Control Network Programming Illustrated: Solving Problems with Inherent Graph-Like Representation

Kostadin Kratchanov*  Emilia Golemanova  Tzanko Golemanov
Dept. Computer Science & IS,  Department of Computing,  Department of Computing,
Mount Royal College,  Rousse University, Rousse,  Rousse University, Rousse,
Calgary, AB, Canada  Bulgaria  Bulgaria
KKratchanov@mtroyal.ca  EGolemanova@ecs.ru.acad.bg  TGolemanov@ecs.ru.acad.bg

Abstract

Control Network Programming (CNP) is a style of high-level programming created to be especially convenient for solving problems with natural graph-like representation. Showing that this goal has been achieved is the purpose of the current report. CNP solutions to four problems representative of four important problem classes are presented. Most of the problem descriptions are nondeterministic and declarative, without specifying an algorithmic solution. These natural problem descriptions are easily converted into working Control Network programs.

1. Introduction

1.1. Control Network Programming as a new programming paradigm

The background, philosophy and implementation fundamentals of Control Network Programming (CNP) were concisely described in [1], as well as its relationship to other programming paradigms. CNP is a merging point and combines and extends features from procedural programming, declarative programming, and rule-based systems.

A program in CNP consists of two major parts: a global declaration section, and 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. The primitives used in the CN are defined in the global declarations section. A human can conveniently and easily apprehend a graphical representation of the CN; however a textual, program-like representation of the CN is used as an actual CN program.

The CN may actually be nondeterministic. The system “executes” the CN by implementing a backtracking-like search strategy for traversing the CN. Powerful means for controlling the execution (also called computation, inference, or search) by the CN programmer are provided.

For more details the reader is referred to [1], as well as to the two web sites especially devoted to CNP, [2, 3].

1.2. Inherently graph-like problem representations and CNP

In many cases, the primary, natural description of a problem to handle takes the form of a tree, a graph (network), a recursive set of networks. We refer to these types 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.

* The first author would like to acknowledge the financial support by the Mount Royal College Research Reserve Fund.
Furthermore, the ‘natural’ and simple problem graph-like description is often of nondeterministic nature.

It is the greatest advantage of CNP that there is no need for the CN 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.

CNP is universal, as most other programming approaches also are. It is especially convenient and effective for solving problems that can be naturally visualized using graphs. As a matter of fact, it has been created with exactly that goal in mind. A CN is a possibly nondeterministic, possibly declarative description of the problem in hand.

1.3. The purpose of this report

The current report is aimed, firstly, to illustrate how CNP can be used to solve certain exemplary well-known problems of different nature. This can be considered as an initial step towards developing a methodology for CNP. Secondly, the report is intended as an argument in favor of our claim that CNP is especially appropriate for solving problems with graph-like representations.

While [1] answers the question “What is CNP?” this report addresses the question “How is CNP used?”.

All CN programs presented here are written in Spider – a language for CNP [1, 4]. A free version of Spider† can be downloaded from [2, 3]. Working copies of the programs discussed are also available on the websites.

2. Identification of Animals

Figure 1 describes a toy animal classification system (adjusted from [5, Chapter 7]). The information illustrated by the tree-like structure allows for the identification of an unknown animal answering appropriate questions about this animal’s appearance and other characteristics. (Technically, the graph is not a proper tree as a node can be accessed through different paths.)

Note that the problem representation above is descriptive (declarative) – it simply describes the problem without any hint for a possible algorithm – solution. The problem description is also of nondeterministic nature – from a given node, there is

† A new version of Spider (called WinSpider) has also been developed. It creates Windows-based and online applications.
in general a set of various paths that may be explored while searching for a solution.

Figure 2 illustrates the process of deriving the CN for solving the animal identification problem. The CN consists of the single main subnet shown on the figure. Each node in the problem representation (Figure 1) corresponds to a node in the CN (Figure 2) under the same name. Two additional nodes have been added: an initial node, Enter and a system node FINISH.

Three primitives are only used. The first one, init only clears the screen and asks what the given name of the animal is.

The second primitive, ifF(question: string) is a typical condition primitive [1]. It asks the question which is the string taken as its parameter, and reads the user’s answer. In case of a positive answer, the execution of the primitive completes successfully (and the control moves to the next primitive associated with the current arrow, or to the destination node if the arrow has no more primitives. In case of a negative answer, the value of the system flag FAILURE is set to true. Let us recall that value true of FAILURE will disrupt the forward execution, the primitives of the current arrow that have been already executed will be executed backwards, and the control will return to the node which is the source of the arrow. A new, not attempted before outgoing from the node arrow will be tried if such an arrow exists, otherwise a backward transition will take place through the arrow along which this node was entered.

The third primitive, thenF(assertion: string) causes a beep and then displays a text similar to the following: “Tara is a tiger!” where Tara is the stored name of the animal, and ‘a tiger’ is the value of the parameter assertion. All arrows to node FINISH are assigned a thenF primitive. For example, the primitive on the arrow from node “Cheetah” is thenF(‘a cheetah’) as shown on Figure 2, the primitive on the arrow from node “Tiger” is thenF(‘a tiger’), and so on. The definition of primitive thenF is shown in Figure 3.

All arrows between the animal nodes are assigned one or more ifF primitives. For example, the arrow from “Animal” to “Bird” in the problem description on Figure 1 was labeled “flies, lays eggs”. In the CN, the corresponding arrow from node “Animal” to node “Bird” will be assigned the following sequence of primitives: ifF(‘flies’), ifF(‘lays legs’). During the execution, if the animal name is Tara, this will cause the following questions to be displayed: “Tara flies?”,”Tara lays eggs?”.

Figure 3. Primitive thenF

Figure 4. CN for Animal Identification

Primitives ifF in Figure 2 are shown only for this arrow; similarly, the labels of all other arrows from Figure 1 must be replaced by ifF primitives in Figure 2.
The CN in Spider is shown in Figure 4.

Note that the value of the system option BACKTRACKING is set to No for the entire main subnet. This makes the computation more efficient. Spider system options are discussed elsewhere.

The solution described above implements the so called forward chaining strategy [5]. Implementing a backward chaining strategy [5] to the problem of Figure 1 is equally easy and is shown in Figure 5. Here, the CN copies the problem representation (Figure 1) “backwards”. Primitive \( \text{if } F \) is identical to the one in the “forward” solution, and primitives \( \text{Init} \) and \( \text{Then} F \) are similar. Primitive \( \text{Set } H \) sets the value of a variable, \( H \) to the name of the corresponding species. Upon finding a successful path, primitive \( \text{Then } F \) on the final arrow to node \( \text{FINISH} \) will display a message similar to “Tara is a tiger!” where ‘tiger’ is the value of \( H \).

\[
\begin{align*}
Z & \rightarrow E \\
E & \rightarrow T + E \\
E & \rightarrow T - E \\
E & \rightarrow T \\
T & \rightarrow F * T \\
T & \rightarrow F / T \\
T & \rightarrow F \\
F & \rightarrow (E) \\
F & \rightarrow a / b / c
\end{align*}
\]

3. Arithmetic Expression Recognition

In this problem, we must determine if a given string is a legal arithmetic expression according to the rules of a specified (context-free) grammar. Often, the grammar is specified by production rules or syntax diagrams. The production rules for our example grammar (of simple arithmetic expressions) are given in Figure 6, and equivalent syntax rules – in Figure 7. They are adapted from those in [6, Chapter 5] and [7, Appendix 2].

For instance, the string \( a^*(b-c/a) \) is a legal expression in this grammar while \( a^*bb- \) is not.

Similar to the Animal Identification example, the problem description in the form of a set of production rules or syntax diagrams is descriptive and nondeterministic. It is straightforward to convert the problem definition into a CN. A CN developed from the production rules of the grammar is given in Figure 8. Its relationship to the production rules is self-evident. For example, subnet Expression corresponds to production rules numbers 2 through 4.

![Figure 5. CN for Animal Identification - backward execution](image)

![Figure 6. Expressions Grammar (production rules)](image)

![Figure 7. Expressions Grammar (syntax diagrams)](image)
Test('c') (where c is a given character constant) is used in the other subnets. It reads the current symbol and checks if it equals the value of c.

Another, equivalent version of the CN can be designed that corresponds directly to the grammar syntax diagrams (Figure 7). For example, subnet Expression will have the form shown in Figure 9. Note that the nodes of the syntax diagram have been converted into arrows, while points in the syntax diagram have been converted into nodes in the CN.

4. Selection Sort

Selection Sort is a well-known simple sorting algorithm [8, Chapter 8]. It is illustrated in Figure 10, and its UML activity diagram is shown in Figure 11. The activity diagram is a procedural description of the problem. It specifies the algorithm of the problem solution.

The activity diagram has a graph-like structure. Therefore, it is straightforward to convert it into a CN. The latter is an example of how a procedural problem description can be easily emulated as a CN. Note that all procedural descriptions are algorithms, and therefore can be represented by an activity diagram.
In the graphical representation of a CN, a minus sign near the beginning of an arrow specifies that the corresponding arrow must be the last outgoing arrow from the source node (and therefore will be executed only after all other arrows are attempted). Such an arrow on Figure 12 is the arrow from node Check to node FINISH. Similarly, an arrow with a + sign is the first to be tried.

5. Sheep, Wolves, and the Boatman

Missionaries and Cannibals [9, Chapter 11] is a well-known problem used widely in AI texts. The problem has many variations [10]; of them the Sheep and Wolves problem is considered to be politically correct. σ sheep and ν wolves arrive on the left bank of a river. There is a row boat with a ferryman who will carry them across the river to the right bank. There are two constrains: (a) the boat cannot carry more than two passengers at a time, and (b) the wolves on either bank at any time cannot outnumber the sheep on that bank. Given numbers σ and ν, it is required to find a sequence of river crossings that will transport the entire party across the river.

This problem is representative of the approach for solving any of the wide class of state space search problems in CNP.

The state space of this problem is illustrated in Figure 13 for the case when σ = 2 and ν = 2. We assume that a state has the format <sL, wL, sR, wR, b> where sL is the number of the sheep on the left bank, wL is the number of the wolves on the left bank, sR is the number of the sheep on the right bank, wR is the number of the wolves on the right bank, and b has possible values L or R that represent the boat being on the left or the right bank, respectively. The initial state will be <2, 2, 0, 0, L>, and the final state - <0, 0, 2, 2, R>. A solution is represented as a path between the initial and the final state in the state space.

In general, there are five possible transformations from a given current state – these correspond to the following possibilities for who is carried across the river in the boat: (a) one sheep; (b) two sheep; (c) one sheep and one wolf; (d) one wolf; (e) two wolves. The five outgoing arrows from a given state in Figure 13 correspond to these five possibilities. However, some of the transformations or new states are impossible or illegal: (a) wolves on a bank outnumber the sheep, e.g. state 1210R; (b) repeating states, e.g. state 2200L at the bottom of the diagram has already been visited along the current path; (c) transformation not possible, e.g. transformation 1s1w from state 0220R, as there are no wolves on the right bank while 1s1w requires one wolf on the boat.

For this particular example, there will be four successful paths (solutions). One of them, for instance, is 2200L > 1111R > 2101R > 0121R > 0220L > 0022R.

The way of building the state space illustrated in Figure 13, is only a problem description not a problem solution. Figure 13 is, once again, an example of a descriptive problem representation (the constraints of the problem), not a specification of a solution. Also, the problem description is nondeterministic (there are multiple outgoing arrows that can be explored).

Figure 14 shows the CN of an iterative solution to the Sheep, Wolves and Boatman problem. The major primitive is move(s,w) where parameters s and w are respectively the number of the sheep and wolves on the boat. Global variable st represents the current state.
Figure 14. CN for Sheep, Wolves and Boatman (iterative)

Primitive \textit{init} initializes it to $<$\sigma00L>. Primitive \textit{move(s,w)} transforms the current state into a new state in correspondence with the values of \textit{s} and \textit{w}. The primitive also checks the three constrains: the new state is legal, the new state is not a repeating one, and the transformation is possible. This primitive has an inverse action that restores the previous state in the case of backward execution. Primitive \textit{isFinal} checks if state \textit{st} is final.

Figure 15. CN for Sheep, Wolves and Boatman (recursive)

Figure 15 illustrates the CN of a recursive solution. The modified primitive \textit{move(st,s,w,var newSt)} has the current state, \textit{st} and the new state, \textit{newSt} as parameters, in addition to \textit{s} and \textit{w}. Subnet \textit{FindSol} implements a transformation (a river crossing), and then calls itself with the new state as a parameter of the call. Note that subnets in Spider can have parameters, similarly to the primitives.

Many other modifications of the solution to the Sheep, Wolves and Boatman problem are also possible.

6. Conclusion

The programming in CNP of four problems has been discussed. The problems are representative of four different typical problem classes. Generally, the problems have natural graph-like representation that can be easily converted into control networks. Although most of the problems are descriptive and nondeterministic, the CN programmer does not have to look for a deterministic algorithmic solution.

Spider has powerful means for control of the computation process. These have not been discussed in this report, and will be the subject of a special publication. The control features allow for more efficient solutions, as well as the implementation of heuristics. This report is meant to present a first introduction into the usage of CNP. The CNP methodology (problem solving in CNP), CNP software engineering, etc. are among the interesting areas for future research and publications.

References