available. Ideas, terminology, and solutions have been constantly evolving.
The current report is aimed as a ‘modern’, concise, and relatively self-contained introduction to CNP – its background and philosophy, implementation fundamentals, relationship to other programming paradigms. The Spider programming environment for CNP is being used where more specific illustrations are needed. Certain aspects discussed in the report have not been covered in the earlier available publications.

1.4. Development of CNP

CNP development (under a different name) began in mid-1980s; one of the early publications is [5]. Besides the current authors, a number of other collaborators contributed to the evolution of the underlying ideas and theory and development of tools, methods, and applications. Numerous programming environments that support this programming style were designed and implemented. The approach was successfully employed in various application areas, in particular in diagnostic and classification systems, compilers, natural language processing systems, robot control, state space search applications, etc. [3]-[9]. In all these applications, the CNP technique proved to be a very efficient methodology, especially in terms of the development time.

The theory and methodology were naturally extended to fuzzy CNP; specific applications in this area have been also reported.

In historical perspective, CNP actually emerged as an attempt to enhance the possibilities and flexibility of controlling the inference process in rule-based systems. Although born in an AI environment, it evolved into a universal programming paradigm. More details can be found in Section 4.

For external reasons not relayed to the qualities of CNP itself, CNP research was slow in the last years. It is gaining a new momentum recently, and we expect tangible theoretical and applied development and results to follow soon. The current report is aimed as a stimulating contribution in that direction.

1.5. Spider: a programming environment for CNP

One of the most used programming environments supporting CNP is Spider [3]. It is a successor of a number of earlier ‘compilers’ and environments for CNP. The core of the Spider methodology is the Spider programming language. The Spider programming environment includes a compiler from the Spider programming language, and other tools.

Technically, Spider was created as a superstructure over Borland Pascal. Its design philosophy envisages that Spider inherits all the possibilities and features offered by a given Pascal implementation; however, at the same time Spider ‘proper’ remains independent of the implementation. The fact that Spider is based on Pascal is not a point of principle for CNP. Environments where the underlying language is C++ or Java are being currently developed. Some earlier environments used no underlying language at all; instead, they featured a special language for writing primitives. (See Sections 2.4 and 3.2 for more details.)

The Spider environment is used here as it is most developed and most easily available.

2. CN Programs

Programming through control networks, or Control Network Programming, or just CNP, is a style of high-level programming that has been inspired by considerations and ideas like those presented in Part 1 above.

In CNP, what corresponds to a program in a conventional programming language is a description of the problem in the form of a ‘Control Network’ (CN). The CN is a finite set of subnets, one of which is the main subnet. The subnets can call each other, potentially recursively. Each subnet consists of labeled nodes (also routinely called states), and arrows between nodes. A chain of ‘primitives’ is assigned to each arrow. The primitives are elementary actions, and if a parallel is to be drawn with the traditional languages, they correspond to user-defined functions. A human can conveniently and easily apprehend a graphical representation of the CN; however a program-like representation of the CN in a text form is used as an actual CN program. Examples of both representations of a CN program can be found in Section 2.2 below.
2.1. General form of a CN program

The two principal parts of a CN program are the global declaration section and the control network (CN). This is illustrated in Figure 2.

The global declaration section can include any global definitions that are allowed in the underlying programming language (e.g., constants, types, variables, procedures and functions). Also, and most importantly, the declaration part includes the definitions of all the primitives. The primitive declarations can be part of the program itself, or can be kept ready for use in libraries.

```
uses Syst;
const ...
type ...
var ...

{&P}
procedure Prim ;
begin ...
end;
...

(*&N
MAIN MainNet;
Body
StartState: ...
> STOP ...
> OtherState;
OtherState: ...
> FINISH;
...
*)
BEGIN
END.
```

**Figure 2.** General form of a CN program

The textual description of the CN is encoded using certain very simple syntax rules. The description of the CN consists of descriptions of all the subnets; the subnets are given unique identifiers. The description of the main subnet begins with the keyword `MAIN` followed by the identifier of the main subnet. Similarly, the description of a subnet (that is not the main subnet) begins with `SUB`. The body of a subnet begins with `BODY` and ends with `END`. The body consists of the definitions of all nodes in the subnet. The nodes of a subnet have identifiers that are unique within this subnet. The definition of a node starts with its name followed by a colon and a list of all outgoing arrows. The node definition ends with a semicolon. An arrow is specified by a list of its primitives with their actual parameters followed by `>` and the name of the target node for the arrow being described.

In Figure 2, `{&P}` marks the beginning of the definition of a primitive. Delimiter `(*&N` indicates the beginning of the CN, and `*)` should be included after the CN’s end.

2.2. An example: graphical and textual representations of a CN

Figures 3a and 3b illustrate the graphical and the textual representations of a CN, respectively.

The CN consists of two subnets – Alpha and Beta where Alpha is the main subnet. The primitives used are `Setx`, `Sety`, `Print`, `Cont`, `p`, `Setz`, and `q`. Some of the primitives have no arguments while others have one argument. `Enter` is the initial node of the main subnet; the initial node is always the first one in the body of the main subnet. `Call Beta` in the description of the arrow from `Enter` to `FINISH` in the main subnet shows that subnet `Beta` is “called”, as described below. In the given example the control flows to node 1 of subnet `Beta`. In general, it is possible to enter any node when calling a subnet; this is shown using the following example format: `CALL Beta:2`. Here, the flow of control moves to node 2 of subnet `Beta`.

**Figure 3a.** Graphical representation of a CN

`FINISH`, `RETURN` and `STOP` are keywords used for special predefined type of nodes. `FINISH` denotes the end of the ‘execution’ of the CN. `RETURN` represents the successful completion of a subnet when the control is passed back to the point where the subnet was called. `STOP` denotes a node without any outgoing arrows. As we will see later, this will force the control to “backtrack”. `FINISH`, `RETURN`, and `STOP` are simple examples of the so called system nodes, or control nodes.
According to their intended role, the primitives can be divided into two types: **action primitives** and **condition primitives**.

The purpose of an action primitive is data manipulation. Figure 4 illustrates the general form of an action primitive, \textit{Prim}.

```
{&P}
procedure Prim;
begin
  if FORW then
    begin
      ...
      \textit{(data manipulation when moving forward)}
    end
  else begin
    ...
      \textit{(data manipulation when moving backward)}
    end
end;
```

**Figure 4.** General form of an action primitive

The value of the system flag \textit{FORW} determines the direction of execution mode: value \textit{True} corresponds to the normal, forward execution while value \textit{False} specifies backward execution mode. Backward execution is used during the backtracking process for restoring the state of the data to the one before the action was executed. This unusual method for backtracking that is used in Spider leads to a much more space-economical backtracking. More details can be found in Section 3.1 below.

Condition primitives control the flow of control along the CN. They allow the programmer to specify the conditions under which the forward movement is stopped (and backtracking is triggered). This effect is obtained by setting to \textit{True} the value of the system flag \textit{FAILURE}. Condition primitives do not change the data; therefore they do not normally have backward execution.

### 2.4. Should the CN program representation be dependent on an underlying language?

Allowing ourselves certain liberty of expression, we will usually say that the CN program consists of the CN and the primitive declarations. Therefore, in order to ‘write’ the CN program, we need to define the CN and the primitive declarations.

The CN has a graph-like structure. In Spider, it is represented using the very simple textual language for defining the CN that was described above. In general, the CN can be alternatively entered using a graphic editor.

Primitives are the elementary actions (operations) used in a CN program. To define them, it is possible in principle to introduce a new special programming language and write a corresponding translator for that language. In that case the overall language used to represent a CN program would not depend on other existing ‘underlying’ programming languages.

Alternatively, existing programming languages could be used for specifying the primitives. In Spider (that was meant to be a simple enough tool for illustrating and experimenting with the CNP ideas), this underlying language is Borland Pascal. As discussed briefly in Section 1.5, Spider is a superstructure over Pascal and inherits all the possibilities and features offered by the language implementation.

Getting tied to an existing underlying programming language can be considered as a drawback. However, this approach also offers substantial advantages. Firstly, the task of translating the CN programs becomes easier by order. Secondly, it allows that all the powerful features of a contemporary programming language be used in full. This effect is far-reaching. For example, the primitives can be written using an OO language; in other words, we can claim that the OO paradigm is used inside our CNP approach! The possible ways of amalgamating CNP and OO programming are a topic outside the scope of this report.
There is also a third and very important benefit of using an underlying language. As we will see later in Section 3.2, Spider uses an approach similar to the recursive decent technique in parsing. The CN is transformed into a proper Pascal intermediate program. This generated program embodies the structure of the specific CN, as well as the general strategy for interpreting CNs. If the primitives are written in the same language as the intermediate program, then the primitive definitions can be simply embedded into the intermediate program thus completely avoiding any translation of primitive declarations from one language to another!

Today, one can think of using programming environments in which any language from a set of languages can be used to specify the primitives, with different primitives possibly programmed in different languages.

3. “Executing” a control network
3.1. The logic of the CN interpreter

Roughly speaking, the CN program is a possibly nondeterministic description of a problem, and can be considered as a 'knowledge base'. The system (i.e., certain part of the Spider environment) must be able to find a solution to the problem using this knowledge base. In the literature on knowledge-based systems this part of a system is called an interpreter, or inference engine. In short, the interpreter has to implement a strategy for search in a recursive network. ‘Executing’ the CN means traversing the CN, starting from its unique initial node (that is, the first node of the main subnet), and executing the primitives along the way. This process will successfully finish when the interpreter arrives at a system node FINISH.

This process is an extension of graph traversal. Different problem settings are used in graph traversal: one might want to find all possible paths between the initial node and a final node of the main subnet, or just any such path, or the best path, etc. In Spider, the chosen setting is specified through the values of certain system options (discussed also in Section 3.3).

In general, the search/inference/computation can be conducted in a forward, backward, or mixed manner. We use forward computation. The interpreter simulates the nondeterministic search, using a certain strategy. Different such algorithms are known, with a great choice of various options. In fact, many of these options vary from problem to problem and it is important to have the possibility to change them. Even more, we can think of changing them dynamically in order to improve the behavior of the system or implement some heuristic considerations.

In Spider, a depth-first strategy is used. It is, in fact, a modification of the well-known backtracking strategy, with some peculiarities and additions (see, e.g., [7,10]). For better efficiency, a simplified user stack can be used (instead of a system stack) that only stores the nodes passed. The state of the global data is not stored; instead, it is restored while backtracking. Theoretically, there is no guarantee that an inverse function for each action primitive exists. In practice, however, it seems a programmer never uses primitives with nonexistent inverse actions in a place where backward movement can happen.

The interpreter starts its work from the initial node of the main subnet. Being in a node, the interpreter tries the first outgoing arrow. While passing the arrow, if possible, the interpreter executes forward, in a sequence, all the primitives assigned to that arrow. After successful execution of the arrow the interpreter moves to the target node for that arrow. This process continues until node FINISH is reached.

If a primitive cannot be executed (technically, the system flag FAILURE has value False), the primitives of the arrow that have been already run, will be executed backwards and in the opposite order, and the interpreter gets back to the source node of the arrow. Now the interpreter tries the next outgoing arrow, if such exists. If no more outgoing arrows exist, the interpreter backtracks one step, that is, it executes backwards in the reverse sequence all the primitives of the arrow along which it came to the current node, thus returning to the previous node.

The effect of CALL and nodes RETURN and STOP has been discussed already. Other types of system nodes will be discussed further, as well as a number of system options that change the execution mode.

An important enhancement of our backtracking strategy is the possibility to backtrack through a subnet. For example, in the case of the CN of Figure 3a, it is possible that after calling SUB Beta from the main subnet, Beta is successfully completed and the control is passed to the primitive q(z) that follows CALL Beta. Imagine now that the execution of q(z) is unsuccessful – in that case the control goes back to the subnet Beta but this time backwards, from the state RETURN.

3.2. Actually, no CN interpreter exists

In reality, as already mentioned in Section 2.4, there is no interpreter in the current version of Spider. Instead, a Spider compiler translates the original source program written in the Spider language into an
ordinary Pascal program. In its strategy the compiler embodies the rules for executing a CN.

The actual compilation process in Spider is illustrated in Figure 5.

![Figure 5. "Compiling" a CN program](image)

### 3.3. Controlling the search process

The built-in almost ten system options in Spider allow for a convenient user control of the inference strategy, or even for dynamic changes in the control that can be used to implement more advanced search strategies, including heuristics.

The systems options have values by default. The values can be also explicitly assigned, with their range being the whole CN, or a particular subnet, or a node, or an arrow. In addition, variables can be used to represent the values of the options. A change in the value of such a variable during computation leads to dynamic control. Finally, the current values of the options are available for use by the programmer.

Along with ‘regular’ nodes, Spider provides the advanced programmer with some additional types of ‘control nodes’. Normally, the outgoing arrows of a node are tried in the order in which they appear in the CN description. Control nodes allow for dynamic rearrangement of the arrows. Control nodes are typically used for implementing various heuristic techniques.

Systems options and control nodes in Spider are discussed in [2-4]. Some examples of their use for implementing heuristics are given in [4].

The following are some examples of system options. Option LOOPS specifies the maximum permitted number of entries into a node in its scope in the current path explored. Option ONEVISIT can prevent the search process from entering repeatedly already tried ‘fruitless’ branches of the CN. Option SOLUTIONS determines the maximum number of solutions sought. Option NUMBEROFARROWS limits the number of outgoing from a given node arrows to be tried. (Setting this option to 1 causes the interpreter to implement the well-known irrevocable search strategy.) Option ARROWCOST can be used when some criterion for optimality is related to the costs incurring when passing along an arrow. Option MAXPATHCOST causes termination of search along an arrow when the sum of the current costs exceeds a specified value. Options BACKTRACKING and RECURSION can prevent backtracking or recursion.

Some examples of control states follow. In state SELECT only arrows are tried whose evaluation equals the value of the node selector. Similarly, in state RANGE such arrows are only tried whose evaluations are within a specified range. In state ORDER the order of the arrows to be tried depends on the difference between the arrow evaluation and the selector.

### 4. CNP as a programming paradigm

As mentioned in Section 1.2, we consider CNP as a merging point and an extension of three programming paradigms: procedural programming, declarative programming, and rule-based systems.

CNP extends the imperative (procedural) programming paradigm by substantially enhancing the potential of the control part which is now a recursive set of non-deterministic networks. New language objects, such as special types of nodes in the CN, have been introduced that make the implementation of heuristics and the control of the computation process much easier and more straightforward. Nondeterministic algorithms are directly ‘programmed’. ‘Combinations’ of structured and OO programming are possible. Any procedural program can be easily represented as a CN.

CNP extends the declarative (descriptive, applicative) programming paradigm, too. A CN can be, effectively, a non-procedural description of a problem, using procedural elementary actions called primitives. The set of primitives might just include problem-domain oriented primitives convenient for end users with no knowledge of and skills in programming. In other cases, new primitives can be added in order to improve the efficiency of the computation/inference process. In particular, inference in PROLOG can be modeled in CNP.
Historically, CNP actually emerged as an attempt to enhance the possibilities and flexibility of controlling the inference process in rule-based systems (RBS). According to the prevailing practice, the rule base of a RBS is amorphous, in the sense that it contains an unstructured (possibly quite large) set of rules, and all these rules are active (i.e., can be fired) at any moment. Instead, we equip the rule base with a structure arranging the rules into a CN. This improves substantially the clarity and the verifiability of the knowledge base, increasing greatly the efficiency of the computation/inference at the same time. For many years, the type of systems that we now call CN programs used to be named ‘controlled RBS’s’, and were used in areas considered usually as typical AI application fields. Gradually, it was realized that there was no formal difference between a condition and an action in a rule. The rules in their traditional condition/action form were replaced by chains of primitives. The conditions and actions are just special cases of primitives, and this change of viewpoint led to tangible practical improvements. Furthermore, this also led to the important development of our perception of RBSs as nothing that special and different from the other computational models. Computation and inference became actually synonyms, either one being used more often than the other in particular implementation or application areas, and based effectively on search in recursive networks.

As a result, CNP emphasized the unity of the world of programming, computation, and AI.

5. Conclusion

CNP allows a convenient combination of facets and advantages of the three programming paradigms. The ‘united’ approach leads to better understanding, better tools, better opportunities to transfer techniques and ideas from one domain to another.

The methodology of solving problems using CNP has been completely ignored in this report. For instance, more than fifteen essentially different solutions to the Shortest Path in a Weighted Graph problem can be suggested: recursive, iterative, procedural, declarative, directly employing (through the search control) insipid-paths cutting, various versions of hill-climbing, A*, etc.

CNP methodology will be the subject of a forthcoming paper. Obviously, a reader needs concrete examples as compelling demonstration of the advantages of the CNP approach.

References


