KUNGL TEKNISKA HÖGSKOLAN

TRITA-NA-9904
ISSN 0348-2952
ISRN KTH/NA/R--99/4--SE

A Visual Programming Language for the Analysis of Uncertain Spatial Data

Johannes Keukelaar

Stockholm 1999

Licenciate Dissertation
Royal Institute of Technology
Dept. of Numerical Analysis and Computing Science

Akademisk avhandling som med tillstånd av Kungl Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknisk licenciatexamen tisdagen den 8 juni 1999 kl 10.00 i hörsal E2, Kungl Tekniska Högskolan, Lindstedtsvägen 3, Stockholm.

NADA, KTH, Stockholm 1999

Abstract

In this thesis, a visual programming language is designed with the analysis of uncertain spatial data in mind, using a modification of the dataflow paradigm. The modification replaces cyclic graphs, often used to implement iteration, with nested graphs, which may also be used for other purposes. This approach improves readability of the programs and also improves scalability.

After some discussion about different representations for uncertain spatial data, a novel approach is chosen, based on rough sets. Some results on the accuracy assessment of such data are given, and an experiment is performed with it to prove the usability of this approach.

TRITA-NA-9904 • ISSN 0348-2952 • ISRN KTH/NA/R--99/4--SE

Contents

List of Figures

Chapter 1

Introduction

The world we live in is a fascinating but complex place. No square kilometer is exactly the same as a square kilometer somewhere else. In fact, the variety is nearly limitless, from frozen (apparent) wastelands to tropical lagoons, from steep mountain tops to deep oceanic trenches, from the parched sands of the desert to the soaking rain forest. There is variation in such things as altitude, temperature, rainfall, soil type, availability of many different natural resources and so on.

In fact, even the variation varies; uncountable interlocked processes change the world continuously. Increased rainfall may wash away valuable topsoil, changing vegetation dramatically. A cold spell in wintertime may freeze water in cracks in the rocks of a cliff face, causing a landslide. As of yet unknown processes affect the ocean currents in the pacific ocean, an effect called el Niño, changing climates all over the world. On longer time scales, tectonic processes create mountain ridges and cause earthquakes.

And then there is life; glorious, abundant, fragile life. From those apparent arctic wastelands to the parched sands of the desert, there is life everywhere. Life, too, enters into the equation. Life influences the physical landscape and even the climate, and the physical landscape and the climate influence life.

Such is the complicated world that man lives in. In that world, man must make decisions, and in recent times, these decisions, for some people at least, have come to involve more and more people, materials, square kilometers and money. Where should a new golf course be located? If we decide to dam this river here, which areas will flood? There is a forest fire at that location, and the wind is from that direction, which towns are at risk, and where should we concentrate our firefighting efforts? If we decide to cut down this forest for its wood, how much money will we make?

These are difficult questions, and the answers are not easy to find. Provided that we have the right data, we will need to perform a complex analysis on that data. For the question about cutting down the forest, we will need information about the age of the stands in the forest and factors that determine the cost of making the temporary roads in the forest to transport the trees out. Then, we will need to estimate how much sellable wood is contained in the trees. Also, we will have to find an optimal network of roads through the forest such that the profit is maximized.

Analyses like these are almost impossible to perform without computers. Humans, unlike computers, just cannot handle the amounts of data involved in such calculations, make too many mistakes in lengthy calculations for such analyses, and come nowhere near the speed at which a computer can calculate such things.

On the other hand, computers notoriously lack what may be termed common sense. When you tell them to do something (usually called to program), they tend to do what you say, rather than what you mean.

When put like that, the solution seems obvious; you should say exactly what you mean. In practice, this means that you get to tell the computer what to do in a programming language. These are languages, especially suited for writing down recipes for computers to follow (called programs), that have been defined in painstaking detail. Books about one single programming language that have 400 or even 500 pages are certainly not exceptions.

So (conventional) programming languages are hard to learn. It takes time, effort and practice to learn to use them. It is still generally true, although to a smaller extent than, say, a decade or two ago, that computer programming is usually done by trained professionals.

The people who want to perform these geographical analyses, however, are not usually computer programmers. Instead, they are people who have knowledge of the problem they want solved, such as geographers, transportation planners, or biologists.

These people often find it hard to program the computer to do the analysis they want done, even if the programs for geographical analysis are often a lot smaller (since they use prefab building blocks) than what today are considered large computer programs. They have busy jobs, and may not have the time to read several hundred pages worth of dry information about how to talk to a computer. They want something that is easier to use, that doesn't take weeks or even months of training to become proficient with.

This is where visual programming comes into the picture. If you try to communicate something, but can't get the message across with words, what do you do? You draw a picture. Visual programming is just that; trying to get the computer to understand what you mean by using pictures, not just text. Of course, since we're talking to a computer here, those pictures will have to be structured, but the idea is that even structured pictures are easier to use than a conventional textual programming language.

Just like there are many computer programs that you can use if you want to edit a simple text file, many different visual programming environments could be constructed to edit programs in any one visual programming language. Given a design for a visual programming language, the design and implementation of an environment supporting this language takes a significant amount of work in itself. This thesis will cover the design of a visual programming language, but not the design of an environment supporting that language; that falls outside the scope of this work.

This was the kind of reasoning that prompted the Center for GeoInformatics, a cooperation between several departments of Stockholm University, the Swedish Defense Research Establishment and the Royal Institute of Technology, all from Stockholm, Sweden, to formulate their 'Visualization and Visual Languages' project. Two graduate students work on this project. As part of this project, my interests have gone towards the visual languages aspects. Therefore, the design of a visual programming language that can be used by inexperienced users for the analysis of spatial data is the first topic of this thesis.

When using computers to perform any kind of analysis, though, there is one more phenomenon that should be taken into consideration. Let us say that the question, once again, is how much money can be made by cutting down a forest. If a computer gives the answer that a certain amount of money can be made, people will put much more confidence in that answer than if a manual analysis had been made, or some expert had expressed some kind of estimate. But is the answer really that much more certain?

Obviously, the quality of the answer depends on the quality of both the analysis performed, and, more importantly, the quality of the input data being analyzed. How certain are we that the trees are really that old? How certain are we that only that many trees died until now? Was there really no forest fire? What about secondary growth? And the slope of the ground under the tree cover, which is important to calculate the costs of the road network being planned, how exact are the measurements of that? How does all that influence the projected earnings?

So if the computer doesn't know the uncertainty in our input data, it can't know the uncertainty in our output data. The computer will not be able to tell us if we're going to make between, say, \$0.9 million and \$1.1 million or between \$0.5 million and \$1.5 million, while the difference between those two answers may make a significant difference to those making the decisions.

In other words, if we want the computer to be accurate about how inaccurate its answers are, we have to tell it how accurate the input data is. The question of how to do that is the second topic of this thesis. This is an important question in any thesis dealing with spatial data, but even more so in one that aims to empower inexperienced users to do more complicated analyses. Of course, these two parts have to be integrated; the programming language developed in part one should use the data representation discussed in part two, so that users can easily perform analyses using uncertain spatial data.

A more basic question about representation of spatial data, the eternal vector-raster discussion, has not been an issue in this thesis. Since the very beginning of this project, an eventual cooperation with another project within CGI has been planned. This other project stands firmly on the vector side of the discussion, and therefore that is the approach taken here.

This thesis, then, consists of two parts. The first part is about visual programming languages. Chapter 2 gives a more thorough introduction to visual programming languages than that given here. Chapter 3 details the design of GROAn (Graphical Rough Object Analyzer), a visual programming language for the analysis of uncertain spatial data. GROAn contains some novel modifications of the dataflow visual programming paradigm that make the language more structured and allow the user to selectively focus on parts of the program in a natural way. Ideas for language extensibility and visual layout are also described.

The second part is about how to represent uncertain spatial data and related questions. This part is a much extended version of part of [AKO98]. Even if the original idea was Ola Ahlqvist's, my contribution to the material in the second part of this thesis is approximately 75%. Chapter 4 gives an overview of the possible approaches, and the reasons for choosing the novel rough representation used in GROAn. The question of how to measure how well two such sets of uncertain spatial data match is discussed in chapter 5. The two parts are linked together through this introduction and chapter 6, where an experiment with uncertain spatial data is performed using the visual programming language designed in the first part. Every chapter ends with a section in which conclusions about the material in that chapter are given. Overall conclusions are given in chapter 7, the final chapter.

Chapter 2

Visual Programming Languages

Research on visual programming languages (VPL) got started in an attempt to make programming and maintenance of programs easier, and therefore cheaper. There is no doubt that programming, especially when program size grows, is difficult; many books have been written about the subject. It is also a known fact that program maintenance is a major contributor to total software costs [GJM91]. Based, at least in part, on the observation that programmers, when trying to figure out some difficult part of a program, often resort to drawing pictures [Chr68], early research on visual programming languages was begun. Even in recent VPL research, terms like ease of use and intuitive are still used prominently.

This chapter describes visual programming languages in general. It starts off with a definition of the term, then briefly discusses some of the subclasses of visual languages, and continues to discuss the advantages and disadvantages to VPL, together with some of the empirical evidence to support these claims. Finally, the current main problem in the field is discussed in some more detail.

2.1 Definition

As with any field of some complexity, different definitions are possible:

- 'A Visual Language manipulates visual information or supports visual interaction, or allows programming with visual expressions. The latter is taken to be the definition of a visual programming language. [...] Visual programming environments provide graphical or iconic elements which can be manipulated by the user in an interactive way according to some specific spatial grammar for program construction.' [GR90]
- Visual programming is 'the use of visual expressions (such as icons, drawings or gestures) in the process of programming'. [Cha90]

- A VPL 'allows the user to specify a program in two-(or more)-dimensional fashion. [...] conventional textual languages are not considered two dimensional since the compilers or interpreters process them as long, one-dimensional streams.' [Mye90]

While the first two definitions focus on the visual aspect of VPLs, the third definition focuses on the multi-dimensional aspect of VPLs. Of course, there are some languages which are covered by one of these definitions, but not by another. For the purposes of this document, the definition by [Mye90] will be adopted, as it seems more well-defined.

This definition ignores *secondary notation*: 'the use of layout and perceptual cues (elements such as adjacency, clustering, white space, labeling and so on) to clarify information (such as structure, function or relationships) or to give hints to the reader' [Pet95] in textual programming languages, which may often have a two-dimensional character. Being *secondary* notation, however, it is not an essential part of the language. Of course, secondary notation occurs in visual programming languages as well.

The difference between static and dynamic representation of a visual language [BBB+95] also falls outside the adopted definition. The *static representation* is that part of the program which can be observed in such a program without interacting with it; observing the *dynamic representation* does require interaction. Examples of dynamic representation may include inspecting values by pointing at places in the program and receiving more information about elements of the program by clicking on them.

2.2 Paradigms

The definitions given above are fairly wide; programming in a two (or more) dimensional fashion can certainly be done in many ways. The VPL classification system [BB94], currently lists eleven main paradigms, and of course there is plenty of variety even within those paradigms. Of course these paradigms are not mutually exclusive; one single language may belong to more than one paradigm. They include, among others, the following:

Data-flow: The dataflow paradigm is based on a graph, where the nodes are operations and data flows from operation to operation through the edges of the graph. There is a quite natural mapping between this paradigm and data analysis. It will be described in more detail in chapter 3.

Spreadsheet-based: Spreadsheets, such as Excel, are a big commercial success. The idea here is that one has an infinite grid, where numbers, formulas and labels can be entered in the grid squares. The formulas can reference other squares, and updating is done automatically, allowing the user to automate fairly complicated computations. Useful when tabular data is used; perhaps not so well suited for spatial data analysis.

Form-based: Grouped together with the spreadsheets in [BB94]. Apparently based on spreadsheets, but more generic, since the program may consist of many forms, which need not be strictly tabular. Form-based programming uses the filling in, often with visual elements, of forms. Examples of this paradigm include Forms/3 [PB93] and Agentsheets [RI97].

Programming by demonstration: In this paradigm, also known as programming by example, one instructs the computer what to do by showing examples of proper behavior. See for example [CS98] and [CM94]. The latter is about a query language for spatial data. This paradigm seems better suited for querying spatial data than for the analysis of such.

Rule-based: Languages such as those described in [Sch90] and [RCS97], where visual if-then rules make up the body of the language. Quite suited for reclassification and such tasks; not as suited for analysis.

2.3 Argumentation

As [Whi97], an article that gives a thorough overview of the published empirical evidence that has bearing on the VPL field, says, there is a definite lack of empirical evidence backing the design decisions in the VPL field.

This has not kept people from voicing opinions and claims about visual programming languages, though. (Perhaps even quite the contrary.) In this section, generally recognized claims will be presented, with substantiation where available.

2.3.1 Evidence For VPL

As everyone knows, 'a picture is worth a thousand words', meaning that it is much easier to get a message across using pictures than it is with words. It would seem reasonable to assume, therefore, that visual representations are superior to textual ones even when the subject of representation is a computer program.

It is certainly a proven fact [Day88] that organized and explicit information improves performance and credible that visual representations can help in organizing information and making it explicit and consistent. In fact, it seems that the advantage of a visual representation grows as the problem size grows [PS74]. Strong evidence in favor of VPLs comes from [PB93], where a visual language consistently outperforms two textual ones, even for users who had prior experience with one of the textual languages.

[BBB$^+$95] gives a more explicit list of factors that contribute to the ease of understanding of VPLs:

Simplicity: Fewer concepts are needed to construct and understand programs.

Concreteness: Concrete languages allow direct exploration of data.

Explicitness: Explicit languages show relationships between elements of the program.

Responsiveness: Responsive languages show the results of changes in the program immediately.

A weak point in favor of VPLs is that it would seem to be easier for the programming environment to prevent syntax errors, since the editing environment is often custom-made. This is a weak point, since even though textual languages are often edited with general purpose text editors, special purpose editors for textual languages *could* be developed. In fact, syntax highlighting features in textual editors may be seen as a step in that direction.

Commercially, VPLs such as IBM Data Explorer, Prograph [SC97], LabVIEW, VEE, AVS [UFK$^+$89] and Khoros/Cantata [RW91] are fairly successful. They seem unlikely to replace, for example, C++ anytime soon, due, perhaps, in part to resistance to change, but in their niche they are used by small but enthusiastic communities as language of choice [Gre95].

All these arguments point towards less steep learning curves and shorter learning processes, leading us to believe that VPLs may be well suited to inexperienced users.

2.3.2 Evidence Against VPL

A hypothesis that has been confirmed by experiments time and again is the match-mismatch hypothesis [GG84]. It says that every notation highlights some kinds of information, while obscuring others, and therefore predicts that for every notation there is a task for which it will perform poorly and a task for which it will perform well. This is certainly strong evidence against the blind acceptance of VPLs as representation of programs; it is, on the other hand, also evidence *for* the guarded and reasoned acceptance of some VPLs for some tasks.

A statement that keeps popping up is the so-called *Deutsch Limit* [McI98], which states that 'you can't have more than 50 visual primitives on the screen at the same time'. There is a certain validity to this statement; visual representations often have a lower density of primitives measured in pixels on the screen when compared with individual characters on the same screen. One thing to keep in mind, though, is that often the visual primitives contain (much) more information than single characters, so comparing visual primitives with characters is not fair. Even correcting for this, though, it seems that visual representations still have a lower information density than textual ones.

Secondary notation is one of the things that sets novices and experts apart [Pet95]. Experts use secondary notation to good effect both when writing and when trying to understand programs; novices use it 'incorrectly' when writing and not at all when trying to understand programs. Perhaps a VPL meant for novices should keep the amount of secondary notation to a minimum.

As [Shn92] notes, 'The keyboard remains the most effective direct-manipulation device for some tasks'. Typing programs may be a lot faster than writing programs using direct manipulation, which may be a reason why experts would prefer text-based languages.

2.4 The Scalability Problem

The *scalability problem* is often identified as *the*, or at least *a*, key problem in the VPL field at the moment [BBB$^+$95, Gre95, CHZ96]. The problem is that, although many VPLs are quite usable for small-scale or perhaps even medium-scale programs, as program size increases, VPLs become less and less usable. As [BBB$^+$95] puts it: '[...] making visual programming languages suitable for solving large programming problems often seems to require the very complexities VPLs try to remove or simplify.'

In the same article a list of issues is given that needs to be dealt with in the context of the scalability problem: static representation, effective use of screen real estate, documentation, procedural abstraction, interactive visual data abstraction (i.e. user-defined types), type checking, persistence and efficiency. [CHZ96] adds two items: increasing complexity and viscosity (i.e. resistance to change).

Things are not quite that bleak, though. Solutions to many of these individual problems exist. For example, there are VPLs with an excellent static representation; zooming and fish eye techniques can alleviate the screen real estate problem; many VPLs have some form of procedural abstraction; the list goes on. It is just that these problems need to be solved together, in one VPL, with the overall goal of scalability in mind, and that is an open problem.

2.5 Conclusions

Certainly, VPLs are useful for some tasks, and can make programming more intuitive and programs more easily understood. These are things that will especially help inexperienced users. Even if VPLs may not yet be ready for large programs, for small to medium size programs they seem very well suited.

GIS users often come from more applied fields such as geography, geology or urban planning and often have relatively little experience with computers, so they can benefit a lot from using a VPL. Many of the analyses done by this group of GIS users are relatively small and simple, and therefore well suited for programming using a VPL. The success of VPL in the scientific computing and visualization field is another indication that such a success may also be expected in the related GIS field.

Chapter 3

Language Design

This chapter describes the design of GROAn (Graphical Rough Object Analyzer), the VPL under discussion. Starting off with a general description of the dataflow paradigm, it continues with some novel modifications to the paradigm for this VPL, and finishes with a description of features of this specific implementation of the modified dataflow paradigm. New things introduced in this chapter include the modifications to the dataflow paradigm that should make the language more structured and more easily read.

3.1 The Dataflow Paradigm

Dataflow is one of the eleven paradigms in VPL listed by [BB94]. It is a very popular paradigm that has a quite natural mapping to a sequence of operations on data [Hil92]. It also has the desirable characteristics given by [BBB^{+}95]: simplicity, concreteness, explicitness and responsiveness.

The dataflow paradigm was first introduced in [KM66], as 'the computation graph model', not as a VPL, but as a model for parallel programs. (In fact, many modern dataflow VPLs have support for distributing their computations over more than one processor.) It was defined there as a graph where each of the nodes represents a functional operation, and each of the edges represents a first-in-first-out queue containing data flowing from one node to another. Operations can only execute if enough data is present in the queue of each of the edges entering the node; some data is then removed, and some output data is produced and queued up on each of the output edges.

In the initial incarnation, therefore, cyclic graphs were the only way in which iteration could be expressed. The representation used in later publications such as [Den74] is very similar.

[McI98] gives references to many historic articles about VPLs, one as far back as 1966, but it is unclear which of those were dataflow VPLs. At least in

1982, an article has been published that proposed the use of dataflow graphs as a VPL [DK82], or, as the authors of that article carefully put it: '... explores the utility of graphical representations for dataflow programs, including the possibility and advantages of dispensing entirely with the text and viewing the graph itself as the program.'

[DK82] mentions a fair number of features of modern dataflow languages:

- Cyclic graphs are allowed for iteration.
- The 'selector' and 'distributor' nodes are introduced for conditional execution of parts of the graph. These are described in section 3.1.3.
- *Runtime* macro expansions are used as a kind of recursive subroutines.
- Tuple construction and tuple indexing is used as a naive kind of type abstraction; the technique, as described, is not type-safe.
- Graphs as data values, i.e. higher order functions.

Unlike more modern dataflow VPLs, however, the entering orientation of the edges is used to determine 'which' inputs they correspond to.

At some high level of modeling a dataflow graph, one can ignore the distinction between different inputs to a function, i.e. the distinction between different incoming edges on one node. Since one of the goals of VPLs is explicitness, however, this is a luxury that VPLs can not afford; the different inputs to a node have to be represented explicitly. Enter ports: Each node has a number of (labeled) input ports and a number of (labeled) output ports. Either of these numbers can be zero of course, for sources or sinks. The input ports differentiate between the different input parameters of the function, and the output ports do likewise with the different outputs of the function. It is from output ports to input ports that the edges in the 'graph' go.

Fan-in and fan-out are two other features found in modern dataflow VPLs. Fan-out is almost always supported; it means that many connections may *start* in one output port. In the unlikely case that fan-out would not be supported, duplication nodes would have to be introduced, whose only use would be to duplicate data values.

Fan-in is not always supported; it means that many connections may *end* in one input port. It is an alternative to selector nodes (see section 3.1.3), but its influence on the readability of graphs is unsure.

Type safeness is easily introduced by associating data types with ports. Connections between ports are then restricted to pairs of ports that are compatible in some way; equal types is a common restriction. Procedural abstraction is likewise easily introduced by specifying a procedure as a graph with certain inputs and outputs; a procedure call is then a node that has a reference to a procedure graph.

3.1.1 Iteration

More recent articles about dataflow VPLs are divided about the way they support iteration. According to [Hil92], there are five different approaches to supporting iteration in dataflow visual programming languages:

- Allow dataflow graphs to be cyclic, creating feedback loops. This requires that the edges of the graph represent queues for the data flowing through the graph, as in [KM66].
- Sequential ports, which execute the node several times, changing the value of the port.
- Parallel ports, which split a list and execute the node for each item.
- A higher order function that takes a lower order function and a list as arguments, and applies the lower order function to each element of the list.
- Control-flow constructs.

Articles based on Prograph, such as [GS95, SC97], seem to have both sequential and parallel ports. Iteration is not explicitly mentioned in [Sch97], but cyclic graphs appear to be supported there. A novel approach is taken by [AD97] that requires lots of extra syntax (they also allow cyclic graphs). It appears to be a mix of parallel ports and higher order functions. Some articles, such as [Dye90, KS94], restrict graphs to acyclic ones, yet do not seem to support any other kind of iteration either (thus missing out on a substantial amount of computational power; perhaps they felt that their application domains didn't require it). [BM94] claims that iteration support is a subject for future research for the authors. Yet other articles, such as [AT95, PJ95], do not mention iteration at all.

The proper way to implement iteration in dataflow languages, in other words, seems to be under some debate.

3.1.2 Extensions

There are also extensions of the dataflow paradigm, such as the object-oriented dataflow paradigm [Kim95], and, related, the object-flow paradigm [BC97]. Interestingly, [Sch97] tries to blend in elements from logic-based programming

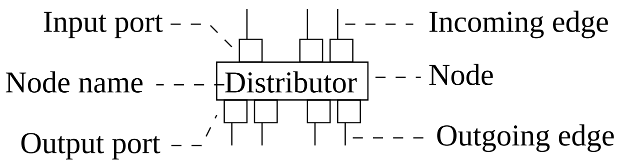

Figure 3.1. An annotated dataflow node.

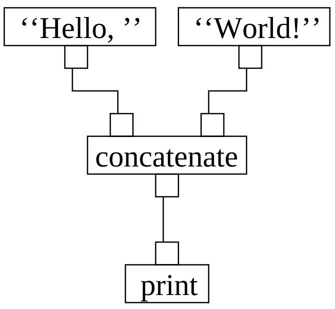

Figure 3.2. A simple dataflow program.

languages. Hi-Visual [HTI90] is a dataflow-like VPL that uses nodes for data objects, and a combination of two (data) nodes for operations. The language described in [AD97] has all kinds of extensions, mostly to support iteration in novel ways.

3.1.3 Notation

In this section, we will give a definition of the visual representation we will use for dataflow visual programs in the rest of this chapter. Since dataflow graphs are built up out of nodes, an annotated node is shown in figure 3.1. The annotations consist of the dashed lines and the explanatory texts associated with them; they are not part of the node itself. The input and output ports are shown as small, unmarked, rectangles, with no indication of their type. In case we have to refer to individual ports in the text, we will assume a numbering going from left to right; i.e. the leftmost port is the first port, and so on. In the visual programming environment being developed, these small rectangles are replaced by small rectangular pictograms indicating the types of the ports.

A complete, if simple dataflow program is shown in figure 3.2. In this case, the two nodes at the top provide string constants, and the whole program implements a slightly more involved version of that most canonical of programming language examples, hello world. The two string constants flow

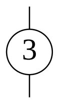

Figure 3.3. A data element in the graph.

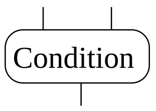

Figure 3.4. A labeled subgraph.

into the concatenate node, which sticks them together. The string "Hello, world!" then flows into the print node, which prints it to the standard output of the program.

Another element of syntax that we will use occasionally is shown in figure 3.3. It is also used in, e.g. [Den74]. In that figure, an edge is shown that is assumed to carry some data even before execution begins. The data element on the edge is written in a circle that is placed on the edge. This syntax will not be used in the visual programming environment being developed; constant nodes such as the ones in figure 3.2 will serve there instead.

Sometimes it will be useful to have some bit of syntax represent any subgraph with a fixed number of inputs and outputs. The element used in figure 3.4 is used for this purpose. The subgraph is labeled; this is so we will be able to tell the difference between different subgraphs in the same figure. Again, this syntax will not be used in the visual programming environment being developed.

A common pair of nodes, introduced in [DK82], are the 'distributor' and the 'selector' nodes, used for conditional execution. We will also use the 'merge' and the 'detector' nodes later on.

Distributor: This node has $n + 1$ inputs and $2n$ outputs. The first (i.e. leftmost) input is boolean. If the value there is true, the other n inputs are copied to the first n outputs; otherwise, they are copied to the second n outputs.

Selector: This node has $2n + 1$ inputs and n outputs. The first input is boolean. If the value here is true, the following chunk of n inputs is copied to the outputs; otherwise, the last n inputs are used instead.

Merge: Can be described as an automatic selector. Has $2n$ inputs and n outputs. As soon as input is available on either all of the first n inputs

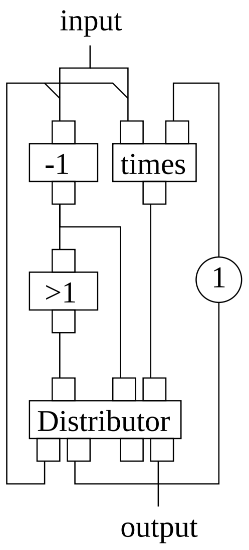

Figure 3.5. A cyclic dataflow implementation of factorial.

or all of the last n inputs, that chunk of inputs is copied over to the outputs.

Detector: Has n inputs and $n + 1$ outputs. When input is available on all n inputs, the first output, which is a boolean, gets the value true, and the others get copied from the inputs.

3.2 The Nested Directed Acyclic Graph Paradigm

The VPL developed here is based on the dataflow paradigm with two changes, neither of which is completely new. These changes have been made to increase readability and structure in the language without losing any expressive power. Readability and structure seem desirable properties for any language, but especially for one aimed at inexperienced users, like this one. The changes are:

- All graphs are acyclic. Several articles mentioned above place this restriction on their graphs, often because they support iteration in some other way. [Dye90, KS94] are the exceptions; they do not seem to support iteration at all.

  In this paper, the opinion is that iteration is needed for the users to get full use out of the language, but *iteration is best implemented in another* *manner*, as cyclic dataflow graphs are confusing, and prone to errors, especially from a user's point of view. A cyclic implementation of the factorial function can be seen in figure 3.5; a simple function, yet not easy to understand, not even when one knows what the nodes do. (In this figure, cyclic fan-in is used on the nodes labeled '-1' and 'times'.)

The node labeled '-1' subtracts one from its input, and produces that as output value. The node labeled '> 0' tests if the input value is greater than zero, and produces a corresponding boolean output value. The node labeled 'times' has the product of its two input values as output value.

- Nodes may contain one or more nested subgraphs. Many dataflow VPLs implement procedural abstraction by allowing a procedure-call node to contain a reference to one procedure graph. Here, and this *does* appear to be new, the idea is carried further, and is also used to implement such things as iteration and conditional execution.

3.2.1 Nesting Nodes

Nesting nodes are nodes that contain one or more nested subgraphs. A subroutine node, as an example, contains only one subgraph; a switch node may execute one of n subgraphs, depending on one of the inputs. To increase reusability of these nesting nodes, it is desirable to give them a variable number of in- and outputs. After all, the in- and outputs of a subroutine should depend on the actual subroutine, and with the enormous potential variety of port types, supplying all combinations of them would be rather tedious.

Examples of nesting nodes follow, with a brief description of each. Descriptions of some of these nodes are also given later, in section 3.3.6.

Procedure call: This node contains one subgraph. This type of node is quite well-known. The actual subgraph that it contains may be one from a library of defined procedures. When run, it will pass on its inputs to the subgraph's inputs, evaluate the subgraph and pass on the subgraph's outputs to its own outputs.

While: This node contains two subgraphs, the condition subgraph and the loop body subgraph. First, the condition subgraph is evaluated, producing only one boolean output. If that output value is true, the loop body subgraph is evaluated, otherwise, the node is done. Then, the loop body subgraph is evaluated, and the loop starts over again by evaluating the condition subgraph. Of course, input and output values of the subgraphs and of the node itself are set to the proper values at the proper times.

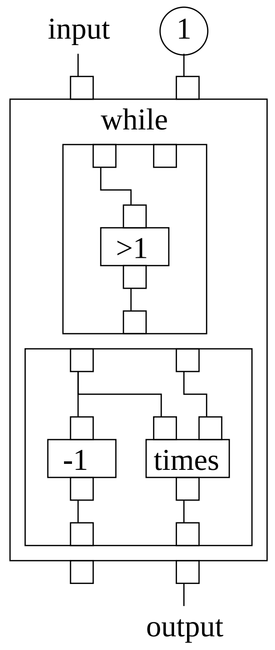

Figure 3.6. A nested acyclic dataflow implementation of factorial.

In figure 3.6 an implementation of the factorial function from figure 3.5 is given again, this time using the nesting while node. Figure 3.6 can also be used to explain the syntax for nesting nodes. The top level of this figure contains one (big) node labeled 'while'. This node, as described above, contains two nested subgraphs; these are drawn in rectangles inside the nesting node. Inputs and outputs to those subgraphs are handled, as usual, by ports.

If: This node contains three subgraphs, the condition subgraph and the true and false subgraphs. The condition subgraph is evaluated, and based on the outcome of this, either the true or the false subgraph is evaluated.

For convenience, variants of this are possible, such as a switch node, that selects one of n subgraphs to evaluate, an if node that doesn't contain an else branch (so the user doesn't have to connect all the input ports to the output ones) or an if node that has an editable expression containing those inputs that are numerical in nature, and no condition graph.

The last two examples, the while and the if nodes, could easily be implemented using the selector and distributor nodes from [DK82], although the while node would require a cyclic graph, see section 3.2.4. Besides the argument against cyclic graphs given above, there are other arguments in favor of this approach, which will be given in the next two subsections.

3.2.2 Structured Programming

In the textual programming language community, there has at one time been a lot of discussion about [Dij68], but this has died down. 'Public opinion' on this subject now appears to agree with Dijkstra; the goto statement or similar constructions which allow jumps to labeled positions in the program should not be used, as they are detrimental to program readability and maintainability.

In modern textual programming languages, the goto statement still exists, but is deprecated. Instead, programmers are advised to use more structured methods of control flow, such as the while, if-else and case statements. Most modern programmers appear to use the goto statement (or its equivalent) only extremely rarely, if at all.

An analogy can be made here to a situation in dataflow visual programming. The analogy can be made between textual programs with gotos and textual programs with structured control flow on the one hand and cyclic dataflow programs and nested acyclic dataflow programs on the other hand.

The uncontrolled feedback occurring in cyclic dataflow programs is in some sense comparable to goto statements in textual languages. Nesting nodes are comparable to structured control flow elements. Nested graphs map almost directly to syntactic blocks (i.e. BEGIN-END blocks and such) in textual languages.

As stated above, it would seem that uncontrolled feedback in the dataflow graph is confusing, hard to understand and prone to error. Maintainability suffers. It seems reasonable to say that the controlled feedback occurring in nested acyclic dataflow graphs suffers less from these problems.

3.2.3 Scalability Issues

As mentioned in section 2.4, one of the main problems of VPLs at the moment is the scalability problem; VPLs work well for small problems, perhaps even for medium size problems, but as problem size grows, usability of the VPL goes down. Even though the application area being aimed for here is one where small problems dominate, still the scalability problem is deemed to be of some concern.

The main way in which the nested acyclic dataflow paradigm addresses this issue is through the efficient use of screen real estate. There exists in this paradigm a very natural unit of information that the user could be allowed to interactively hide or show, namely the nested graph. This enables the user

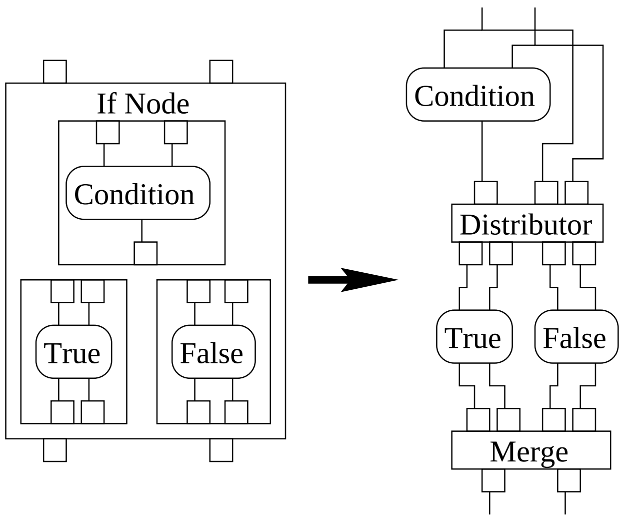

Figure 3.7. Substitution For An If Node

to selectively view only those details that the user is interested in. Thus, uninteresting details do not take up valuable screen real estate.

An apparent problem is the size of nesting nodes on the screen when their nested graph(s) are shown. These nodes would seem to take up lots of screen real estate. One should realize, however, that this real estate is *shared* with the nested graph(s).

Viscosity and complexity are two other scalability-related issues that are addressed by this paradigm. The way in which these issues are addressed has to do with graph size. Compared with classical dataflow graphs, the actual graphs in the nested acyclic dataflow paradigm will contain fewer nodes and edges, since, in effect, the single classical dataflow graph gets split up in a number of smaller ones, one top level graph and a number of graphs that nest (to various levels) in the top level graph. Because graphs are smaller, viscosity and complexity become less important issues.

3.2.4 Expressive Power

Theoretically speaking, an interesting question is how the expressive power of NDA-DF relates to 'canonical' DF (C-DF), or even to known Turing-equivalent languages such as Lisp and C. This begs the definition of C-DF; we will assume a dataflow model in which graphs may be cyclic containing procedural abstraction (not necessary, as one can use run-time macro expansion [DK82], but convenient) and the merge and distributor nodes, both with an arbitrary number of inputs and outputs.

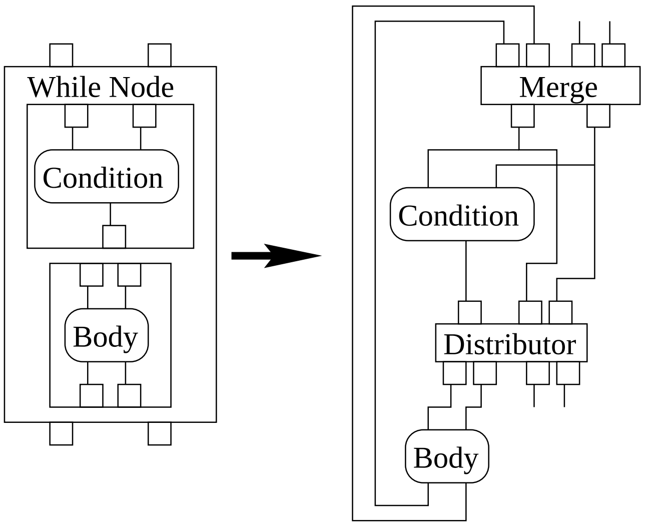

Figure 3.8. An Attempted Substitution For A While Node

It is fairly easy to see that any program that can be expressed in NDA-DF can also be expressed in C-DF. This can be proven using simple node substitution. The only two nodes needing substitution are, of course, the if and while nodes. The substitution rule for the if node with two inputs and outputs is given in figure 3.7; changing the rule to some other number of inputs and outputs is not hard. Similarly, the rule for the while node with two inputs and outputs is given in figure 3.8.

Let us try to more formally define how to read those figures. The rules consist of two parts, a left part, which contains one nesting node with some input and output ports, and a right part, which contains a part of a dataflow graph with dangling input and output links (the same number as the node on the left side of the rule). The nesting node has a name, and some nested graphs, which are noted by labeled rounded rectangles with dangling links. These same labeled rounded rectangles occur in the graph on the right hand side of the rule, and denote exactly the same graphs there.

The substitution for the while node is not perfect, however. In some sense, the while *node* is atomic; the sequence of operations is as follows: the node accepts and 'eats' some input data, does some processing, produces the corresponding output data, and only then is it ready to accept new input data. The graph part in the right hand side of figure 3.8 doesn't have that property: it will absorb input data as soon as it is available, whether the previous set of input data has been processed completely or not.

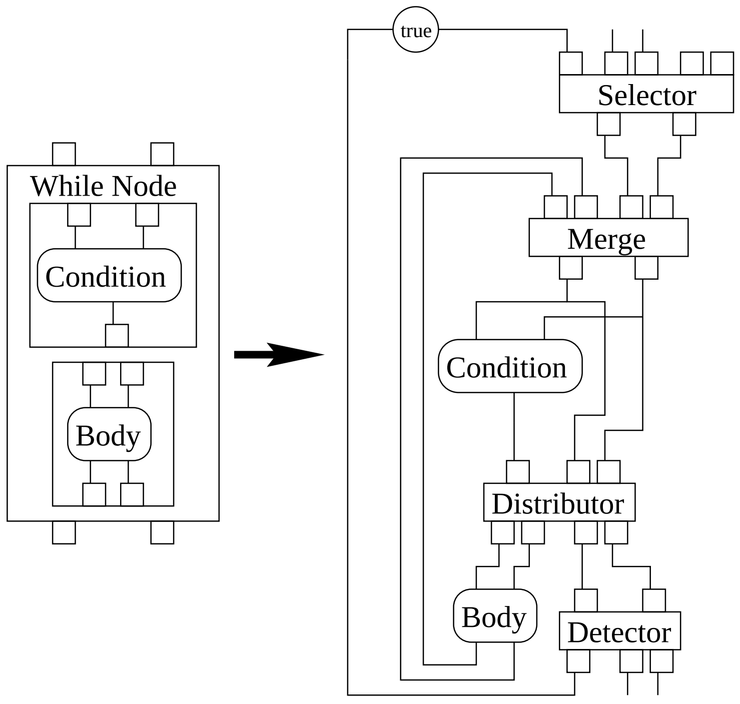

Figure 3.9. An 'Atomic' Substitution For A While Node

It should be noted that this will not create problems, the parallelism is just more fine grained in the substitution graph. This does, however, mean that there may be a difference in behaviour, which is not quite satisfactory; a rule with a substitution graph that has a level of atomicity that more closely matches that of the while node is given in figure 3.9.

What these substitutions prove is that any program that can be expressed using NDA-DF can also be expressed using C-DF. That means that NDA-DF is not more powerful than C-DF. To prove that they are equally powerful, we would also have to prove the reverse, i.e. that any program than can be expressed using C-DF can be expressed using NDA-DF, too.

This proof, assuming it exists, is more difficult to come by, precisely because C-DF programs may be much less structured than their NDA-DF counterparts. Even if in some situations one may find parts of C-DF programs that exactly match the graphs given in the right-hand side of the rules described above, there is nothing that restricts the C-DF programmer from adding in extra links to or from the rest of the graph, or even constructing graphs without any subgraphs that even come close to matching.

3.3 Features of This Implementation

Any abstract description of a programming paradigm such as the one above, may be implemented in a variety of ways, depending on the design decisions made when implemeing the system. This section describes some of the particular design decisions made for GROAn.

3.3.1 Subroutine Graphs

Each graph may have an associated library of subroutines. Nodes that call such a subroutine (being proper nesting nodes) would contain a reference to a subroutine graph from the library of the graph they are part of, or from a library at any higher level. This construction allows for subroutines that are only defined within certain other graphs, a form of detail hiding.

Editing subroutine graphs may be slightly problematic, as changes in input and output ports may mean that one has to make corresponding changes in many places in the calling graphs, one for each call to the subroutine. Of course, this is not a problem specific to this programming language; every programming language suffers from it.

3.3.2 Method-based Polymorphism of Types

Type polymorphism in the context of dataflow programming is concerned with whether or not a connection between a certain output port and a certain input port is allowed. One could of course allow any and all connections, thus losing any semblance of type safety; this does not seem like a very good idea. Most modern dataflow languages support the idea of types of data, and may only allow a connection to be made between two ports if the type of their data is equal.

In this implementation, ports are objects that know how to copy their value to another port. These ports have associated named methods that may be invoked upon them. Specific ports can therefore be created that accept connections only with other ports that implement a specific method. As an example, a printable port could accept only ports that implement the method 'print'. A printing node could use such a port type to print arbitrary data (assuming a proper implementation of the relevant method).

Type conversion is also possible. When a connection is attempted from an integer port to a string port, the integer port could be checked for a 'convert-to-string' method, or the string port could be checked for a 'convert-from-integer' method.

3.3.3 Method-based Inspection

When trying to understand a program, for example when debugging, immediate interactive inspection of values in a dataflow graph is desirable. This gives the user a more clear image of the data flowing through the graph. Two kinds of inspections are possible; printing the value of the port as a string, or showing a window with the value in some widget.

There are three methods that come into play for this functionality:

- convert-to-string: This is the basic fallback method if neither of the other two has been implemented. It will return a stringified version of the value in the port, which can then be printed or popped up in a little window.
- print: This is the default method used for printing. It should print the value of the port.
- display: This is the default method used for widgetization. It should return a widget illustrating the value of the port. This widget will then be shown on the screen in a separate window.

Of course, one should consider including a save method in this, which would save the value of the port to a file. Though not directly related to inspection, it seems a useful piece of functionality anyway.

3.3.4 Packages

The language as described above is inextensible, apart from the definition of procedures. It does, however, seem applicable to many domains of analysis, so it would seem appropriate to introduce some functionality that would ease incorporating and distributing new types and operations.

This is done using the concept of 'package'. A package contains additional functionality in the form of extra port types and node types and all that goes with them, and can be added at any time. There is one basic package, described in section 3.3.6, which is always available. This package contains truly basic functionality, such as a while loop and a procedure call. One other package will be described in section 3.3.6, and that is the spatial package, which contains some functionality for working with spatial data.

The user should be able to add existing packages to the visual environment during runtime, and saved programs should include a list of required packages.

3.3.5 Visual Type Abstraction

Using packages as a method to extend the visual language has a drawback in that it requires one to leave the visual language to actually create the package and program the new port types and operations in a conventional programming language. That is, unless some provision is made for visual type abstraction. This is a point that is acknowledged in literature [BBB^{+}95]; visual type abstraction is needed for scalability.

Schemes for visual type abstraction have been proposed for visual programming languages before. [BBB^{+}95] presents one for a form-based language and [CDZ95] presents an object-oriented one for Vipr. A scheme for visual type abstraction suitable for integration into GROAn is presented in this section.

A 'type', in this context, consists of three parts, each of which will have to be covered by this scheme:

Data: This is the actual data stored in the port. If one wanted to create a three dimensional (integer) vector type, for example, it would be three integers.

Given that basic data types have been implemented, such as integers, floating point numbers, strings and arrays, a visual interface for the composition of types seems quite feasible.

Methods: These are the actual methods associated with the object that is the port. They include ones mentioned above, such as convert-to-string and print, but also ones such as accept-connect, which determines whether or not connections will be accepted. Of course, it should also be possible to define completely new methods.

There are two things we need to be concerned with, here. One is the method's signature, the other is the method's body. The signature consists of three parts: a possible return value, the first argument, which is the port type we're in the process of defining, and the other arguments, if any. All of these can be defined in terms of port types. The method body is just a graph.

Of course, for methods such as accept-connect, special measures will have to be taken when one is editing the method graph and the method gets called. Also, for the implementation of the display method in this way, widgets will have to be implemented as basic types.

Private nodes: Nodes that have this port type as one of their input or output ports and that need access to the internals of this type.

For the last two of those, two new node types will be needed; one to access the internals of a type and one for the reverse, i.e. constructing a new type from its internals. These two node types, which may be created automatically from the data composition described in the first item, would of course not be generally available, but only to methods and private nodes. For the constructor node, a check graph may be desirable, to check certain conditions on the inputs of the constructor node; e.g. for a three dimensional plane constructor, one may wish to check that the three points given are not collinear.

3.3.6 Node and Port Types

Here a brief overview will be given of node and port types. Not all have been implemented yet. Some of the descriptions of nodes will say that some things may be specified using a parameter. In the visual programming environment, this means that the user can cause a dialog to appear allowing him to manually enter the value of this parameter.

The 'Basic' Package

The 'basic' package contains node and port types required for fundamental functionality. The following port types should (at least) be supported:

Scalar: A base type for scalar values that only implements a default copy method.

Integer: An integer number (derives from scalar).

Float: A floating point number (derives from scalar).

Boolean: A true/false value (derives from scalar).

String: A character string.

Array-of: An array of any number of elements. The elements must all have the same type.

Any: A wildcard port type that, by default, accepts any connection. Any number of these can be grouped together so that once one port type from the group has accepted a connection, all others in the group will only accept connections of the same type.

Any-Array: Matches any kind of array.

Printable: A wildcard port type that really only makes sense as input port. It accepts only connections with ports that implement a `print` or a `convert-to-string` method. Derives from any.

The following node types should (at least) be supported:

Integer: An integer constant. Has no inputs and one integer output.

Float: A floating point constant. Has no inputs and one floating point output.

Boolean: A boolean constant. Has no inputs and one boolean output.

String: A string constant. Has no inputs and one string output.

Expression: Compute the value of an expression. Has any number of integer/float inputs, and one integer/float output. Has one parameter which is a string representation of the expression to be computed, possibly allowing the use of common functions such as sine and logarithm. The type of output has to be investigated further; this should possibly be another parameter.

Boolean Expression: Much like the above. Compute the value of a boolean expression. Has any number of integer/float/boolean inputs, and one boolean output. Again allows the specification of the expression to be computed using a string representation, this time also involving logical operators such as 'and' and 'or'. Also allows the use of comparison operators such as equal and less than.

String Split: Split a string on a substring. Has two string inputs and one array of strings output. The first string output is split on each occurrence of the second string output, creating an array of strings as output. Should also allow the substring to be specified as parameter.

String Match: Checks whether a string matches a pattern. Has two string inputs and one boolean output. Checks whether the first string input matches the pattern given in the second string input, outputs true if it does, false otherwise. Should also allow the pattern to be specified as parameter.

String Find: As the above, except the pattern may only be specified as a parameter, not through an input, and has a number of additional outputs, depending on the pattern specified. If the pattern contains subpatterns, the strings matching these subpatterns will be output on the corresponding outputs, otherwise empty strings will be output there.

StringConcat: Concatenates any number of strings. Has any number of string inputs and one string output.

Print: Prints any number of objects. Has any number of printable inputs and no outputs.

Array Create: Create a new array. Has any number of inputs of type any (linked in a group so the types are identical), and one output of type array of whatever type the input has.

Array Size: Compute the number of elements in an array. Has one any-array type as input and one integer as output.

Array Index: Returns a certain element from an array. Has one any-array type and one integer as inputs and one output matching the type of the elements in the array.

Source: When executing a subgraph, the input parameters for the subgraph are copied to the output ports of the source node of the subgraph. Has no inputs and a variable number of outputs of variable types.

Sink: After a subgraph has been executed, the input parameters of its sink node contain the output values of the subgraph. Has a variable number of inputs of variable types and no outputs.

While: A node containing two subgraphs, a condition and a loop body, behaving as a while loop. Has any number of in/outputs of any type. The condition subgraph has a set of inputs identical to that of the node, and only one boolean output. The loop body graph has a set of inputs and outputs identical to that of the node.

First the condition subgraph is run. If the value produced by that graph is true, the body graph is run. Then the whole thing starts over again, by running the condition graph again. Of course, the proper copies from the proper ports to the proper ports are done at the proper times.

A maximum loop count may also be set, to attempt to detect infinite loops; the node will abort with an error if the maximum loop count is reached and this functionality has been enabled.

If: A node containing three subgraphs, one condition subgraph, one true subgraph and one false subgraph, behaving as an if-statement. Has any number of in/outputs of any type. The condition subgraph has a set of inputs identical to that of the node, and one boolean output. The true and false subgraphs both have sets of in- and outputs identical to that of the node.

First the condition subgraph is run. If the value produced by that is true, the true subgraph is run, otherwise the false subgraph is run. Again, values are copied from and to ports at the right times.

Subroutine: A node containing a reference to one subgraph as explained in section 3.3.1. This node has any number of inputs and any number of outputs, all of any type. Of course, the numbers and types of inputs and outputs is determined by the subgraph being referenced. This node copies the values on its input ports to the input ports of the subgraph, runs the subgraph and finally copies the values on the output ports of the subgraph to its output ports.

At the moment there is no support for recursive subroutines. This would involve saving all the port values of all the ports of each subgraph when it starts executing, and restoring them after it is done.

Foreach: A node containing one subgraph. It has at least one input, an array, and any number of other inputs of any type. The outputs match the inputs minus the array. The subgraph is executed once for each element of the array, copying input and output values appropriately.

Shell Command: Run a shell command. Has one string input and two string outputs and one integer output. The string input is the command to be executed plus any arguments. The first string output is the standard output produced by the command, the second one is the standard error produced by the command. The integer output is the return code produced by the command.

The 'Spatial' Package

As has been noted before, GROAn is meant to be used for the analysis of uncertain spatial data. The 'spatial' package is meant to contain support for this kind of analyses, according to the rough model described in sections 4.2 and 5. The following port types should (at least) be supported:

Rough Polygons: One class of rough polygon data.

Rough Lines: One class of rough line data.

Rough Points: One class of rough point data.

Rough Class: One class of rough data, polygon, line or point.

Rough Truth: A rough truth value, being yes, no or maybe.

A layer of data may be formed by grouping a number of classes together in an array. Of course, certain conditions on overlap must be satisfied for it to be a proper layer; those are discussed in section 4.2.1.

Since this package is based on the ROSE library [Sch95], the spatial operators implemented in that library will be supplied here as nodes (for as far as they can be extended to rough data). Those nodes will not all be listed here; suffice it to say that they include operators such as intersection, union, difference, operators to compute the number of points in a set, the size of the area covered by a polygonal class and so on.

That doesn't cover all our needs, however. At least the following node types should also be supported. Where the notation *Layer or *Class is used, it means there are three variants of that node, one for polygons, one for lines and one for points. The same * may be used in the description, too, and then stands for the same thing.

Rough2Boolean: Has one Rough Truth input and three boolean outputs. Depending on the value of the input, exactly one of the outputs will get the value true, the other ones will get the value false. To be used for decision making.

Load *Layer: Load a layer of * data from a file. Has one output, type array of Rough *, and one input, type string. The string is the name of the file to load from; this may also be specified as a parameter.

Save *Layer: Save a layer of * data to a file. Has two inputs, one type Rough *, and one type string. The string is the name of the file to save to; this may also be specified as a parameter.

Visualizer: Show a number of rough layers. Has any number of inputs, all of type array of rough class. Allows one to selectively view any or all of the rough layers, zoom in and out, change colors, etc.

Reclass: Reclassify a layer of data. One rough layer input, one rough layer output. The reclassification rules are given in a file; the filename may be specified as parameter.

Merge: Merge two rough layers into one, according to the rules given in section 6.2. Has two rough layer inputs, one rough layer output.

TotalArea: Compute the union of all the classes in a layer. Has one rough layer input and one rough class output. Of course, this is just a convenience node.

3.4 Visual Layout

Since this is a visual programming language, visual aspects of the language are also important. They will be covered in this section.

3.4.1 Layout of Nodes

Since the graphs to be layed out are directed acyclic graphs, it is easy to sort the nodes of the graph by 'level', the maximum path length to that node from a node with no inputs. These levels can be used as a basis for layout, with the lowest levels closer to the top of the screen, so that data flows downwards. For subgraphs, though, it turned out to be desirable to treat the Source and Sink nodes (see section 3.3.6) separately, to insure that they were always alone on the first (well, zeroth) respectively last level.

Of course, this means that the graph will have to be re-sorted every time a new connection is made. One could also re-sort every time a connection is broken, but that would seem to unnecessarily disturb the user's mental image of the graph. The user always has the option to manually force a resort of the graph, for example at the end of an editing session.

This leaves considerable freedom *within* the levels. Rather than have this be computer-controlled, trying to minimize edge crossings or some such, the choice has been made to let the user control this aspect of the layout. This in-level ordering would be easily disturbed, though, as resorting occurs often while editing, and once the node has been moved to another level, the ordering within the previous level would be lost.

The implemented solution to this problem is that each node, as it gets inserted into a graph, gets assigned a unique (within the graph) sequential number. These numbers are then used for sorting within a level.

As for rearrangements of this sorting by the user, they are composed of switchings. Say the user wants to switch node A with node B, both of which are in the same level. We can assume, without loss of generality that n_A, the sorting number of node A, is less than n_B, the sorting number of node B. What will happen is that the nodes with sorting numbers n_A through $n_B - 1$ will have their sorting numbers increased by one, and node B will get sorting number n_A. This was done in an attempt to preserve as many interrelations as possible.

There is yet another degree of freedom in this arrangement, one which has not been addressed in any way in the current implementation. This degree of freedom occurs when a node is not tightly constrained from below, for example in the partial graph in figure 3.10. The node on the right in that figure can easily be moved down by one level without influencing the rest of the graph.

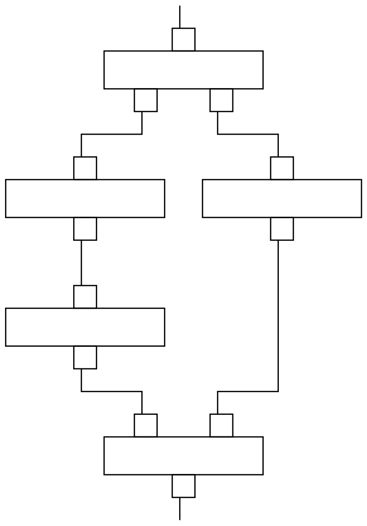

Figure 3.10. A sketch of a partial graph with layout freedom.

Exploiting this degree of freedom may be very interesting, however, especially if one considers the fact that nodes containing subgraphs that are being shown may be substantially bigger than nodes not containing any subgraphs. Special considerations for such extremely large nodes may significantly improve layout. One could try to move other nodes away from the level of the big node by exploiting the degree of freedom discussed above, or if the big node itself has that degree of freedom, one might consider letting the big node span multiple levels.

3.4.2 Layout of Edges

The edge layout problem is something that has hardly been addressed at all in the current implementation. An edge from an output port to an input port is just drawn as a straight line, which may be partially hidden by nodes. This is clearly unacceptable in any non-prototype system. For a non-prototype system, a modified train-seat-allocation solution could be used.

The train seat allocation problem is the problem where, given a train with a variable number of seats and a number of passengers who wish to travel from a certain station to a certain other station, one is asked to allocate seats to passengers, (initially) such that no passenger has to change seats during the journey and the number of used seats is minimized. From a theoretical point of view, this is an easy problem, which is solved by sorting the passengers by starting station, and then allocating seats on a first-fit basis.

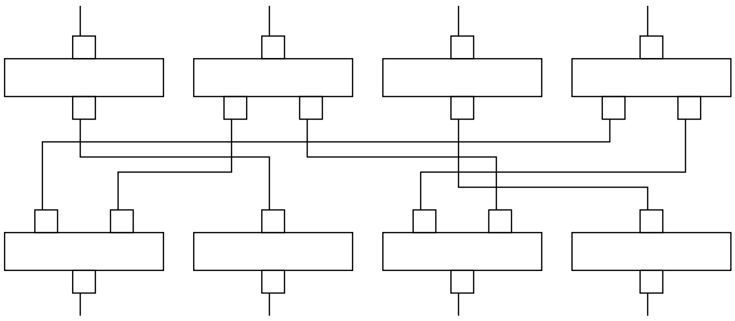

Figure 3.11. A partial graph with edges layed out.

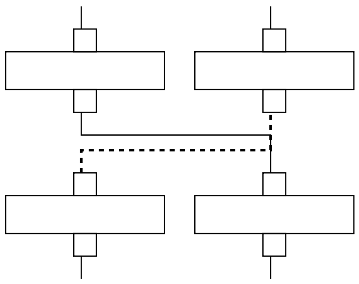

Figure 3.12. A partial graph with badly layed out edges.

Let us first look at edges which span only one level; a simple example is shown in figure 3.11. These edges start at a certain port, with a certain x coordinate on the screen, and end at another port, at another x coordinate; these are the stations. In terms of trains and passengers, we will assume that this edge boards the train at the port with lower x coordinate, and gets off at the other port, the one with the higher x coordinate. Note that this direction may be the reverse from the direction of data flow. There are twelve stations in figure 3.11, and four seats are required. If many seats are needed, the levels below this may have to be moved down a bit, to make room.

Special problems occurs when one input and one output port have exactly the same x coordinates (or the x coordinates differ only by a very small amount). In that case, unless we adapt the layout algorithm, we can get confusing layouts such as the one in figure 3.12, where one edge is dashed in bold for clarity. The confusing thing in this layout is on the right-hand side. There, it becomes hard to visually follow the edges because they partially overlap. Clearly, the situation on the left side of that figure requires the dashed edge to be below the other; the converse is true for the right side. As figure 3.12

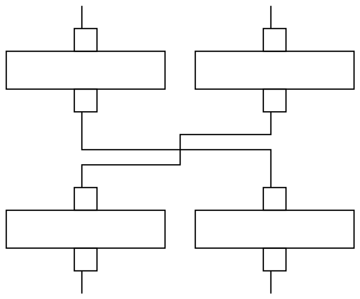

Figure 3.13. An improved layout of figure 3.12.

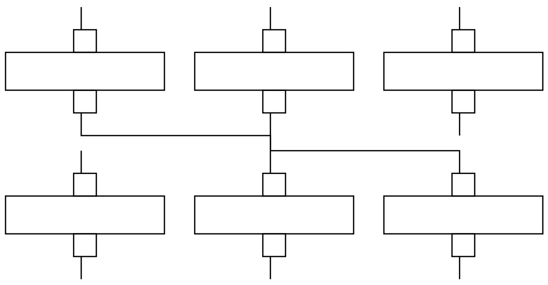

Figure 3.14. Another partial graph with badly layed out edges.

clearly shows, just adding more seats to the train is not going to solve the problem.

Layouts such as those in figure 3.12 are deemed unacceptable. Edges should not overlap unless they go to or come from the same port. This means that we will have to require our passengers to switch seats during the trip. In cases such as this, where there is horizontal as well as vertical overlap, a switch will be introduced. The improved layout is given in figure 3.13.

Another possible source of confusion is shown in figure 3.14. There, two edges overlap vertically, but not horizontally. In this case, we do not have to introduce any switches, we just switch the seat allocations for those two edges. Of course, if either or both of those seats are not free, one or two new seats may have to be introduced. The results are shown in figure 3.15.

A more complicated product of this algorithm is given in figure 3.16. Clearly, this is not a very nice layout. The long parallel horizontal lines are quite confusing to the eye. Of course, by moving the nodes in the lower level around a bit, one can easily improve the situation to that shown in figure 3.17. That may have repercussions on the layout further down in the graph, but it does show that it is not possible to say how common situations like the on in

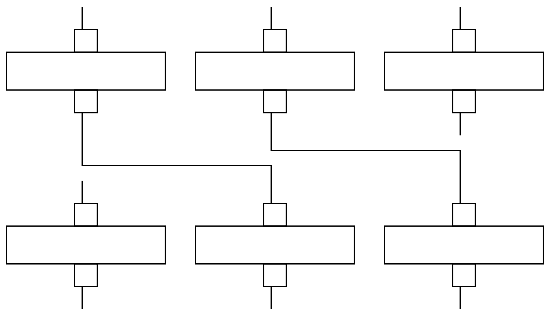

Figure 3.15. An improved layout of figure 3.14.

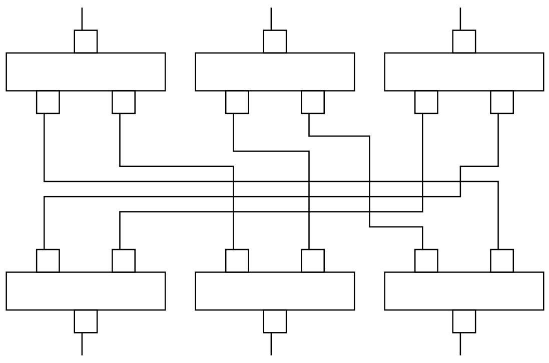

Figure 3.16. A bigger partial graph.

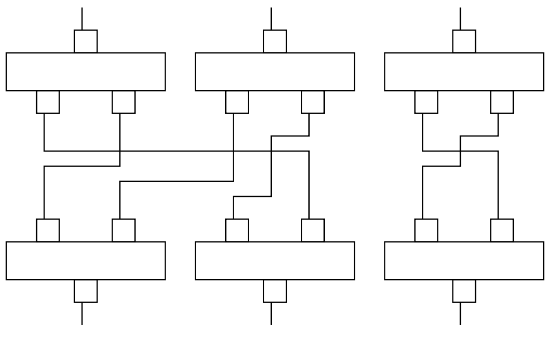

Figure 3.17. A rearranged version of figure 3.16.

figure 3.16 are. Regardless, further research on this subject seems warranted, but falls outside the scope of this work.

Multi-level edges are somewhat more complicated, because one also has to decide between which nodes the edge goes down to the next level. Here it would seem preferable to have as little weaving back and forth as possible, and one could even look into minimizing intersections. Also here, if many multi-level edges go down to the next level between one specific pair of nodes, one may have to move some of the nodes a bit to make space.

Finally, the process should be incremental, meaning that local changes should have only local effects, and these local effects should be minimal in some sense, so as to minimize the impact on the mental map of the user. This may mean that after a number of these incremental changes, the solution is no longer optimal, in the sense that more train seats are used than needed. In this case, one can reconstruct the solution from scratch when the user manually requests a resorting of the graph, since then the user will expect to have to rearrange his mental map anyway.

3.5 State of the Implementation

Even though, strictly, the programming environment is not a part of this thesis, some words should be said about it here. At the time of writing, an implementation of part of the ideas presented above does exist, but not all the ideas have been implemented.

A framework for packages is in place, and two packages exist, the basic and the spatial packages. Most of the node types listed under the basic package have been implemented, with the exception of some string operations, the array operations and the subroutine node. The user can add and remove nodes and edges from the program. The environment can successfully run the "Hello world!" program given in figure 3.2, and the "Factorial" program given in figure 3.6. The spatial package is currently very incomplete, only a load layer operation exists.

Two layout algorithms are available, although one has to choose between these two at compile time. The simpler one draws edges as straight lines. The other one is an almost complete implementation of the edge layout algorithm specified above.

The printable port and the print node exist, proving the concept of method-based polymorphism of types. Simple method-based inspection has also been implemented, based on a check for the existence of the `print` method.

3.6 Conclusions

This chapter has described a dataflow-based visual programming language with novel modifications that offers some advantages over traditional dataflow languages (natural detail hiding, more structured), while preserving the advantages that traditional dataflow languages already have (natural mapping to analysis, simplicity, concreteness, explicitness and responsiveness). Some ideas about how to extend this language and about how to organize the layout have also been presented, along with other design ideas.

In the future, more research on the question about expressive power raised in section 3.2.4 seems warranted. The layout of the graph is also something that may be investigated in more detail.

Chapter 4

Representation of Uncertainty

The previous chapters have talked about visual programming languages and the design of a specific visual programming language for use with geographic data. Apart from the statement in chapter 1 that a vector representation was desired, they have not, however, discussed the representation of that geographic data. This chapter is about the need for representing the uncertainty present in such geographic data, some possible ways in which to represent this uncertainty, and the repercussions this has on the representation of the geographic data.

4.1 Why Is Uncertainty Important?

Before deciding how to represent uncertainty, there is one question that needs to be asked: "Why bother?" Why should uncertainty be represented at all in situations like this? Other people have asked this question before, and representation of uncertainty is becoming more and more of a concern [Goo93].

One thing that should be realized is that people have very high confidence in results produced by a computer. After all, everybody knows that computers are very good at calculations with exact numbers, and (almost) never make mistakes. And, assuming the modeling is accurate, the computer-calculated results will be precise, *if* the input data is accurate.

And now we come to the core of the question. Let us look at some simple geographic data, for example a layer of data that tells us which areas in our neighborhood are forest, and which aren't. Assuming that this layer doesn't have any representation of uncertainty, it will give us a sharp delineation between the areas which are forest and those which aren't. Now, walk up to any forest and try to draw a line on the ground where the edge of the forest is; you'll see that this is not an easy process, as it is not possible to sharply define the edge.

Nor is this the only possible source of uncertainty. Any measurement in the field, be it an aerial photograph or a radar sensor, has a certain uncertainty due to the sensor(s) being used; this is a problem that only increases if data from several (types of) sensors has to be fused together. Natural features change over time. Other sources are possible, which can contribute uncertainty along each of the tree dimensions that define a geographical entity: theme, space and time [LV92]. Various combinations and interactions of these dimensions create other types of uncertainty, such as uncertainty on the number of objects and their topological relationships [GGS92], but these are considered effects of the primary dimensions.

So, the uncertainty is there, whether we represent it or not. And representing uncertainty in our input data is a prerequisite for getting an estimate of the uncertainty in output data. The conclusion that we should represent it seems inevitable. Then the question remains, how should we represent it?

Of course, uncertainty may be expressed in many ways. In the following sections three methods will be described and evaluated. In section 4.2, a novel approach based on rough sets will be described. Following that, in section 4.3 the more traditional approach based on fuzzy sets will be described, and a short look at qualitative approaches is given in section 4.4. An evaluation of these possibilities follows in section 4.5.

4.2 Rough Sets

The theory of rough sets is based on the idea of classification with limited data [Paw91]. For example, let us assume that we wish to classify cars into trucks or personal vehicles, and that we are supposed to do this based on some data on each of the cars. For each car, we know whether it is heavy or light and whether it is big or small. If the car is heavy and big, we are sure it is a truck. Conversely, if it is light and small, we are sure it is a personal vehicle. But what to do if the car is heavy and small? Then it seems impossible to decide either way; we know it is either a truck or a personal vehicle, but we do not know which one. This kind of reasoning can be described with rough sets. Also, recent work [Sch95] has shown clear advantages of using a rough set approach in dealing with imprecise spatial data.

A *rough set* [Paw91] is a pair $(\underline{X}, \overline{X})$ of standard sets, the *lower approximation* and the *upper approximation*. In the representation chosen here, $\underline{X} \subseteq \overline{X}$. The meaning of these two sets is that if a data point lies in $\underline{X}$, it is sure that the point is in the rough set, if a data point lies in $\overline{X} - \underline{X}$, it is unsure whether or not the point is in the rough set, and if a data point is outside $\overline{X}$, it is sure that the point is not in the rough set.

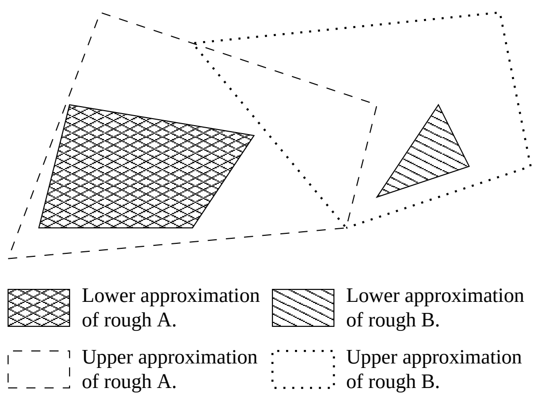
Lower approximation of rough A.
Lower approximation of rough B.
Upper approximation of rough A.
Upper approximation of rough B.

Figure 4.1. A rough classification

$\overline{X} - \underline{X}$ will often be called the *area of uncertainty* of a rough set. As opposed to rough sets, standard sets are often called *crisp*, a term that also applies to a rough set where $\underline{X} = \overline{X}$, which implies an empty area of uncertainty. Conversely, a rough set with an empty lower approximation and a non-empty area of uncertainty can be called *completely rough.*

As shown in [Dün97], the basic set operations, union, intersection and negation, can easily be extended to rough sets. Union and intersection as follows: $(\underline{X}, \overline{X}) \cup (\underline{Y}, \overline{Y}) = (\underline{X} \cup \underline{Y}, \overline{X} \cup \overline{Y})$, $(\underline{X}, \overline{X}) \cap (\underline{Y}, \overline{Y}) = (\underline{X} \cap \underline{Y}, \overline{X} \cap \overline{Y})$. Negation can be extended in two ways: $(\underline{X}, \overline{X})^* = (-\overline{X}, -\overline{X})$ and $(\underline{X}, \overline{X})^+ = (-\underline{X}, -\underline{X})$. He does not list the more intuitive extension of negation: $(\underline{X}, \overline{X})^- = (-\overline{X}, -\underline{X})$.

4.2.1 Rough Classification

A standard, or crisp classification C consists of a number of classes $X_i, i \in I$, each of which is a crisp set. I will be called the *index set* of a classification. For a crisp classification, $\forall i \forall j \neq i : X_i \cap X_j = \emptyset$: the classes are pairwise disjoint. We will define $|C| = |\bigcup X_i|$, which, in this case, is equal to $\sum |X_i|$.

Likewise, a *rough classification* R consists of a number of *rough classes* $X_i = (\underline{X}_i, \overline{X}_i)$, $i \in I$, each of which is a rough set, and I is the *index set* of the classification. In a rough classification, however, we would like to be able to express our uncertainty about which class, if any, a certain area belongs to. Therefore, instead of pairwise disjointness, we impose only the following restriction: $\forall i \forall j \neq i : \underline{X}_i \cap \overline{X}_j = \emptyset$. Of course, since $\underline{X} \subseteq \overline{X}$, this implies that $\forall i \forall j \neq i : \underline{X}_i \cap \underline{X}_j = \emptyset$. We do not impose such a restriction on pairs of upper approximations, though, so there may be areas where two or more upper approximations overlap, but none where any lower approximation overlaps with anything else than the corresponding upper approximation. We will define $|R| = |\bigcup \overline{X}_i|$, which, in this case, is *not* equal to $\sum |\overline{X}_i|$. See figure 4.1 for an example of a rough classification.

In this way, we have expressed uncertainty along two of the fundamental dimensions:

- Space: If, for a rough class $(\underline{X}, \overline{X})$, $\underline{X} \subset \overline{X}$, uncertainty about the spatial location of (part of) that class has been expressed.
- Theme: If a certain area is assigned to the area of uncertainty of more than one class, it is no longer certain to which class that area belongs. Thus, uncertainty of attribute value has been expressed.

It should, perhaps, be noted here that rough classification is a true extension of classical, or crisp classification, in that a crisp classification can be captured exactly in a rough classification. If, for each of the classes of the rough classification, the area of uncertainty is empty, the rough classification becomes, in effect, a crisp one.

4.3 Fuzzy Sets

The theory of fuzzy sets is based on the idea of non-sharply-defined concepts. For example, fill a glass to the rim with water. Is the glass full? Yes, it is. Now, take a small sip from the glass. Is it still full? You'll probably say that it is. Continue taking small sips from the glass and asking yourself whether the glass is still full. Eventually there will not be any water left in the glass; surely the glass is not full anymore. In-between, you'll end up with answers to your question from 'Yes, of course!' through 'Well, yes.' and 'I guess so.' to 'I don't think so.'. This kind of reasoning can be described with fuzzy sets.

According to [Zad93], a fuzzy set A, embedded in a crisp universe of discourse U, is characterized by a *membership function* $\mu_A : U \to [0, 1]$, with $\mu_A(u)$ representing the grade of membership of u in A. The *support* of A is $\{u \in U | \mu_A(u) > 0\}$. The *height* of A is the maximum of μ_A over U. If the height of A is unity, A is called *normal*, otherwise it is *subnormal*.

Also here, basic set operations extend. The membership function for the union of A and B, $A \cup B$, is $\max(\mu_A(u), \mu_B(u))$, that for the intersection of A and B, $A \cap B$, is $\min(\mu_A(u), \mu_B(u))$, and for the negation of A it is $1 - \mu_A(u)$.

4.4 Qualitative Representations

Another approach is to use a qualitative representation for the relationships between objects. An example of this is symbolic projection [CJ96], where two strings of symbols represent the spatial relationships between the projections of objects. This may be used to encode something similar to the 13 possible spatial relationships between intervals as given by [All83].

Since only the spatial relationships between the objects are represented, and since they are represented only in a qualitative manner, many different images may be represented with the same two strings of symbols. Therefore, if one were to use the symbolic strings as primary representation, that could be seen as a way to represent uncertainty.

[CJ96] also gives an overview of some other well-known approaches to qualitative spatial reasoning. These approaches are often not used completely by themselves, but are often combined with more quantitative approaches. However, there is a tradeoff between the qualitative and quantitative approaches. Qualitiative approaches may also be useful as indexing method and as a cognitive model.

4.5 Fuzzy vs. Rough

Given that qualitative approaches seem unsuitable as primary representation in an environment where quantitative results are desired, the two prime candidates here are the rough and fuzzy representations. So how do we choose between those two?

Vector representation: It was clear almost from the beginning that a vector representation for our spatial data would be chosen, as explained in chapter 1.

While a fuzzy representation is quite easy to implement in a raster environment, it is significantly more difficult in a vector environment. Some continuous function of x and y will have to be devised that accurately captures the membership function for the fuzzy class we are trying to define. While this may seem possible, the complexity of a straightforward approach soon becomes prohibitive once we start overlaying different classes and doing union and intersection operations on them.

There is some recent progress in this field, though. [WH96] shows methods to use fuzzy sets to represent both polygon boundary and attribute uncertainty. [Mol96] discusses fuzzy spatial objects using a semantic formalism, but expresses doubt whether this can readily be handled using existing tools.

A rough representation, on the other hand, seems suited for both approaches. For a vector-based approach, $\overline{X}$ and $\underline{X}$ may both be defined using vectors; of course one must make sure that $\underline{X} \subseteq \overline{X}$ at all times.

For a raster-based approach, one can use a two-bit number to keep track of membership in the rough set. The simplest approach is perhaps to use one bit for membership in $\overline{X}$ and one bit for membership in $\underline{X}$. Again, the requirement of $\underline{X} \subseteq \overline{X}$ means that one of these four combinations should never be used. This can be ensured by using values zero, one and two directly in the obvious manner. To save memory, at the cost of execution time one may combine a number of these membership values by using base three to encode them in one number. (One can store five membership values in a byte instead of four, this way.)

The point here is that a fuzzy representation does not work well at all for a vector-based approach; a rough representation does.

Membership function: It is clear that the membership function used in fuzzy sets gives more information than the two sets used in rough sets. In fact, some parts of the basic definitions of rough sets correspond to the basic definitions of fuzzy sets with a membership function that only takes values in $\{0, 0.5, 1\}$.

A problem with membership functions, though, is that this information is not always available. Given a forest, how does the gradual transition between no forest and forest translate to a membership function? Aren't those values somewhat arbitrary? Even though formalisms exist for choosing membership functions [BM98], these formalisms still feature parameters that have to be set. As [BM98] puts it: "... the greatest difficulties come with choosing the values of the control parameters ...".

So, certainly in the cases where such information is not available, it is better to continue without it than to arbitrarily make it up. Especially if one realizes that such an arbitrary choice may have an effect on the final results of the analysis.

Accuracy measurement: Assessing the accuracy of fuzzy data seems difficult; more so than when using rough data. An approach for this using rough data is given in the next chapter.

4.6 Conclusions

The uncertainty present in spatial data should be represented, because that may allow us to give estimates of the uncertainty in our end results. This is especially important in an environment where inexperienced users perform analyses, such as in the visual language discussed in earlier chapters of this thesis. Furthermore, for various reasons, a rough-set based approach seems most suitable.

Chapter 5

Accuracy Assessment of Rough Data

Given that our spatial data is uncertain, we will surely want to know how uncertain it is. This chapter covers just that subject. After a short introduction to accuracy assessment, several things are measured about various kinds of spatial data. First off, we discuss how to measure the degree of roughness in rough data. After that, accuracy assessment of spatial data is covered going from the most specific to the most general cases: crisp data compared to crisp data, rough data compared to crisp data and finally rough data compared to rough data.

5.1 What is Accuracy Assessment?

Given that all spatial data contains uncertainty, the inevitable question crops up as to how uncertain a specific set of data is. This is the question that accuracy assessment tries to answer, but it is not a question that is easily answered, and many answers are possible; see for example the methods listed in [Con91].

What all these methods have in common is that they assess the accuracy of one classification, called the *assessed* classification, as compared to the *reference* classification, which is assumed to be accurate. Different formulas are used to measure different kinds of accuracy about the data. As [Con91] puts it: "It is not possible to give clearcut rules as to when each measure should be used.".

5.2 Measuring the Roughness of Data

There are certain things about rough spatial data that can be measured even without introducing a reference classification. Specifically, we can measure how much overlap there is in the rough classification and we can measure how much of the rough classification is actually part of a lower approximation. We will measure these things for a rough classification R, consisting of the rough classes $X_i = (\underline{X}_i, \overline{X}_i), i \in I$. Examples of both these measures may be found in section 6.3.

To measure the amount of overlap in a rough classification, we can use the overlap measure, which can be computed as follows: $M_o = (\sum |\overline{X}_i| - |R|)/|R|$. Since all the area covered by the rough classification is included at least once in $\sum |\overline{X}_i|$, $M_o \geq 0$. If $M_o = 0$, there is no intersection between any of the rough classes. If $M_o > 0$, $\exists i \exists j \neq i : (\overline{X}_i \cap \overline{X}_j) \neq \emptyset$ (since the lower approximations are pairwise disjoint, this area of intersection must be in the area of uncertainty of said rough classes). M_o can grow if either the total area of intersection grows or (parts of) this area are shared by more rough classes. The upper bound for M_o is the number of classes in the rough classification minus one, as the maximum overlap occurs when all classes contain the whole area of the classification.

To measure how much of a rough classification is actually part of a lower approximation, we can use the overall crispness measure, which can be computed as follows: $M_c = \sum |\underline{X}_i|/|R|$. Obviously, $M_c \geq 0$, and, since the lower approximations are pairwise disjoint, $M_c \leq 1$. If $M_c = 0$, all the lower approximations of the classes of the rough classification are empty, making the classification completely rough. If, on the other hand, $M_c = 1$, the areas of uncertainty of the classes of the rough classification are empty, and the classification is in fact crisp. This can easily be extended to a class based measure (one measure for each rough class), instead of an overall measure.

5.3 Accuracy Assessment of Certain Data

This section gives a brief overview of the methodology and the various measures given in [Con91]. These will all turn out to be specific cases of the more general approaches described below.

We will consider the case where two crisp classifications A and B are being compared. A consists of the classes $X_i, i \in I$, while B consists of the classes $Y_j, j \in J$. Much of the following will only be valid if $|I| = |J|$ (let us define $N = |I|$), and will probably only make sense if, in fact, $I = J$.

5.3.1 Error matrices

When comparing two crisp classifications, the standard first step is to compute an error matrix. The various measures can then be computed from this matrix.

The error matrix is defined as an N by N matrix with elements $x_{i,j}$ having values $x_{i,j} = |X_i \cap Y_j|$. We will assume that $\bigcup_i X_i = \bigcup_j Y_j$, i.e. that the two classifications cover exactly the same area. Because of that and the fact that all the X_i are pairwise disjoint, and all the Y_j are also pairwise disjoint, the error matrix has the following three properties:

- The *row-sum* property: $\sum_j x_{i,j} = |X_i|$, i.e. the sum of the elements in a single row of the matrix is the area of the corresponding class of A.
- The *column-sum* property: $\sum_i x_{i,j} = |Y_j|$, i.e. the sum of the elements in a single column of the matrix is the area of the corresponding class of B.
- The *total-sum* property: $\sum_{i,j} x_{i,j} = |A|$, i.e. the sum of all the elements of the matrix is the area covered by either of the classifications.

5.3.2 Some measures

Overall accuracy is defined as the total match between the two classifications divided by the total area of the classifications, i.e.:

$$A_o = \sum_i x_{i,i}/|A|$$

This can be split up according to the classes in the classifications, but in that case one is left with the choice of dividing by the column total or by the row total. Assuming that the reference classification is associated with the columns of the matrix, dividing by the column total gives the *omission error*, also called *producer's accuracy*. Dividing by the row total gives the *commission error*, also called *user's accuracy*. In other 'words':

$$O_j = x_{j,j}/\sum_i x_{i,j}, \text{ and } C_i = x_{i,i}/\sum_j x_{i,j}$$

A more complicated statistical measure, which makes various assumptions about the input data, is $\hat{K}$ [Con91]:

$$\hat{K} = \frac{|A|\sum_i x_{i,i} - \sum_i(\sum_j x_{i,j} \cdot \sum_j x_{j,i})}{|A|^2 - \sum_i(\sum_j x_{i,j} \cdot \sum_j x_{j,i})}$$

Please note that, apart from the naming of omission and commission errors, these three measures are completely symmetrical with respect to the columns and rows of the matrix, i.e. transposing the matrix, by switching A and B, will have no effect on the computed value.

5.4 Accuracy Assessment of Rough Data Compared to Crisp Data

We first discuss how to extend error matrices to comparing rough and crisp classifications, and then we discuss the various measures we can compute from such a matrix. We also introduce a method by which we may apply any measure that can be applied to the comparison of two crisp classifications.

Here, we will assume that we are comparing a rough classification, R, with a crisp one, C. R consists of the rough classes $(\underline{X}_j, \overline{X}_j), j \in J$ and C consists of the crisp classes $Y_i, i \in I$. Again, we will assume $|I| = |J|$ (and hope for $I = J$) and define $N = |I|$. We will also assume again that $\bigcup \overline{X}_j = \bigcup Y_i$.

5.4.1 Error Matrix Extension

The extended error matrix will be of size $2N \times N$, having elements $x_{i,k}$. The definition of $x_{i,k}$ depends on k. If k is odd, $x_{i,2j-1} = x_{i,j+} = |\underline{X}_j \cap Y_i|$. Otherwise, k is even, and $x_{i,2j} = x_{i,j?} = |(\overline{X}_j - \underline{X}_j) \cap Y_i|$. When we talk about the diagonal of this matrix, we will mean the elements $x_{i,i+}$ and the elements $x_{i,i?}$.

The column-sum property still holds: The sum of all the elements in a column is exactly the area of the corresponding part (lower approximation or area of uncertainty, as the case may be) of the corresponding rough class. However, the row-sum and total-sum properties do *not* hold any longer. If we compute the sum of a row of the matrix, $\sum_k x_{i,k}$, we do not, as we would like, get $|\bigcup_j \overline{X}_j \cap Y_i|$, but we get $\sum_j |\overline{X}_j \cap Y_i|$, thus counting all the overlapped areas multiple times, and similarly for the total sum of all the matrix elements.

5.4.2 The relative crispness measure

Assuming that the crisp classification is the reference one, the relative crispness measure compares the crispness of a rough classification in the areas where it corresponds to the crisp classification (where it is 'right') to its crispness in areas where it does not correspond to the crisp classification (where it is 'wrong').

If, on the other hand, the crisp classification is the one that is being assessed, it measures whether the crisp classification is more correct in areas where the rough classification is certain than in areas where it is uncertain.

Given the following two definitions:

$$D = \begin{cases} \frac{\sum x_{i,i+}}{\sum (x_{i,i+} + x_{i,i?})} & \text{If } \sum (x_{i,i+} + x_{i,i?}) \neq 0. \\ 0 & \text{Otherwise.} \end{cases}$$

$$O = \begin{cases} \frac{\sum_{i \neq j} x_{i,j+}}{\sum_{i \neq j} (x_{i,j+} + x_{i,j?})} & \text{If } \sum_{i \neq j} (x_{i,j+} + x_{i,j?}) \neq 0. \\ 0 & \text{Otherwise.} \end{cases}$$

D measures the crispness in the diagonal elements of the matrix, whereas O measures the crispness in the off-diagonal elements. The crispness is not measured in a way compatible with M_c, since overlapping areas of uncertainty are counted twice.

The relative crispness measure can be computed as follows: $M_r = D - O$. Since $0 \leq D \leq 1$ and $0 \leq O \leq 1$, $-1 \leq M_r \leq 1$. If $M_r = -1$, that means that there is no crispness on the diagonal of the matrix, whereas there is perfect crispness off the diagonal of the matrix. Values between -1 and 0 mean that there is more crispness off the diagonal than there is on the diagonal. $M_r = 0$ means that there is no difference in crispness on or off the diagonal. Values between 0 and 1 mean that there is more crispness on the diagonal than there is off the diagonal. If $M_r = 1$, there is no crispness off the diagonal, while there is perfect crispness on the diagonal. Higher values of M_r are 'better' than lower values.

5.4.3 Overall accuracy

As stated above for the case of two crisp classifications, overall accuracy is the total correct area divided by the total area. This can be extended to the case of one crisp and one rough classification. However, we will not get one single answer, but, fittingly, an upper and a lower bound.

The upper and lower bounds for overall accuracy are as follows:

$$\sum x_{i,i+}/|R| \leq A_o \leq \sum (x_{i,i+} + x_{i,i?})/|R|$$

We can be sure that we are not counting anything twice, since we only use elements from the diagonal of the matrix. The areas corresponding to these diagonal elements are all subsets of different classes of C, and those are disjoint.

The way to arrive at the bounds given above, is to think of the ways the rough classification can be converted to a crisp one. If the assumption is made that it is permissible not to assign (parts of) areas of uncertainty to any crisp class at all, the bounds given above are valid and tight. If this assumption is not valid, we will have to resort to the procedure outlined below.

5.4.4 Error matrix parameterization

Since the row-sum and total-sum properties do not hold for extended error matrices, it is hard to directly apply traditional measures that involve elements beyond those on the diagonal. For this kind of global measures, we will convert the rough classification into a crisp one. However, doing so in a consistent fashion is not straightforward, since there may be overlap in the areas of uncertainty, and this information is missing from the matrix. So, we will have to go back to the rough classification, R, and convert this rough classification to a virtual crisp classification, V, on which (together with the crisp classification, C) we will base our crisp error matrix. V will have the same index set as R, namely J. So there will be a one to one correspondence between the classes $(\underline{X}_j, \overline{X}_j)$ of R and the classes Z_j of V.

Since there is no unique way to convert a rough classification to a crisp one, and the value of the measure that we want to compute depends on exactly how we do this, we will have to parameterize our matrix depending on to which classes of V, if any, we assign certain component areas. The component areas will be $Y_i \cap R^?_C$, the intersections of the classes of C with the rough components of R. We do not need to parameterize on the crisp components of R, since we have no choice where to assign them; they will become zeroth order terms in our parameterized matrix.

Given the definition $\Gamma_j = \{R^?_C | j \in C\}$, the N by N parameterized error matrix has elements $p_{i,j}$ given by:

$$p_{i,j} = |Y_i \cap \underline{X}_j| + \sum_{K \in \Gamma_j} (x_{i,j,K} \cdot |Y_i \cap K|)$$

The following two restrictions must be imposed upon the parameters:

$$0 \leq x_{i,j,K} \leq 1 \text{ and } \sum_j x_{i,j,K} \leq 1$$

This last inequality assumes that it is permissible not to assign some of the area covered by the areas of uncertainty to any class of V. If this is not permissible, the inequality becomes an equality: $\sum_j x_{i,j,K} = 1$.

Now we can apply any accuracy measure to this new matrix, and we will get a formula that gives us the value of the accuracy measure depending on exactly how we treat the areas of uncertainty in the rough classification. Theoretically, we could then maximize and minimize this equation within the bounds imposed on the parameters, and obtain the maximum and minimum values for the accuracy measure applied.

Let us look at an example. Omission and commission errors are the diagonal elements divided by, respectively, the sum of the column and the sum of the row. In any parameterized error matrix, this comes down to a quotient of two linear expressions, combined with the appropriate restrictions mentioned above.

For another example, let us look at $\hat{K}$. Each of the $x_{i,j}$ is a linear expression in an unknown number of parameters in our case, the same goes for $\sum_j x_{i,j}$ and $\sum_j x_{j,i}$. $|A|$, meanwhile, is perhaps best translated as $|C|$, which is constant. Thus, the whole comes down to the quotient of two quadratic expressions which should be optimized over a hypercube limited by some hyperplanes.

For both these examples, at least local minima and maxima can be found with software such as Matlab. Since finding the extremes of a quadratic expression over an N-dimensional box is known to be NP-complete [GJ79], optimizing the expression for $\hat{K}$ is at least that hard. We have been unable to find any mention about the complexity of finding the extremes of a quotient of two linear expressions, so have no such information about omission and commission errors.

5.5 Accuracy Assessment of Rough Data Compared to Rough Data

In this section, we will assume that we are comparing the rough classification R, consisting of $(\underline{X}_i, \overline{X}_i), i \in I$, with the rough classification S, consisting of $(\underline{Y}_j, \overline{Y}_j), j \in J$. We will make the usual assumptions $|I| = |J|$ and $\bigcup \overline{X}_i = \bigcup \overline{Y}_j$ and the definition $N = |I|$. We first discuss the error matrix extension for this case and its properties, then explore measures for it, and finally parameterize it to apply conventional measures.

5.5.1 Extended Error Matrices

When comparing two rough classifications with each other, we will use the *two-dimensionally extended error matrix*, or *2Deem*. This matrix will be of size $2N \times 2N$, having elements $x_{k,l}$. The definition of $x_{k,l}$ depends on both k and l. If both are odd, $x_{2i-1,2j-1} = x_{i+,j+} = |\underline{X}_i \cap \underline{Y}_j|$. If k is odd, but l is even, $x_{2i-1,2j} = x_{i+,j?} = |\underline{X}_i \cap (\overline{Y}_j - \underline{Y}_j)|$. Symmetrically, if k is even, but l is odd, $x_{2i,2j-1} = x_{i?,j+} = |(\overline{X}_i - \underline{X}_i) \cap \underline{Y}_j|$. Finally, if both are even, $x_{2i,2j} = x_{i?,j?} = |(\overline{X}_i - \underline{X}_i) \cap (\overline{Y}_j - \underline{Y}_j)|$. When we talk about the diagonal of this matrix, we will mean the four sets of elements $x_{i+,i+}$, $x_{i+,i?}$, $x_{i?,i+}$ and $x_{i?,i?}$.

For a 2Deem, none of the row-sum, column-sum and total-sum properties hold any longer, since neither the parts of classes associated with the individual columns nor the parts of classes associated with the individual rows are pairwise disjoint any more.

5.5.2 Overall accuracy

For this case, as well, an upper and a lower bound for overall accuracy can easily be computed from the error matrix: $\sum x_{i+,i+}/|R| \leq A_o \leq \sum(x_{i+,i+} + x_{i+,i?} + x_{i?,i+} + x_{i?,i?})/|R|$

5.5.3 Parameterized Error Matrices

To apply conventional crisp measures to this case, we can use an approach similar to the one we used for the rough-crisp case in section 5.4.4; constructing a parameterized error matrix. This time, we will parameterize on the intersections of a component of R with a component of S. We will repeat the definition $\Gamma_i = \{R^?_C | i \in C\}$ and add $\Delta_j = \{S^?_C | j \in C\}$. The elements of the $N \times N$ parameterized error matrix are now given by:

$$\begin{aligned} p_{i,j} &= \underline{X}_i \cap \underline{Y}_j + \sum_{K \in \Gamma_i} (x_{i,j,K,\underline{Y}_j} \cdot |K \cap \underline{Y}_j|) + \\ &\quad \sum_{L \in \Delta_j} (x_{i,j,\underline{X}_i,L} \cdot |\underline{X}_i \cap L|) + \sum_{K \in \Gamma_i} \sum_{L \in \Delta_j} (x_{i,j,K,L} \cdot |K \cap L|) \end{aligned}$$

Of course, still

$$0 \leq x_{i,j,K,L} \leq 1 \text{ and } \forall K, L : \sum_{i,j} x_{i,j,K,L} \leq 1$$

(Or make that $\ldots = 1$ if you do not believe in not assigning parts of areas of uncertainty to any class). Like in section 5.4.4, this parameterized error matrix is linear in its parameters, so the conclusions we've drawn there apply to this case as well.

The only difference is in the number of parameters. The maximum possible number of parameters is much larger for this case (on the order of 2^{2N} rather than 2^N, a quadratic difference). In practice, however, many, if not most of these parameters (in either case) will only ever occur in the matrix multiplied by zero, and can thus be dropped from calculations. It is harder to estimate this practical number of parameters, but it seems likely that it will be larger in this case than in the case in section 5.4.4.

5.6 Conclusions

Accuracy assessment of rough data seems quite feasible. Some measures have been introduced to describe various aspects of the roughness of the data. One measure has been introduced that can be used to compare rough data with data with a crisp representation, and describe the kind of match between them. Overall accuracy can be measured both in rough vs. rough and for rough vs. crisp situations. A methodology has also been given to apply traditional measures to rough data.

The number of measures applicable in the rough vs. rough case is quite small so far. Future work in this area certainly seems appropriate.

Chapter 6

Experimental Verification of Rough Classification

This chapter describes the experiment performed in [AKO98]. It starts with more information on exactly what analysis was performed; this is presented as a visual program. Then follow some details on some steps in the analysis, the numerical results, and finally the conclusions.

6.1 The Analysis

The goal of our experiment was to compare two vegetation layers, covering approximately the same area of about 1800 by 2300 meters in Stockholm county, just north of Stockholm. These two layers, however, were classified according to two different classification systems, done by different people at different times.

In our experiment we considered the two layers as two different representations of the same area. We started out with two vegetation maps which provided two crisp vegetation classification layers called veg9 and veg35. Building on vegetation concepts introduced by [Påh72], veg9 was classified using moisture and nutrient status as classification basis, giving nine different vegetation classes for the experiment area. The other layer, veg35, used a Nordic classification system described by [Påh95], giving 35 different vegetation classes for the experiment area. (Class descriptions may be found in [AKO98].) To compare the two, we needed to reclassify one of them; for obvious reasons we picked veg35.

Reclassification of veg35 into veg9 would be a straightforward task if a one to one or many to one correspondence between the two classification systems existed. As this is not the case, we used the idea of rough classification to

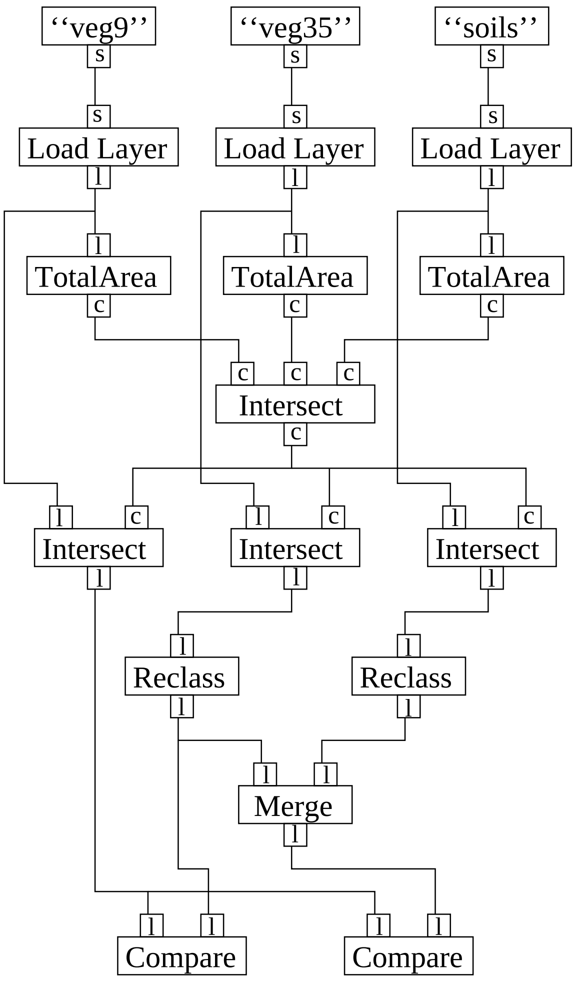

Figure 6.1. A visual program implementing the analysis.

represent the uncertainty in the reclassification operation and the following analyses of the results.

Given that the veg9 classification system uses nutrition and moisture properties as class definitions we also tried to resolve some of the uncertainty introduced by the reclassification. To achieve this we used a soil map of the same area to give additional information on the moisture component.

Rules to reclassify from the classification system given by [Påh95] to the one given by [Påh72] were constructed using guidelines given in [Påh95] and the knowledge of Ola Ahlqvist on the association between the different classification systems and soil classes. (The complete set of constructed classification rules may be found in [AKO98].)

Our analysis, then, could have been implemented using the visual program shown in figure 6.1. This figure introduces one new bit of syntax, and several new types of nodes. The new bit of syntax is that ports are labeled with a letter to indicate their type. The following port types occur here:

s: A string.

c: A rough class.

l: A rough layer.

Perhaps a brief explanation of the program, going through it from top to bottom, is in order. First, the three layers are loaded; since the original data is crisp, these rough layers contain no uncertainty. Then, the three layers are cut down to their common area; the original data covers approximately, but not quite the same area. The veg35 and soils layers are reclassified into (truly) rough classifications sharing the same classification scheme as veg9. These two reclassified layers are now merged into one according to the rules given in section 6.2. Finally, the veg9 layer is compared to both the reclassified veg35 layer and the merged layer produced in the previous step; this merge node has not been defined yet; basically, it produces the numbers given in section 6.3.

Another node that merits some more detail is the intersect node. This node occurs in two configurations in the program in figure 6.1. The node takes any number of inputs, and has one output. The first input can be either a layer or a class. All other inputs are classes. If the first input is a layer, the output is a layer; if the first input is a class, the output is also a class. For the all-classes case, it computes the intersection of all the classes. In case the first input is a layer, each class of the layer is intersected with all the other inputs, and the result of that forms a class in the output layer.

6.2 Merging Two Rough Classifications

The problem under consideration in this section is, given two rough classifications of the same area, with the same classification scheme, how do we combine them? For example, if, about a specific piece of land, the one classification claims that it is either forest or city, while the other classification claims that it is either lake or forest?

Let us call the input classifications A and B, and let us say that we are looking at these classifications in point x. We will call the set of classes of A of which x is a member K; for B, we will call that set L. The output classification we will call C, and the set of classes of which x is a member in C we will call M.

Note that in this representation we can not differentiate between if x is in the area of uncertainty of exactly one class or in the lower approximation of exactly one class. This is acceptable, because we are looking at complete classifications, and in that case x being in the area of uncertainty of exactly one class doesn't really make sense. This is true for M, the set of output classes, too. If M contains more than one class, x will be in the areas of uncertainty of all those classes. If M contains only one class, x will be in the lower approximation of that class.

There are two cases that have to be considered separately. Either the two classifications contradict each other (i.e. $K \cap L = \emptyset$) or they do not (i.e. $K \cap L \neq \emptyset$). It is quite likely that any two classifications will contradict each other in some points, if only in sliver areas along the not-quite-equal borders of classes.

Let's consider the non-contradictory case first. In this case, we will apply the rule that $M = K \cap L$. For a class c, if $c \notin K$ or $c \notin L$, that means that that classification says that it is certain that x is *not* in c. So only if $c \in K$ and $c \in L$ can we conclude that $c \in M$.

But what about the contradictory case? Here, we will adopt the reasoning that, even if they can not both be right, one of them probably is. Since we do not know which one, we will just say that $M = K \cup L$. So if classification A says that our point X is forest, while B says that it is really water, we will conclude that it must be either forest or water.

It should, perhaps, be noted that these are not the exact same rules as those used in [AKO98]. However, for the data in our experiment, there is no difference; the rules presented here are just an extension.

6.3 Results

We have computed various measures for our resulting data. However, at the time when we conducted this experiment, we had not yet derived the relative crispness measure from section 5.4.2, so we have not computed that measure.

The reclassified veg35 layer has an overlap measure of 1.058 and a crispness measure of 0.308. That means that about 30% of the area is certain, and the remaining 70% is uncertain. On that uncertain area, there is overlap (overlap measure is greater than zero). In fact, the overlap covers a little bit more than the whole classification; there must, in other words, be areas which are in at least three areas of uncertainty. When comparing the reclassified veg35 layer with the veg9 layer, we get an overall accuracy in the range of $[0.039, 0.461]$, not very good.

The merged classification has an overlap measure of 1.981 and a crispness measure of 0.168. That means that the certain area has almost been halved as compared to veg35, from about 30% to about 17%. About 83% of the area is uncertain, and on those 83% there is an overlap that covers almost twice the whole classification. Since $1.981 > 0.83 * 2$, there must be areas where there is a triple overlap; in other words, there must be areas which are in the area of uncertainty of at least four classes. The overall accuracy when comparing this classification to veg9 is in the range of $[0.031, 0.697]$, still not very good. It does mean, however, that if we get more information about the area under consideration, we could improve the overall accuracy at least up to 0.697.

6.4 Conclusion

Reclassifying from the input classification scheme to the output classification scheme would have been very difficult, if not impossible, without the use of rough classification. Hence, this experiment would not have been possible without this theory. Even if it had been possible to perform it using an approach with only crisp classification, that would not have given any information about what kind of uncertainty is introduced by the reclassification rules that were used; this approach does give such information. The low quality of the match does not imply that the rough classification approach is bad; perhaps some change in the landscape has occurred between the two classification events.

Chapter 7

Conclusions

A design for a visual programming language has been presented. This visual programming language is more structured and less confusing (both because feedback in the graph can only occur in a few well-defined ways) than existing similar languages. It should be easier to focus only on some details of the program being constructed, thus aiding readability. These features make the language especially suited for inexperienced users, which is a common class of users for geographical information systems.

Despite being simple to use, the language is still fully-featured. There is support for conditional execution, iteration and procedural abstraction, all using but one additional item of syntax. New types and operations can also be added to the language, both through external packages, programmed in a textual language, and directly in the visual language itself.

Since the intention is for this language to be used for the analysis of spatial uncertain data, operations supporting this kind of data have also been designed. They use a representation based on rough set theory, which enables the user to perform accuracy assessment of his data.

An experiment has been performed using this representation for uncertain data, comparing two incompatible vegetation classifications over the same area, something that would have been very hard without the use of this representation of spatial uncertainty. The analysis performed for this experiment has been phrased in the visual language developed, demonstrating the usability of the visual language. The experiment was successful, in that the two vegetation classifications have been compared, proving the usefulness of the representation.

Acknowledgments

This document would not have been produced without the help and support of many people; here I will try to thank all of them in no particular order. I am sure I will forget some of them (I just do not seem to remember who, just now), my apologies to those people.

My primary supervisor, Stefan Arnborg, has always been ready with advice. With Erland Jungert, my assistant supervisor, I have had many fruitful discussions about, among other things, this thesis and much of the material leading up to it. Per Svensson and Hans Hauska have also been a great help in this.

Of course, I happily thank Ola Ahlqvist and Karim Oukbir for letting me coauthor a paper with them on rough classification; without them, part two of this thesis wouldn't have been written. I have quite enjoyed my stay here at the theoretical computer science group; everyone here seems open to any kind of question or discussion. Special thanks for good times past (and hopes for more to come in the future) should go to Karim Oukbir and Lars Arvestad. Thanks to Lars Engebretsen for TEXnical help. Others with whom I've had interesting discussions during this time (not always about research, but always entertaining) include Mats Näslund, Rand Waltzman, Jesper Fredriksson, Gunnar Andersson and Öjvind Johansson.

For excellence in long-distance support, I thank my parents. Far away though they may be, yet they seem as close by as ever.

Last, but certainly not least, I thank Anna-Maria for everything.

Bibliography

[AD97] M. Auguston and A. Delgado. Iterative constructs in the visual data flow language. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pages 154–161. IEEE, September 1997.

[AKO98] O. Ahlqvist, J. H. D. Keukelaar, and K. Oukbir. Using rough classification to represent uncertainty in spatial data. In P. Firns, editor, *Proceedings of the SIRC Colloquium 1998*, pages 1–9. Spatial Information Research Centre, University of Otago, New Zealand, 1998.

[All83] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[AT95] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. In *Proceedings of Visualization'95*, pages 263–270. IEEE, 1995.

[BB94] M. M. Burnett and M. J. Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, 5(3):287–300, September 1994. More recent copy of classification system on the web: http://www.cs.orst.edu/~burnett/vpl.html.

[BBB$^+$95] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee. Scaling up visual programming languages. *IEEE Computer*, 28(3):45–54, March 1995.

[BC97] L. Braine and C. Clack. Object-flow. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pages 422–423. IEEE, September 1997.

[BM94] M. Bernini and M. Mosconi. VIPERS: A data flow visual programming environment based on the interpretive Tcl language. Technical report, Dipartimento di Informatica e Sistemistica, Università di Pavia, 1994. Available through ftp from ftp://iride.unipv.it/pub/vipers/papers/.

[BM98] P. A. Burrough and R. A. McDonnell. *Principles of Geographical Information Systems*, chapter 11. Oxford University Press, 1998.

[CDZ95] W. V. Citrin, M. Doherty, and B. Zorn. The design of a completely visual object-oriented programming language. In M. M. Burnett, A. Goldberg, and T. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments.* Prentice-Hall, New York, 1995. Available through ftp from ftp://ftp.cs.colorado.edu/pub/techreports/citrin/VOOP-VIPR.ps.Z.

[Cha90] S. K. Chang, editor. *Principles of Visual Programming Systems.* Prentice Hall, Englewood Cliffs, New Jersey, first edition, 1990.

[Chr68] C. Christensen. An example of the manipulation of directed graphs in the ambit/g programming language. In M. Klerer and J. Reinfelds, editors, *Interactive Systems for Experimental Applied Mathematics*, 1968.

[CHZ96] W. V. Citrin, R. Hall, and B. Zorn. Addressing the scalability problem in visual programming. Technical Report CU-CS-768-95, Department of Electrical and Computer Engineering, University of Colorado, Boulder, Colorado, USA, 1996.

[CJ96] S. K. Chang and E. Jungert. *Symbolic Projection for Image Information Retrieval and Spatial Reasoning.* Signal Processing and its Applications. Academic Press, 1996.

[CM94] D. Calcinelli and M. Mainguenaud. Cigales, a visual query language for a geographical information system: The user interface. *Journal of Visual Languages and Computing*, 5(2):113–132, June 1994.

[Con91] R. G. Congalton. A review of assessing the accuracy of classifications of remotely sensed data. *Remote Sensing of Environment*, 37:35–46, 1991.

[CS98] P. T. Cox and T. J. Smedley. Visual programming for robot control. In *Proceedings of the 1998 IEEE Symposium on Visual Languages (VL'98)*. IEEE, September 1998.

[Day88] R. S. Day. Alternative representations. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, volume 22, pages 261–305. Academic Press, New York, 1988.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium Proceedings, Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, Paris, April 1974.

[Dij68] E. W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[DK82] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, 15(2):26–41, February 1982.

[Dün97] I. Düntsch. A logic for rough sets. *Theoretical Computer Science*, pages 427–436, 1997.

[Dye90] D. S. Dyer. A dataflow toolkit for visualization. *IEEE Computer Graphics and Applications*, 10(4):60–69, July 1990.

[GG84] D. J. Gilmore and T. R. G. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21:31–48, 1984.

[GGS92] M. F. Goodchild, S. Guoqing, and Y. Shiren. Development and test of an error model for categorical data. *International Journal of Geographic Information Systems*, 6(2):87–104, 1992.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility–A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[Goo93] M. F. Goodchild. Data models and data quality: Problems and prospects. In M. F. Goodchild, B. O. Banks, and L. T. Steynaert, editors, *Visualization in Geographical Information Systems*, pages 141–149. John Wiley, 1993.

[GR90] E. J. Golin and S. P. Reiss. The specification of visual language syntax. *Journal of Visual Languages and Computing*, 2(1):141–157, 1990.

[Gre95] T. R. G. Green. Noddy's guide to visual programming. *Interfaces*, Autumn 1995.

[GS95] H. Glaser and T. J. Smedley. Psh – the next generation of command line interfaces. In *Proceedings of the 1995 IEEE Symposium on Visual Languages (VL'95)*, pages 29–36, September 1995. Available on the World Wide Web from http://kogs-www. informatik.uni-hamburg.de/~haarslev/vl95www/talks/T4.html.

[Hil92] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992.

[HTI90] M. Hirakawa, M. Tanaka, and T. Ichikawa. An iconic programming system, HI–VISUAL. *IEEE Transactions on Software Engineering*, 16(10):1178–1184, October 1990.

[Kim95] T. D. Kimura. Object-oriented dataflow. In *Proceedings of the 1995 IEEE Symposium on Visual Languages (VL'95)*, September 1995. Available on the World Wide Web from http://kogs-www. informatik.uni-hamburg.de/~haarslev/vl95www/talks/T21.html.

[KM66] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal for Applied Mathematics*, 14(6):1390–1410, November 1966.

[KS94] D. Koelma and A. W. M. Smeulders. A visual programming interface for an image processing environment. *Pattern Recognition Letters*, pages 1099–1109, November 1994.

[LV92] D. P. Lanter and H. Vereign. A research paradigm for propagating error in layer-based gis. *Photogrammetric Engineering and Remote Sensing*, 58(6):825–833, 1992.

[McI98] D. McIntyre. The comp.lang.visual FAQ. Posted regularly on the newsgroup comp.lang.visual, also available by ftp from rtfm.mit.edu, March 1998.

[Mol96] M. Molenaar. A syntactic approach for handling the semantics of fuzzy spatial objects. In P. A. Burrough and A. U. Frank, editors, *Geographic Objects with Indeterminate Boundaries*, pages 207–224. Taylor and Francis, 1996.

[Mye90] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[Påh72] L. Påhlsson. *Översiktlig Vegetationsinventering.* Statens Naturvårdsverk, Stockholm, Sweden, 1972.

[Påh95] L. Påhlsson. *Vegetationstyper i Norden.* Tema Nord 1994:665. Nordic Council of Ministers, Copenhagen, Denmark, 1995.

[Paw91] Z. Pawlak. *Rough Sets.* Kluwer Academic Publishers, 1991.

[PB93] R. K. Pandey and M. M. Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *Proceedings of the IEEE Symposium on Visual Languages (VL'93)*, pages 344–351, 1993.

[Pet95] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[PJ95] S. G. Parker and C. R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proceedings of Supercomputing'95*, 1995. Available on the World Wide Web from http://www.cs.utah.edu/~sci/publications/sc95_pj/PARKER_JOHNSON.html.

[PS74] J. M. Polich and S. H. Schwartz. The effect of problem size on representation in deductive problem solving. *Memory and Cognition*, 2:683–686, 1974.

[RCS97] A. Rau-Chaplin and T. J. Smedley. A graphical language for generating architectural forms. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pages 264–271. IEEE, September 1997.

[RI97] A. Repenning and A. Ioannidou. Behavior processors: Layers between end-users and java virtual machines. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pages 406–413. IEEE, September 1997.

[RW91] J. Rasure and C. S. Williams. An integrated data flow language and software development environment. *Journal of Visual Languages and Computing*, 2:217–246, 1991.

[SC97] T. J. Smedley and P. T. Cox. Visual languages for the design and development of structured objects. *Journal of Visual Languages and Computing*, 8(1):57–84, February 1997.

[Sch90] Andy Schürr. PROGRESS: A VHL-language based on graph grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science, 4th International Workshop*, volume 532 of *Lecture Notes in Computer Science*, pages 641–659. Springer Verlag, 1990.

[Sch95] M. Schneider. *Spatial Data Types for Database Systems.* PhD thesis, Fernuniversität Hagen, Germany, 1995.

[Sch97] A. Schürr. BDL – a nondeterministic data flow programming language with backtracking. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL'97)*, pages 398–405. IEEE, September 1997.

[Shn92] B. Shneiderman. *Designing the User Interface–Strategies for Effective Human-Computer Interaction.* Addison Wesley, 1992.

[UFK$^+$89] C. Upson, T. Jr. Foulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A computational environment for scientific visualization. *IEEE Computer Graphics and Applications*, 9:30–42, 1989.

[WH96] F. Wang and G. B. Hall. Fuzzy representation of geographical boundaries in gis. *International Journal of Geographic Information Systems*, 10(5):573–590, 1996.

[Whi97] K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, February 1997.

[Zad93] L. A. Zadeh. Fuzzy sets. In D. Dubois, H. Prade, and R. R. Yager, editors, *Readings in Fuzzy Sets for Intelligent Systems*, pages 27–64. Morgan Kaufmann Publishers, 1993.