Control Network Programs and Their Execution

KOSTADIN KRATCHANOV¹, EMILIA GOLEMANOVA², TZANKO GOLEMANOV²

¹ Department of Computer Engineering
Yasar University, Izmir, TURKEY
kostadin.kratchanov@yasar.edu.tr

² Department of Computing
Rousse University, Rousse, BULGARIA
(EGolemanova, TGolemanov)@ecs.ru.acad.bg

Abstract: Control Network Programming (CNP) as a new programming paradigm has been presented elsewhere, and the CNP approach has been illustrated through solving representative typical applications, many of which belong to the AI area. This report aims at more fully specifying the interpretation of a CN program, and how the inherently nondeterministic CN program will be computed.

Key-Words: Control network programming, CNP, control networks, recursive networks, programming languages, programming paradigms, AI programming, nondeterministic programming, non-procedural programming, Spider.

1 Introduction

CNP stands for Control Network Programming and means “Programming through control networks”. CNP is a style of high-level programming where what corresponds to the program in traditional programming, is called a “Control Network” (CN).

1.1 CNP as a merging point of three programming paradigms, with origins in AI

CNP combines and extends features from three fundamentals: imperative programming, declarative programming, and rule-based systems. An overview of CNP as a new programming paradigm was presented in [1]. The easiness of developing a CN program was illustrated in [2] through solutions to selected representative applications. These included typical AI problems such as the identification of animals problem (adjusted from [3]) and the Sheep, Wolves and Boatman problem. Solving inherently nondeterministic problems was also illustrated with arithmetic expression recognizers. Another well-known example, Selection Sort represented the class of imperative algorithms.

From a historical perspective, 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 verifiability of the knowledge base, at the same time greatly increasing the efficiency of the computation. For many years, the type of systems that we now call CN programs used to be referred to as “controlled RBS” or “rule networks” [4-6].

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 became 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 RBS as nothing that special and different from the other computational models [5-7]. Computation, execution, inference and search became synonyms either one being used more often than the other in particular implementation or domain areas.

1.2 Purpose and structure of the report

The structure and syntax of a CN program were introduced in [1]. The execution of the CN program was also touched on – however, its description was far from being complete. The current presentation is the first in a series of two reports (the other one being [8]) presented to this Conference and is aimed at specifying more fully the interpretation (execution) of a CN program.

We follow the “triple general computational model” approach developed in [5-7] where a clear separation is made between a computational model behavior and the computation of that behavior. Such a separation is important as CN programs are an intrinsically nondeterministic model. The behavior of a CN program is introduced in two steps. First, the notion of a recursive network is introduced in Section 4 as a simpler, convenient basis. Control Networks which are the major component of a CN program, are then
considered in Section 5 as an extension over the skeleton of recursive networks. Using the fundament laid out in this report, more advanced built-in search control features are studied in [8].

For consistency with [1,2], Spider* will be used as the specific programming tool for illustrations. Spider is a CNP-based programming language, as part of the Spider environment for CN programming. Spider is available for free download from websites [9,10].

2 The triple general computational model

The goal of a computational model (algorithm, Figure 1a) is to realize its behavior [4]. In the classical theory of algorithms, the behavior is a partial function \( B: X \rightarrow Y \) from an input set \( X \) to an output set \( Y \). In AI settings, it is most often a relation but can also be a fuzzy relation, a stochastic mapping, etc. In our further considerations, we assume that the behavior of an algorithm is a relation. For a given input element, \( x \in X \) its image, \( B(x) \) is a set; we refer to it as the solution set for \( x \).

Considering a given specific type of algorithm, we make the black box from Figure 1a transparent and equip it with a particular structure. The various computational models (classical algorithmic models such as Turing machines, rewrite systems, register machines, while-programs, etc., various types of programming languages such as assembly or high-level imperative languages, programming paradigms such as functional programming, logic programming, or programming rule-based systems) can be represented [5-7] as special cases of the so-called triple universal computational model illustrated in Figure 1b. It consists of two primary parts - operational unit and control unit - and a secondary part – interpreter (tactical control unit, inference engine).

In CN programs, the operational unit is comprised by the variable declarations and definitions of primitives [1]. The control unit is the CN.

3 The (non-existent) interpreter is responsible for computing the behavior

The behavior of a computational model is completely determined by the pair of an operational unit and a control unit. The behavior is defined as a certain composition of relations and functions [5,6]. Although well-defined, the definition is in general not constructive. This results from the fact that the behavior is a relation (in other words, the algorithm is nondeterministic) and, if using a deterministic programming language one must implement a deterministic strategy to compute the behavior relation.

A computation is the implementation of a concrete strategy to compute the model’s behavior. The computation of the behavior is the task assigned to the third, secondary part of the triple general computational model – the interpreter. (We use the words computation, execution, realization, implementation and interpretation of a CN program interchangeably.) For deterministic models the interpretation is almost trivial; however, it plays a central role in the case of nondeterministic models (such as CNP).

![Figure 1 The triple general computational model](image)

**Figure 1** The triple general computational model

**Computing the behavior** means determining the solution set \( B(x) \) for any given input element \( x \in X \). The solution set may be large or even infinite; finding all solutions may be expensive even for small solution sets. Therefore, before even starting the computation, the solution scope must be determined – that is, whether the computation will be halted after finding the first solution, or after finding a predetermined number of solutions, or finding the best solution, etc.

One of the elements of the computation strategy is the chosen direction of computation. In principle, the computation of the composition of two relations can be implemented forwards, backwards, or using a mixed strategy. We will use forward execution as we execute primitives which are functions, and executing functions forwards is easier. (However, we can easily imitate backward chaining - see [2].)

The execution of a CN program is discussed below in this report and in [8]. We use a special, unusual generalization of the well-known backtracking strategy.

In reality, the interpreter is for us a conceptual notion only. There is no real interpreter in the current version of Spider [1]. Instead, a compiler translates the original source program written in the Spider language, into an ordinary Pascal program. In its strategy, the compiler also embodies the rules for interpreting a CN.

4 Recursive networks

The notion of a recursive network and its behavior are formally defined in this section. A CN as an extension of this notion is introduced in the following section.

A graph is an ordered 6-tuple \( G = (V, A, source, target, L, label) \) consisting of: a finite set, \( V \) of elements called vertices (nodes, states); a finite set, \( A \) of arrows; two functions source: \( A \rightarrow V \) and target: \( A \rightarrow V \) mapping an arrow \( a \in A \) into its source \( v \in V \) and target \( w \in V \).
A **Recursive Network** consists of a finite set of initialized graphs, \( N = \{N_i \mid i \in I\} \). The graphs, \( N_i \) comprising the recursive network, are called **subnets**. This set of graphs satisfies three properties:

a) One of the subnets, \( m \in N \) is identified as the **main subnet**.

b) All subnets share the same label components set, \( L \). This label components set consists of two types of elements: **outputs**, i.e., elements of a given set \( Y \), and **invocations** (also called **subnet calls**), i.e., ordered pairs of the form \(<n, v>\) (more often denoted \( n:v \)) where \( n \) is a subnet, \( n \in N \), and \( v \) is a vertex of \( N \).

c) The graphs may also have three special types of vertices: **FINISH**, **RETURN**, and **STOP**. We will refer to these as **system vertices**. The system vertices have no outgoing arrows.

Formally, a recursive network is defined by the ordered triple \(<N, m, Y>\).

An example of a recursive network is shown in Figure 2. The various graphical symbols used to represent vertices are specified below:

<table>
<thead>
<tr>
<th>Graphical symbol</th>
<th>(Ordinary) vertex</th>
<th>Default initial vertex</th>
<th>FINISH vertex</th>
<th>RETURN vertex</th>
<th>STOP vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the recursive network on Fig. 2 \( N = \{N1, N2\} \) and \( Y = \{y1, y2, y3, y4, y5, y6\} \). For convenience, the vertex sets of the two subnets are disjoint; this however is not a requirement. The labels of the arrows are shown. Recall that a label is a finite sequence consisting of zero or more outputs and invocations. There are two invocations shown: \( N2:5 \) in the main subnet \( N1 \), and \( N1 \) in subnet \( N2 \). The former shows that subnet \( N2 \) is called with initial vertex 5 which is not the default initial vertex of \( N2 \). If a subnet is called with its default initial vertex, then the initial vertex may be omitted. Therefore, the later invocation \( N1 \) is equivalent to \( N1:1 \).

Intuitively, the “functioning” of the exemplary recursive network can be explained as follows. Initially, the “control” is in the default initial vertex, 1 of the main subnet. Vertex 1 has two outgoing arrows. The control can “take” any of them. Assume that the arrow to vertex 3 is chosen. The recursive network “produces” output \( y2 \) (the label of the arrow) and the control moves to vertex 3. Assume that now the outgoing arrow with label \( y4,y5 \) is chosen. The control moves to the system vertex **FINISH**, and the functioning is halted. As a result, the following sequence of outputs is produced: \( y2, y4, y5 \). Such a successful sequence will be called a potential solution path.

As the recursive network is nondeterministic, a whole set of potential solution paths exist. Let us find another one of them (Figure 3). From the default initial vertex of the main subnet the control moves to vertex 3, producing output \( y2 \). From here, invocation \( N2:5 \) is activated. This means that the control moves to the (non-default) initial vertex 5 of subnet \( N2 \). From vertex 5 the control moves to vertex 7 producing output \( y5 \). From vertex 7 the arrow towards **RETURN** is taken, and subnet \( N1 \) is invoked with the default initial vertex 1. Assume that now the control moves from 1 to 2 with output \( y1 \), and then to **RETURN** with output \( y3 \). The subnet call is ended, and the control returns to the point after the subnet invocation, namely, after \( N1 \) on the arrow from 7 to **RETURN** in \( N2 \). This, however, is the end of the label, and therefore the control moves to **RETURN**, and that is the end of \( N2 \). Now the control returns to the point after the invocation of \( N2 \), on the arrow from 3 to 1 in \( N1 \).

There is another output, \( y3 \), in the label of this arrow, and this output is produced. The control moves to 1, and from there to 3 and **FINISH**, producing outputs \( y2, y4, y5 \). The effect of **FINISH** is the total halt of the functioning. The potential solution path produced is: \( y2, y5, y1, y3, y3, y2, y4, y5 \).

The set of potential solution paths includes the two paths found above, as well as other paths. We will refer to this set of potential solution paths as the **behavior** of the given recursive network. The behavior can be an empty set, a non-empty finite set, or an infinite set. The
latest happens when the recursive network contains a loop; a loop can spread within a given subnet, or can include vertices of different subnets.

More formally, the behavior of a recursive network includes all potential solution paths obtained using the following rules. Initially, the current vertex is the default initial vertex of the main subnet, and the current path is empty. From the current vertex the control may take any outgoing arrow. The control passes the components (outputs or invocations) of an arrow’s label in the order they are written in the label. If the label component is an output, this output is appended to the current path. If the label component is an invocation, then the control moves to the subnet and the initial node specified by the invocation. A RETURN vertex denotes a “successful exit” from a subnet. The control returns to the point in the label right after the latest invocation of the subnet. (In the main subnet, if it has not been invoked by a subnet, a RETURN state is equivalent to a FINISH state.) A FINISH vertex denotes “success” and causes halt of computation. The current path is declared a potential solution path. If the current vertex is a STOP vertex, the control cannot propagate further and no potential solution path will be produced as an extension of the current path. A vertex with no outgoing arrows is equivalent to a STOP vertex. After passing all the label components of an arrow, the control moves to the vertex that is the arrow’s destination.

Note that the definition of the behavior of a recursive network is not constructive – the process of identifying a potential solution path is nondeterministic. A CN interpreter must deal with the nondeterminism and implement a deterministic algorithm for finding one or more solutions.

Constructs embodying similar ideas have been used in the literature before, e.g., under the name of “Recursive Finite-State Automaton” [11, Sect. 8.3], “Recursive Transition Network” [4; 12, Sect. IVD2; 13, Sect. 7.4], “Recursive Control Graph” [5, Sect. 4.4].

5 From Recursive Networks to Control Networks

To formally introduce the concept of control networks we will represent them as recursive networks, with some additional properties related to labeling. Usage of primitives in labels is the most important of these.

As defined in the previous section, an arrow label in a recursive network is a finite sequence of outputs (i.e., elements of $Y$) and/or invocations. The arrows in CNs are labeled with sequences of primitive calls and invocations (subnet calls) [1,2].

The invocations in CNs have syntax and semantics very similar to those defined for recursive networks. Subnets in CNs may have formal parameters, local variables and system options, and invocations may have actual parameters. System options will be discussed in detail in [8]. Parameters and local variables are used in the technical example below, as well as in the examples in [1,2,8]. The focus in this report is on the fundamental CN interpretation algorithm. Therefore, for our purposes here we can assume that invocations in CNs are identical to invocations in recursive networks.

![Figure 4 Technical example](image)

An example CN is shown in Figure 4. This technical example is especially chosen as an illustration of various features of the CN programs and first of all of the details of their execution. In contrast, numerous meaningful examples can be found in [2].

The following primitives are used in the CN depicted in Figure 4: `SetStr(var para: string), SetInty, Print(mess: string), Cont, IncPr (para: integer)`. We don’t need to know their definitions for the considerations here. A potential solution path is shown in Figure 5. In the recursive network terminology, the output set $Y$ is the set of primitives (with actual parameters) used in the CN.

CN’s in this paper are shown graphically. Actually, in Spider, a CN is represented in an equivalent program-like notation using a very simple language for representing CNs [1].
Proceedings of the 8th WSEAS Int. Conf. on ARTIFICIAL INTELLIGENCE, KNOWLEDGE ENGINEERING & DATA BASES (AIKED ’09)

6 Computing the CN behavior

The computation process is essentially a process of searching the CN with the purpose of finding a solution path from the default initial node of the main subnet to a FINISH state. The method used in Spider is an extension of the well-known backtracking algorithm, with a number of specifics. It is discussed in [8].

Let us consider here a few possible execution scenarios for the technical example of Figure 4. First, we need to define the primitives used. Assume that primitive SetStr(var para: string) sets the value of its parameter para to an entered string. Primitive SetInty sets the value of the global integer variable y to an entered integer. Primitive Print(mess: string) prints the value of its string parameter mess. Primitive Cont asks for a response from the user whether to continue - the response is considered positive if its first letter is not ‘n’ or ‘N’. Primitive IncPr (para: integer) increments and then prints out the value of its integer parameter para.

As described in [1], two pre-specified system flags are used in primitive declarations: FORW and FAILURE. Intuitively, flag FORW determines the direction of execution mode: value true corresponds to the normal, forward movement along the CN, while value 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 primitive was executed. Value true of flag FAILURE identifies a situation when the forward execution of a primitive was unsuccessful and backtracking must be triggered.

The code for two of the primitives in our technical example, Cont and IncPr, is shown in Figure 6. Cont is a typical condition primitive [1]. In forward execution, it prompts the user for an answer to the question ‘Continue forward? > ’. If the user’s response begins with ‘n’ or ‘N’, the value of the system flag FAILURE is set to true; otherwise it is set to false. Condition primitives typically have no backward action. IncPr is an action primitive [1]. Its forward action is incrementing and printing the value of its reference parameter para. The backward action is the inverse of the forward action, i.e., decrementing para.

Let us describe a few possible scenarios of executing the example CN program.

The execution starts from state Enter which is the default initial state of the main subnet Alpha. The values of the systems flags FORW and FAILURE will be initialized to true and false, respectively. The only arrow outgoing from Enter is taken, and its primitives are executed in a row. The first such primitive is SetStr(x). Assume the user enters ‘mango’ – as a result the value of x will become ‘mango’. Then SetInty will be successfully executed, and assume the user sets the value of y to 5. Then primitive Print will cause the printing out of the text ‘mango’. The primitive Cont will be activated now, and will ask if the user wants the forward execution continued. Assume that the user’s answer is ‘no’. This will cause the value of FAILURE to be set to true (see Figure 6). For its part, this will cause a change in the value of FORW to false. Primitives Cont, Print, SetInty, and SetStr (the primitives of the current arrow that have been executed forward already) will be executed backwards, in this opposite order. The control will reach state Enter again. State Enter has no remaining outgoing arrows – therefore, the computation will be completed. Completing the computation means nothing has been found – we will call this event an unsuccessful completion.
A second scenario is illustrated in Figure 7. After executing primitives SetStr(x), SetInty and Print as in the previous scenario, the user answers with ‘yes’ to the question by Cont. The control goes to CALL Beta(x), and to state 1 of subnet Beta. The arrow that is first in the description of state 1 will be tried first. This is the arrow with target the system state STOP. Primitive IncPr(y) will increment the value of y from 5 to 6, and will print out 6. Then the user answers positively to Cont. The control is passed to state STOP which causes the value of FORW to be reset. Primitives Cont and IncPr will be executed backwards. The backward execution of IncPr will restore the previous value of 5 of y. The next not-attempted arrow outgoing from state 1 will be tried now - the arrow with label Print(‘Monday’). Text ‘Monday’ will be printed, and the control will be passed to state 2. The arrow towards RETURN will be tried. Assume that Cont is passed successfully. State RETURN causes exit from subnet Beta. The control is back in subnet Alpha, right after the invocation CALL Beta(x). Assume that the user now answers with ‘no’ to the question posed by Cont. Cont will be executed backwards, and the control will enter backwards into the subnet Beta through the RETURN state! The backward movement will continue through Cont, Print(‘Monday’) and into state 1. There is one more arrow outgoing from 1, and this arrow will be attempted. ‘Tuesday’ and ‘mango’ will be printed. Note that ‘mango’ was passed to Beta as value of its parameter str. Assume the control passes forwards through 2, Cont, RETURN, and again into Alpha after CALL Beta(x). Assume that again the result of Cont is negative. This will cause backward movement through Cont, RETURN and Cont into state 2. Note that the system remembers how state 2 was reached! Therefore, the backward movement will take a different route now, and will pass through Print(str), Print(‘Tuesday’), 1, CALL Beta(x) in subnet Alpha, Cont, Print(x), SetInty and SetStr(x) into Enter. The CN will be once again completed unsuccessfully.

A third, successful scenario is illustrated in Figure 8. The solution part actually is exactly the potential solution path shown in Figure 5.

7 Conclusion

CNs were introduced as an extension of the notion of a recursive. The behavior of a CN was introduced as the set of potential solution paths. Then, the computation of the behavior of a CN program was discussed. A technically reach but not meaningful example was used in order to illustrate the concepts and algorithms introduced. In [8] which is a continuation of this report, the basic algorithm of the CN interpreter will be formally specified followed by a discussion of the so called static system options that are powerful means for user control of the computation process.

References:


