

Avoiding Knotty Structures in Design: Schematic Functional Programming.

Thomas R. Addis¹ and Janice J. Townsend Addis²

University of Portsmouth, School of Computer Science and Mathematics,
Mercantile House, Hampshire Terrace, Portsmouth PO1 2EG,
Tom.Addis@port.ac.uk.

Abstract. Designers in general have used diagrams and sketches to help in the process of creation. This is particularly so for system designers whose output is a set of programs. It would seem reasonable that the conversion of diagrams directly into a program would be desirable and yet the work of Green and Petre [12,13,14,19] and Citrin [8] has placed doubt on the viability of graphical programming notations. Some of this work is reviewed in this paper. The use of secondary notation and the match-mismatch hypothesis is reconsidered in the light of functional programming. It is proposed that much of the criticism of graphical notation is due to the imperative (or process orientated) nature of programming. Many of the limitations observed in using graphical notation are lifted when functional programming is used. Eight engineering dimensions and four engineering relationships (coherences) are proposed to describe programming environments (including notation). The key to the success of a functional language as a general representation as well as its coherence with a graphical notation comes from its unique extensibility. Support for these arguments is drawn from examples of a schematic programming language used for industrial scale projects.

*Nothing ever becomes real till it is experienced - even a
proverb is no proverb to you till your life has illustrated it.
Keats, letter, 1819*

1 Introduction

The value of using graphical notation for programming is normally addressed as a study in psychology. The authors re-evaluate the utility of such a notation from the engineer's viewpoint. We will thus make our main focus the product of design; the computer program.

¹ Visiting Research Fellow at the Science Studies Centre, Department of Psychology, University of Bath.

² Visiting Researcher at the School of Computer Science and Mathematics, University of Portsmouth.

1.1 The Collapse of Visual Languages

In June 1996 the MIT Laboratory for Computer Science hosted a workshop on Strategic Directions in Computing Research. The 300 participants were divided into 22 working groups where each group explored different topics. Their reports were eventually published in the ACM Computer Surveys [23]. One of these reports was concerned with Visual Languages [8]. The report noted that despite the great intuitive appeal of such languages they have not been adopted except as curiosities. Citrin suggested that there are three distinct areas which make visual languages unacceptable for programming. These are:

- *Appropriateness of mapping*, deciding which aspects of problems map naturally into visual representations and which are best left as text.
- *Scalability*, the ability to construct and comprehend very large visual programs
- *Transparency of the interface*, allowing changes and modifications of visual programs to be as easy as editing text.

The suggestion here is that if these were ‘solved’ then visual languages would become more acceptable.

Green [14] some six years earlier had already asked the question:

“Is there evidence that diagrammatic notations genuinely offer something that symbolic notations cannot match?”

and noted that:

“It is disturbing to observe that many workers have not even asked *this* question, and instead assert uncritically that ‘Pictures are more powerful than words .. Pictures aid understanding and remembering.....Pictures do not have language barriers. When properly designed, they are understood by people regardless of what language they speak.’ If the more dewy-eyed propositions of program visualisation enthusiasts are accepted, then we can expect diagrams to be better for all purposes.”

Soon after this observation Green with Petre [19] published the results of a series of careful studies exploring different graphical and text notations for ease of programmer interpretation. They concluded:

- Overall, graphics was significantly slower than text to interpret.
- Moreover, graphics was slower than text in all conditions and the effect was uniform.
- The intrinsic difficulty of the graphics modes was the strongest effect observed.

These damning results were illustrated by selected observed behaviour and they concluded that:

“Far from guaranteeing clarity and superior performance, graphical representations may be more difficult to access.”

Their most vivid description was concerned with ‘Knotty Structures and Working Memory Load’:

“Program structures can contain ‘knots’ which force working memory load on the reader.The sight of subjects crawling over the screen with mouse or fingers, talking aloud to keep their working memory updated, was remarkable. One of the distinctions between expert

and novice behaviour was that the experts made better use of their fingers, occasionally using all 10 to mark points along a circuit.”

These are indeed remarkable observations which were supported by further work published in 1996 [13]. Particularly since engineering of many flavours would be impractical without diagrams. It is evident that the experiments are only concerned with interpretation. On the other hand, an engineer is mainly concerned with design and it would seem reasonable that design should depend upon interpretation. So the questions arise ‘Does design depend upon interpretation?’ and ‘What might be the difference between engineering a computer program and other engineering practices that seems to cause diagrams to become a positive disadvantage?’

1.2 Engineering Drawings

There is no problem in understanding the utility of an engineering drawing that represents the design of a bridge, engine or aeroplane. A scaled schematic provides a means of trying out designs on paper before the expense of manufacture. It is a skilled task involving considerable precision which requires training from an expert in the field. The diagrams are drawn from views employing many established conventions. *There is no pretence that the diagrams are easy to produce or necessarily simply understood.* The drawings are surrogates for the real thing [11] providing images that allow some direct measurements and that act as a guide for the creation of equations.

Electronic circuit diagrams are also surrogates. Switches, uniselectors, relays, electric motors, valves, transistors, resistors and capacitors are all represented by icons that reflect something of the physics of how they work. There is sufficient in the abstract representation of these icons to remind the engineer about the nature of the devices to the extent that their unusual properties can be deployed. For example, screen grids of valves can be used to mix signals, the coils of relays can double as inductances, and transistors can be used (with care) symmetrically. Logic circuits are also surrogates even though they are not so well endowed with icons that represent their workings. The logic gate is too complex to represent its structure compactly.

In this direct case of engineering drawings it would appear that the formal and the informal semantics are folded into each other; what the notation represents in the working system and what the engineer intends it to represent in the world are one and the same thing. However, this simplicity diverges slightly when VLSI was introduced to replace the normal components of electronic circuits. Transistors and other components are now intersections of laminar areas of different materials. To cope with this, a new kind of diagram was introduced to bridge the gap between the implemented device and the electronic circuit, the stick diagram [17]. Sometimes the engineers combine both kinds of notation into one diagram. The equivalence of the formal and the informal semantics no longer holds directly because each kind of diagram now represents a different abstraction of the artefact.

However, in all cases of engineering drawings they are also used to help create calculations and thus make predictions of behaviour or performance. This involves looking up information in tables and deploying laws of physics, such as Newton’s three laws for mechanical devices or Ohms and Kirchoff’s Laws for electrical circuits.

There is no need to carry out experiments to demonstrate the value of these drawings for the design of artefacts. They may not be perfect but they would not be used if there was any better method.

1.3 Engineering Drawings used for Programming

The notion of a surrogate has its place in software engineering. A typical set of diagrams are those used by the Jackson Method [22]. In this case the icons represent physical files and processes (i.e. data and program). The key to the effective use of these diagrams for design is familiarity with and the skilled use of the normalisation process as applied to data structures [9, 16]. This normalisation process provides a mechanism which will identify the elements to which the diagrams should refer. The fundamental principle upon which all this rests is 'functional dependency'. Functional dependency is an abstract property of a function's mapping potential. This property distinguishes between the unique identifying attributes of an object or concept and its other properties. Thus the attribute 'part number' uniquely identifies a 'part type' but not the other way round; it says something about the information properties of data.

Something has changed here in that the elements referenced by the icons are fluid. They are no longer easily identified objects as is a logic gate or a switch. The focus of design has shifted towards the relationship between the design process and the designed artefact. The involvement of the designer in deciding what the nature of the elements of the world are to be, although rarely acknowledged, is paramount to the ultimate design.

LabView, MatLab and Prograph are all graphical languages that draw upon the circuit diagram as their representational paradigm. The obvious appeal is to the skills that engineers already have in understanding and designing with such languages. The advantage of such languages over normal engineering drawings is that a circuit can be tried out before being built. This gives security of design; it ensures the artefact works as expected.

The oddity about these languages is that people would choose to use them as a general programming tool given that the elements have been fixed for reasons that have little to do with the potential of a von Neuman machine or the possibility of abstracting elements from a problem domain. It is thus not too surprising that they are confined only to certain areas of work.

Current design methods for software are inadequate and do not provide the same security of design as found in electronics. We will now ask the question 'Given complete freedom of choice on a representation, unbounded by physical constraints, what would it be?'

2 Problems of Software Design

2.1 A Typical Example

A typical example of the problems encountered in design and design flexibility is illustrated by a company that manufactures specialised automata for industry. Because

their customers wish to run flexible manufacturing systems the company would like to allow users to interface easily with their machine system to facilitate modification to the system during use. Based on current techniques each bespoke variant is costly to produce. Every item has to be individually programmed for each customer. The programs are non-trivial and in many cases cannot be tested until at the customer's site.

This example is typical of the difficulties with which industry is faced given present approaches to software systems: development, design, maintenance and modification costs. Complex systems are open to continuous modification throughout their lifetime; it is rare to have a final design.

2.2 The Engineering Dimensions of Design

In order to be able to evaluate possible program representations and the design environment in which it is used we propose eight dimensions and four relationships. These describe the potential for change that can be designed into an artefact. They give a language with which to relate important properties of design with a notation or a programming style. They suggest the effects that this potential has on the subsequent quality of the artefact where this quality can be detected through 'indicators' or characteristics of the program or the environment.

The first four properties are very similar in that they all have the effect of fragmenting the functions into small units.

- **Referential transparency** The advantage of referential transparency is that a function can be tested as an isolated unit.
Indicators: There will be no side-effect programming. e.g. no assignments, no global variables and no sequence (order dependent) groups of functions.
- **Functional Decomposition** of a problem will reduce the design to a set of functions that may be either intensional (composed of other functions) or extensional (lookup table). Such decomposition should be independent of any particular computer architecture since computer architectures impose a restriction on what can be represented and on how designs are perceived.
Indicators: There will be a network of user defined functions that will tend to form a hierarchy. Relatively long path lengths would be expected between top level functions (i.e. functions with no parents) and basic language functions. This is because the problem domain concepts will need to be represented as functional units. These units are likely to be very different from that of the computer.
- **Interoperability** indicates that designs can be responsive to change [18] and in particular it means creating systems of independent working units. Such independent working units can then be used with different systems.
Indicators: Combines referential transparency with decomposition.

- **System Flexibility** refers to the artefact's capacity to be changed or to cope with extensions of its use. It reflects the systems ability to be modified. Flexibility depends upon the generality built in to each function. Input parameters should be tested and strategies devised to cope with a range of possible inputs. Thus functions are built to always explore the assumptions about their use. Functions would use functions that would analyse the data (e.g. counts on lists, data type testing, conditionals).

Indicators: Input parameter pattern sensitive (pattern match) extensions could be expected which trigger off different strategies. Good referential transparency. Older programs should show traces of change through large numbers of redundant top level functions.

The next two properties are consequences or give extra benefits that depend upon the first four.

- **Reversibility** is the possibility of re-construction or reverse engineering a design. This ensures that nothing of the basic mechanism is lost in going from design to implementation or from one design representation to another. What losses there are should be easily reinstated or at least detected. Reversibility can also refer to the *inverse* of a function; going from result to the range of parameter values.

Indicators: In the design case this depends upon referential transparency and in the function case this depends upon the existence of an 'inverse' function.

- **Design Transparency** is the ease of comprehending the overall structure of a system.

Indicators: Local understanding achieved through simple functions (i.e. functions with few children). The potential for clear overviews from the design environment.

The last two properties are concerned with reducing the effort needed by the designer to create an artefact.

- **Environment Reactivity** is the immediate responsiveness of the design environment to the designers decisions and actions.

Indicators: An interpreter is used.

- **Language Extensibility** is how a particular language or representation may be augmented through its own constructions. A measure of this is based upon knowing the 'ground state' of the language³. The ground state identifies the minimum set of primitives from which all other functions can be derived. The ground state of a functional language has only *three* primitive elements, namely constructor definitions, parameter pattern match and recursion [5]. However, what these three primitives cannot do is provide the side-effects that link the abstract functions with the world. This requires engineered code .

³ A functional language is derived from lambda calculus. It is this calculus that ensures its coherence as a tool for abstraction and applicability.

Indicators: The smallest number of elements that provides total control over the machine. Thus the notation should be simple with a minimal grammar involving few symbol types. The machine may be virtual as in the case of the functional language.

2.3 The Engineering Coherences of Design

Coherences describe the similarity relationship between a programming language and elements of design:

- **Operation:** This relates language with the engine it controls.
Indicators: The language reflects the actual operations being done by the machine.
- **Semantic:** There are two kinds of semantics. The *formal semantics* and the *informal semantics*.
Indicators: The semantic coherence can be assessed by the introduction of one or more abstract layers. This will be indicated in a similar way to decomposition. This assumes that the formal semantics is different to the informal semantics and that this is overcome by the introduction of mediating functions. New concepts would be introduced through constructors which are used to create new types of variables and objects.
- **Representation:** This coherence occurs with computers which are primarily sequential and sequential programming languages are appropriate. However, time ordered tasks, in general, are both sequential and parallel and as such a two dimensional representation would be more consistent. Representational coherence is related to the match-mismatch hypothesis [12].
Indicators: If there is good coherence then the degrees of freedom of the representation are the same as its functionality. Thus the notation will be used to represent only the functionality. For example, in the incoherent case of text notation there are variables used to make links; links that do not in themselves take part in the processing except to pass information. These variables may cause unwanted side-effects.
- **Design:** *Design coherence* ensures well formedness at the formal level through the structure of the language and support of the design environment.
Indicators: An indication of design incoherence is when there are type mismatches or redundant code. There will be a reduction of errors during implementation. Hence, simple functions with few children would be expected.

2.4 Functional vs. Imperative Programming

Essentially, we have two major choices in programming styles: functional or imperative. Logic programming has the functional style and OO has functional properties embedded in an imperative style.

A pure functional approach has advantages over an imperative approach in having *referential transparency*. This makes for clear design structures and thus it does not require very sophisticated error tracing facilities. However, it has two disadvantages:

1. it requires that the links to real world operations are side effects. That is the input from the world to a function, and the output to the world from a function are not definable within the formal structure of a pure functional language. These are events that are not expressible within the formal semantics.
2. the problem domain states must be passed/copied between functions (also see *interoperability*). That is if many functions process the data then each function can have access only through one of its input parameters.

Imperative languages have good *operation coherence* in that they reflect the computer processes in their syntax and semantics. Thus machine code maps directly into the computer device and describe exactly what is done within the machinery. Operation coherence begins to be reduced as higher level languages are introduced. Unfortunately, because the state of such a program depends upon its history and because communication is done through side-effects, problems arise when errors occur or changes have to be made. Consequently, the cause of an error may be many thousands of operations distant from its observed symptoms. In particular, changes can have effects that are not traceable until a problem arises.

3 Creating a Schematic Language

We asked the question ‘Given complete freedom of choice on a representation, unbounded by physical constraints, what would it be?’ There are good reasons for choosing a functional language but if that is the case then why are we not all using such a language style now?

3.1 The Problems of a Functional Language

Notwithstanding the lack of take-up of functional languages to date, we will argue that in principle, they offer many of the desired attributes we need for design and are a good starting place to resolve many of the difficulties encountered. However, we must consider why functional languages have failed, thus far, to find acceptance.

The three major problems with a functional language are the three incoherences of operation, semantic and representation.

1. Functional languages do not match the computer architectures on which they run. This leads to unexpected and massive processing, sequential changes of operations and heavy main memory demands. Links with the real world are indirect and have the reduced status of being just ‘side-effects’.

2. The formal assumption behind a pure functional language is that the informal world for which a program is to be implemented is completely known and definable. Although this gives a language in which all formal concepts may be represented (a powerful property) it creates a barrier to many practical applications. So even processes that depend upon time, a basic assumption with imperative languages, requires a laborious and explicit description of time itself. The representation becomes unwieldy as soon as the complexities of the world are included.
3. The mathematical parentage of a pure functional language makes it obscure to many people. Processes are described as relationships and although in many cases the representation is the same there are some important differences (e.g. the use of recursion instead of iteration).

3.2 Solving the Problems

We will consider the three incoherences of operation, semantic and representation with the possible options and then explore the effect of these choices on design and design support.

Operation incoherence. This will be exaggerated through the use of an interpreter but an interpreter needs to be used in order to retain the *reactivity* and the *flexibility* of the design environment. Experience has shown that there is an ideal and engineered balance between imperative and functional specifications that optimises the advantages of both styles of expression (e.g. LISP). If we can take a small step away from a pure functional language and invoke ‘side effect’ programming as a clear and distinct activity then we can support *operation coherence* when and where required. Such a decision should remain under the designers control. However, to retain the *semantic coherence* such a step would still have to operate through an interpreter.

In many cases and with advances in technology the performance overhead of an interpreter is not as significant as it was (say) ten years ago. However, in cases where the real time performance becomes critical and because of *referential transparency*, *interoperability* and *decomposition* then any function on the critical path could be translated into its imperative form without much loss of *design transparency*. Such a modification would neither alter the original design nor interfere with the *design coherence*. It would also allow the natural integration of already existing software.

Informal-semantic incoherence. This is mainly caused by the need to refer to a world that changes. If an extensional function is allowed to have its extension changed dynamically then this function can act as a global variable. The parameters of such a function would then behave like an array index with the additional advantage that this index can be any conceivable type. In this way real world data can be represented without the need to pass it around as parameters. If this is combined with both ‘side effect’ programming and the functional notation then this combination will increase the *extensibility* of the representation into another dimension and still retain *semantic coherence* in both.

Representation incoherence. This can be alleviated by representing the two dimensions of a process as a set of diagrams. To aid the transfer of design skills of engineers and imperative programmers graphical equivalent representations have been

used [21]. These graphs have three characteristics: *first* it makes clear the relationship between the component parts of a user constructed function, *second* the directed graphs are at their most transparent when they have few elements connected together and *finally* functions are best constructed from only a few component functions. Given this match and the observation that many designers choose to use a graphical representation to explore design options then a functional language that can be directly interpreted through a graphical representation seems to provide the optimum design environment [15]. In this way we have good *representational coherence*, *design coherence* and *transparency*.

We will thus choose an approach that will combine graphical (schematic) and functional representation. Taking our lead from those in VLSI design we will give the designer the option to select and mix any combination of the two representation schemes. We now ask ‘what kind of diagrams should we use?’

3.3 Pedagogic Use of Schema

In response to the needs of his students Reade [21] developed a schematic representation of a functional language (Fig.1). For example, the function ‘stringcopy’ which appends ‘n’ copies of the string ‘s’ together into one long string is described as a ‘white⁴’ box. The function ‘stringcopy’ uses other functions shown as ‘black’ boxes that are feeding their results into each other. The order of the parameters are from top to bottom. The ‘=’ function appends two strings together and ‘”’ is an empty string.

This kind of schematic did not develop any further. It was able to describe recursion and there were attempts at illustrating higher order functions. There was no intention of the schematic being anything other than pedagogic. The underlying functional language was Standard ML.

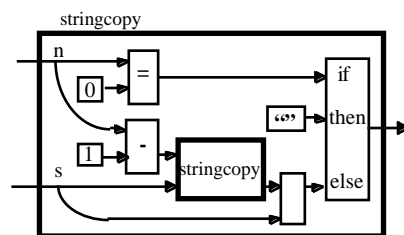


Fig.1. stringcopy : int × string → string
`fun stringcopy (n,s) = if n=0 then "" else stringcopy (n-1,s) s`

Design methods, based upon normalised relations, use semantic functional dependency graphs (sfd) or their equivalent to choose a coherent transformation into file structures and processes [6]. These graphs are limited to a few but important relationships (e.g. many-to-one). Attempts at extending the semantic functional dependency graphs by Addis and Bull into a conceptual language CL came to halt in 1987 [1] for two fundamental reasons:

⁴ ‘White’ here means transparent as opposed to a ‘black’ box where the contents cannot be seen.

1. the number of possible relationships were infinite and since the language needed to have every relationship predefined it was clearly not possible.
2. even though a number of useful relationships were described few people seemed to be able to use the language; it was just too difficult.

3.4 A Schema for Functions

It was proposed in 1988 during a project involving a content addressable database [2] that a functional database language (fdl) [20] could be used to overcome the problems of extending the relational representation. Knowing the difficulty some people have in understanding a functional language the schematic diagrams of Reade were considered as a potential route to follow. The result was the functional database language Faith and the formal schematic language equivalent Clarity [3, 4, 5]. Faith is generated from the Clarity diagrams and vice versa. The process is ‘nearly’ reversible; ‘nearly’ because layout information is lost in either direction. Only the formal semantics are reversible.

Fig.2. shows the Clarity version of the function ‘stringcopy’. The input parameters of the function are shown in the top field as bold lozenges. The type or a constant is shown inside these lozenges. The # indicates an integer. The ‘result’ of the function is shown in the bold lozenge in the bottom field. The central field shows the body of the function. Functions in the body are shown as oblongs where the order of the input parameters is counted *clockwise from the single output*. The dotted lozenge shows the type. In this case the ‘ ’ function is replaced by ‘str_concat’ which takes a list of strings and (- 1) is replaced by ‘sub1’.

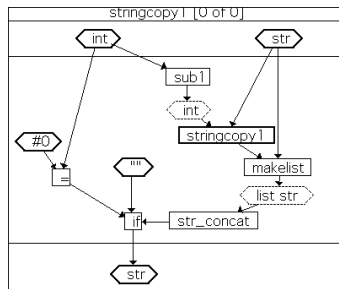


Fig.2. An exact equivalent of Reade’s function ‘stringcopy’

Fig.3. shows a version of ‘stringcopy’ that uses ‘pattern-match’ instead of the function ‘if’. The functional languages gives the provision for functions to be defined with a range of input patterns where each pattern may perform different operations. The best match pattern is always the one selected. This ability to respond to patterns, involve recursion and declare constructors allows new types of objects to exist. The functional language is extensible. We will be using this pattern-matching to avoid ‘knotty’ structures.

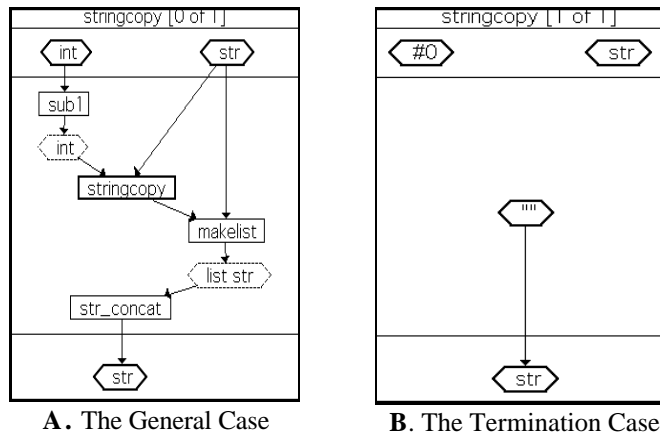


Fig.3. The function stringcopy (n,s) in the Clarity representation

Both the functions 'stringcopy1' and 'stringcopy' can be tested by typing them into the control window as a normal function thus:

```
QUERY> stringcopy #3 "xyz"
"xyzxyzxyz"
```

4 Comprehension of Conditionals

4.1 The Problem

Petre and Green [19] investigated a wide range of notations using a standard test example involving conditionals (Fig 4). The performance of subjects in answering questions on the program written in these different notations was used to assess the notations as mechanisms for transmitting and describing a process clearly to a person. The task for the subject when faced with a program was to determine what it does and what given results might be implied about a given input condition. A further task was to compare programs in the same and different notations. The questions are then to do with determining how the programs differ. Every attempt was made to layout each notation at its best. Experts in each notation were used to design the test programs. It was a good experiment and well designed. The results are well known.

```

if high:
  if wide:
    if deep: weep
    not deep:
      if tall: weep
      not tall: click
    end tall
  end deep
not wide:
  if long:
    if thick: gasp
    not thick: roar
  end thick
not long:
  if thick: sigh
  not thick: gasp
end thick
end long
end wide
not high:
  if tall: burp
  not tall: hiccup
end tall
end high

```

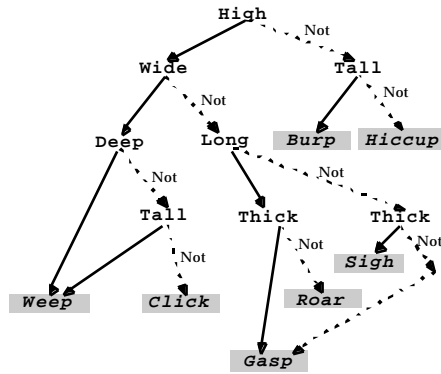


Fig. 4. One of the problems in **Fig. 5.** A decision tree/network that Nest-INE notation from Petre and Green 1993 describes the test problem.

4.2 A Schematic Functional Equivalent

As engineers and designers our first instinct is to abandon all the representations investigated. They are all difficult to use for this problem and the given graphical representations particularly so. Instead we would chose to produce a decision tree/network (fig 5) as our common interpretation of each notation⁵. This matches the problem nicely. It shows that there are two types of variables; features and actions. 'Not' may be considered a feature. In Clarity these features can be represented by constructors belonging to the types 'feature' and 'action'. 'Not' becomes a unique feature in that it takes a feature as a parameter. This is shown in fig 6 and fig 7.

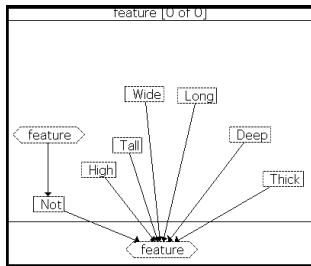


Fig. 6. The set of type 'feature'

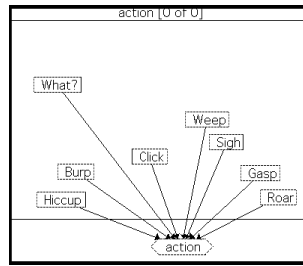


Fig. 7. The set of type 'action'

Going down all the paths of the decision tree (fig. 5.) it is possible to list the features that lead to a particular action. These lists of features can be used to index an extensional function 'respond' the output of which is an action⁶. This function is

⁵ Some designers may prefer a decision table. This really is a matter of choice and the designer's habits rather than any fundamental issue. Figure 9 can be considered to be such a table.

⁶ Each schema extension can be used as a pro-forma to produce the next extension.

equivalent to a look-up table. All non-specified sequences of features are caught by the general case (fig. 8a) which will respond ‘What?’. Such a nicety is optional.

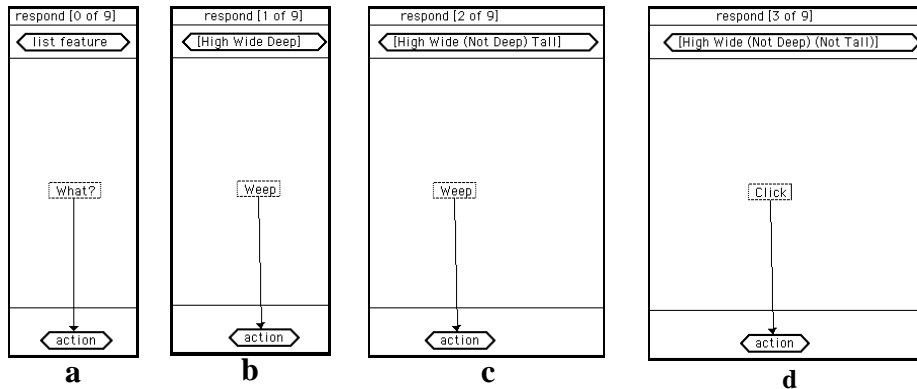


Fig. 8. A sample of the extensions of the function ‘respond’.

Within the Clarity environment it is possible to look at the Faith code generated from the schema (Fig 9), flick through the extensions of ‘respond’ sequentially or display all the extensions at the same time. It is also possible to ‘query’ the function in the control window.

```

fdec
  respond ::= list feature ->action ; ;
fdef
  respond ?0 ::= What? ;
  respond (: High (: Wide (: Deep nil))) ::= Weep ;
  respond (: High (: Wide (: (Not Deep) (: Tall nil)))) ::= Weep ;
  respond (: High (: Wide (: (Not Deep) (: (Not Tall) nil)))) ::= Click ;
  respond (: High (: (Not Wide) (: Long (: Thick nil)))) ::= Gasp ;
  respond (: High (: (Not Wide) (: Long (: (Not Thick) nil)))) ::= Roar ;
  respond (: High (: (Not Wide) (: (Not Long) (: Thick nil)))) ::= Sigh ;
  respond (: High (: (Not Wide) (: (Not Long) (: (Not Thick) nil)))) ::= Gasp ;
  respond (: (Not High) (: Tall nil)) ::= Burp ;
  respond (: (Not High) (: (Not Tall) nil)) ::= Hiccup ; ;
  
```

Fig. 9. The complete Faith code generated by Clarity

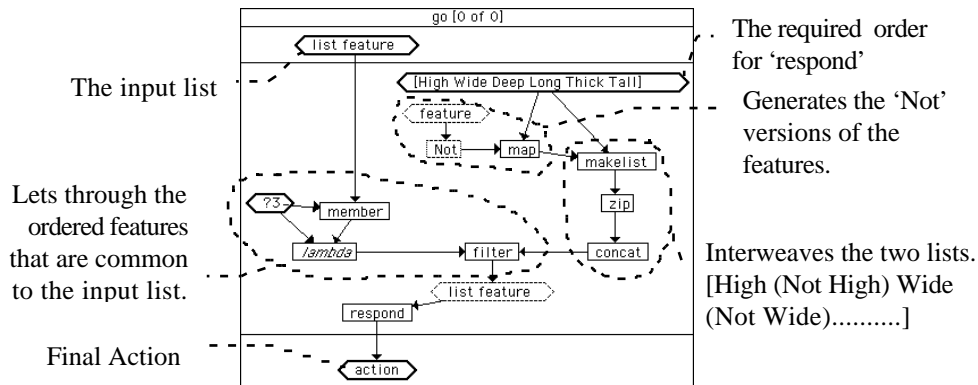


Fig. 10. A simple function ‘go’ to avoid feature sequence
It also shows the potential for decomposition

```
QUERY> respond [High Wide Deep]
Weep
QUERY> respond [High (Not Wide) Long (Not Thick)]
Roar
```

We can also ‘query’ the function in reverse:

```
QUERY> inverse respond Weep
[ [ High Wide Deep ]
  [ High Wide ( Not Deep) Tall ] ]
```

The current arrangement requires that the features are listed in a particular order, so a simple function ‘go’ was written to overcome this problem (see fig 10). The function ‘go’ can be used thus:

```
QUERY> go [ Tall (Not High)]
Burp
QUERY> go [Deep Thick]
What?
```

This program has been written without a single ‘if’. It relies upon two useful characteristics of a functional language; pattern match and constructors. We doubt if any fingers will be required to answer questions on the behaviour of this program other than to use the key board. The overall nature of the function ‘respond’ is that it *responds* to patterns, the details of which can be ignored or changed when used within a larger context such as ‘go’ (figure 10). However, we now need to ask ‘Is it a general characteristic of the functional language that it always requires less use of ifs and if so, by how much?’

5 Exploring ‘Real’ Programs

5.1 The Approach, Results and Interpretation

All the programs we have chosen to examine are ‘real’ in the sense that they were created to perform a job. In most cases they are programs paid for by business and written by professional programmers. ‘Novice’ refers to inexperienced (months rather than years) programmers in the language referenced. A further requirement is that they are all non-trivial programs. We have selected a range of authors from expert to novice.

The three programs X, Y, Z and Q have been written in Clarity. The lines of code refer to the generated Faith. Approximately, 17K of the lines of code of X are related to the Expert System rule set and have been discounted from the analysis. These rules are stored as the extensions of three functions (c.f. section 4.2). X was created in one year. Y is non-commercial and is an on-going and ever changing tool for exploring ‘agent’ group behaviour. Y is about seven years old but ‘cleaned’ by the removal of redundant functions from time to time. Z is a new industrial control system that has been created in Clarity and written by a novice in two months. Q is a program used to

modify bridge designs through Genetic Algorithms and written by a competent programmer over two years. The four C programs have been written by four different authors. They have not been dealt with individually. Initially, they will be taken as a single source and used as our standard (Table 1). The tables 2 to 3 are headed with the dimensions for which the containing data are indicators (see section 2.2 & 2.3).

Table 1. Summary of programs used for the analysis.

	X 'Clarity'	Y 'Clarity'	Z 'Clarity'	Q 'Clarity'	C 'C'
Job	Expert System Rules (No Rules)	Agents Model	Industrial Control	Engineering GA	Interpreter, Statistics, 2 translators
Total Size Kb	946 (372)	58	65	72	139
Programmer	Expert	Expert	Novice	Competent	4 Progs
Code (lines)	31961 (15354)	2033	1838	413	5444
Tot Functions	194	176	90	52	125
Code/Function	80	12	20	8	44

The natural unit for a functional program is the 'function'. The composition of a function consists of a selection of pre-defined library functions and a set of user defined functions. What we need to show is that these user functions are equivalent to the functions in a C program. We will assume that a designer has a completely free choice in the size and content of a function within the constraints of the problem. The designer will thus chose a 'unit' of design (the function) with which he/she is comfortable and which can be handled easily. However, the aberrations of a problem to be solved by a designer may demand an undesirable unit size. Nevertheless, we would expect that the units should tend towards a norm. The units, by their nature, will be defined in terms of other units and it is the number of *different*⁷ units per function that we will use as our measure (Table 2).

⁷ Rather than the total number.

Table 2. The sets of **children** functions used to define a function

<i>Functional Decomposition, Interoperability, Design Transparency, Semantics Coherence, Design Coherence</i>					
	X	Y	Z	Q	C
Children					4 Programs
Average All ()	6.3	6.0	8.5	4.9	$3.8 + 3.7^8 = 7.5$
Sigma _a All	4.2	3.6	7.6	2.7	4.0
Standard Err _a All	0.30	0.27	0.80	0.37	0.36
Expected# User (μ_c)	2.0	2.0	4.0	1.2	1.9
Sigma _c User	2.0	1.9	4.9	1.4	2.2
Standard Err _c User	0.14	0.14	0.51	0.19	0.20
Expected# Library (μ_l)	(4.3)	(4.0)	(4.5)	(3.6)	5.6
Sigma _l Library ⁹	(5.6)	(4.7)	(9.5)	(3.5)	(5.2)
Standard Err _c Library	(0.40)	(0.35)	(1.00)	(0.49)	(0.47)

The nature of the functional decomposition can be assessed by considering the number of parents of a function; this is the function's deployment. If there is more than one parent then we are detecting the existence of an intermediate conceptual layer. Such a layer suggests that there is a tension between the formal and informal semantics and that, in effect, a new set of concepts needs to be constructed to better match the problem domain. Many functions with zero parents indicates either that the program itself is a kit of tools or that functions have been created, replaced and abandoned. If flexibility is an issue then the number of the latter type of function should grow with age.

Table 3. The sets of **parent** functions per function (usage)

<i>Flexibility, Functional Decomposition, Semantic Coherence,</i>					
	X	Y	Z	Q	C
Parents					4 Programs
Zero ()	19%	24%	6%	34%	14%
One ()	50%	44%	61%	47%	42%
Expected# (μ_p)	1.5	1.8	2.2	1.1	1.9
Sigma _p	1.5	2.6	2.4	1.5	2.3
Standard Err _p	0.11	0.14	0.25	0.20	0.20

Functions with only one parent suggests that complex functions have been simplified. The simplification is by the removal of a set of functions within a function to form a new single unit. For example, this could have been done by extracting one or more of the operations circled in the function 'go' (see Fig 10). The results show that the expected number of library children remains reasonably constant in all cases in that they overlap within the standard error. This supports the hypothesis that the decomposition comes primarily from the user functions. Where there is a

⁸ These are the language library functions such as '==', 'while', and 'for'.

⁹ Use equation $\sigma_l = [2 \sigma_a(n-1)/n - \sigma_c^2]$

small number of expected user children per function then this implies that either the library functions are used more effectively or there is a better match of the notation with the problem. The high value of expected children for Z suggest that the problem domain is not well matched to a functional approach. This was confirmed by the programmer and his colleagues.

Table 4. The number of ifs per function

<i>Design Transparency, System Flexibility</i>		
	XYZQ	C
Progs (Authors)	4 (4)	18 (10)
Ifs/Fun μ_p	[0.17 0.17 0.26 0.38] = 0.25	3.38
Sigma σ_p (crctd)	0.086 (0.099)	2.04 (2.10)
Standard Err σ_p	0.05	0.49

The expected ‘ifs’ per function for C were determined by counting the number of ‘if’s used per program for 18 different programs (Table 4). The population is now based upon programs rather than functions. These include the 4 programs used in the function equivalence study (Tables 1 to 3), financial, utilities and textbook C. The Snedecor’s F test for the difference between the variances = 450 which gives a significance better than 0.1%. The difference between the two populations is greater than six standard errors; a very significant difference.

5.2 Discussion on Results

We have shown for experienced programmers that the functional unit tends to consist of about six different child units. The expected number of library programs per unit for all programmers, problems and notations is about four units. The observed variation in the number of user functions per unit could be explained by differences in experience and the problem domain. In the latter case, the relevance of library functions becomes less as more user functions are developed to overcome the semantic tension experienced between the problem and the resources of the language.

We have shown that the expected frequency of ‘if’s in a C program is an order of magnitude higher than in a functional language. This seems to be the case for both Expert and Novice programmer. Generally novice programmers will use more ‘ifs’ than experienced programmers. Even so, a novice Clarity programmer will still use about 8 times less ‘ifs’ than a C programmer.

6 Conclusion

We can now respond to the deliberations on page 2 given by Citrin, Green, Petre et al. Consider the issue of ‘appropriateness of mapping’. So long as the formal semantics of a range of notations are the same then the choice of notation is not relevant to the structure and size of a function. This we have demonstrated in section 5.1. The choice of notation is an independent matter; a matter of preference. Such choice can be made

by considering the potential to use the informal features of the notation to some advantage. In the case of Clarity, the schema provides a useful transformation on the layout of a functional language.

One of the difficulties with a functional programming language is that the functions are imbedded in each other, the interpreter works from the outside in and it evaluates from the inside out¹⁰. Imperative languages are more straightforward in that they evaluate in the order they are written unless explicitly commanded otherwise. This is much more easily understood. Clarity, however, provides the option to arrange the schema to place the inside of functions at the top and the outside of the functions at the bottom. In this way the display of the functions has all the benefits of the imperative ordering of functions and yet it also has all the formal equivalence and the advantages of a functional language.

The issue of 'scalability' has been resolved partly through the use of a functional language, partly through a good supporting environment that gives views of children and parent functions and partly due to the reactivity of an interpreter. The responsive environment of a functional database interpreter supports a designer by allowing questions to be asked of the program during development. We have shown that sizeable commercial programs are not only feasible but quick to design and create. Our current example (Program X), with a small team of three, took an idea to a product in less than 18 months .

The issue of 'transparency of the interface' has also been resolved in that the schema has the positive advantage of representing a form where the text in the function boxes and parameter lozenges can be easily edited to produce functional extensions or used to construct new functions. The manipulation and editing of the schema, with modern GUI, is as simple as editing text. Move a function box and the arrows change to follow, highlight a box to edit or delete (as in text), press a key and any icon can be placed. The transmission of diagrams is now easy but the text code representing the diagrams can also be transmitted where there might be problems.

The difficulty of 'interpreting graphics' depends upon the graphics. The examples used by Petre and Green were grossly miss-matched to the problems they addressed. Figure 5 shows that better diagrams do exist and may be used in preference to a text representation.

Although we have not explored the issue of the speed in creating diagrams, our experience supports the observations made by Green et al. Oddly, the expert users tend to be slower than novices because the experts spend a lot of time doing minor adjustments to the diagram; adjustments such as getting the lines straight or rearranging for minimum line cross-over or where that is not possible using labelled junctions to span the schema (as used in engineering drawings of circuits). This is an indication that 'interpretation' is a concern for the experienced designer.

¹⁰ It is the case that not all evaluations may conform strictly to the process. Lazy evaluation is designed to avoid unnecessary work. However, evaluation of any kind for any function cannot go to completion unless all the parameters of that function have been evaluated to completion.

The source of the difficulties in the visual programming languages explored was that they were derived from designing fixed physical devices whereas programming in general is different. This was fully discussed in section 1.3.

Finally, the major problem of 'knots' in program structures is created by an excessive use of 'ifs'. We have shown that using a functional programming language through a graphical representation naturally reduces the number of 'ifs' used by an order of magnitude. We have argued that this effect is a property of the functional language rather than the visual interface. However, the real advantages of the graphical notation is that it makes a functional language easier to use.

Acknowledgements

We would like to thank the Warsash Maritime Centre and Portsmouth Technology Consultants Limited in giving us permission to use their source code for analysis. The authors would also like to thank David Gooding for all his support during the development of Clarity.

References

1. **Addis T. R. and Bull S. P.**, (1988) '*A Concept Language for Knowledge Elicitation*'. Proceedings of the Second European Workshop on Knowledge Acquisition. Bonn, pp. 1/1-1/11. June.
2. **Addis T. R. and Nowell M. C. C.** (1990) '*Scaling Up Knowledge Systems: An architecture for the GigaKnowledge-base*'. Proceedings of the BCS Specialist Group on Expert Systems, London, September, ISBN 0-521-40403-7, pp238-251.
3. **Addis T. R. and Townsend Addis J. J.** (1996) 'Clarity: Built-In and System Functions with FAITH Examples', Version 3.6.5, May. for <http://www.sis.port.ac.uk/research/clarity/index.html>.
4. **Addis T. R. and Townsend Addis J. J.** (1996a) '*The Clarity Manual*', Version 3.6.5, May. for <http://www.sis.port.ac.uk/research/clarity/index.html>.
5. **Addis T. R. and Townsend Addis J. J.** (1998) '*A Functional Schematic Interpreter: an environment for Model Design*'. International Journal of Systems Research and Information Science, Vol. , Pub Gordon Breach Science, ISSN 0882-3014, Pages 263-299.
6. **Addis T. R.**, (1985) '*Designing Knowledge-Based Systems*'. Kogan Page/Prentice Hall/Chapman & Hall. published October. Hardback ISBN 0 85038 859 7. Soft back ISBN 1 85091 251 3.
7. **Addis T. R., Pretlove, A. J. and Townsend, J. J.** (1994). '*A Functional Approach to Creating Evolutionary Models for Engineering Design Illustrated by a Bridge Design*.' Proceedings of the 14th Annual Conference of the British Computer Specialists Group on Expert Systems (ES94), ISBN 1-899621-01-6, pp. 275 - 284.
8. **Citrin W.** (1996) '*Strategic Directions in Visual Languages Research*' ACM Computing Surveys, Vol. 28, No 4. December.

9. **Codd E. F.** (1971) Further normalisation of the database relational model, IBM Research Report 909, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.
10. **Crowther W. J.** et al. (1995). 'Knowledge Acquisition for Engineering Systems using Bond Graphs' Research and Development in Expert Systems XII. Proceedings of Expert Systems 95, Cambridge, December, pp41-56.
11. **Davis R., Shrobe h., & Szolovitz P.** (1993), '*What is a Knowledge Representation*', AI Magazine, Vol. 14, No 1, pp 17-33, Spring..
12. **Gilmore D. J. & Green T. R. G.,** (1984) 'Comprehension and Recall of Miniature Programs' International Journal of Man-Machine Studies Vol. 21, pp31-48.
13. **Green T. R. G. & Petre M.** (1996) '*Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework.*' Journal of Visual Languages and Computing 7, pp 131-174.
14. **Green T. R., G,** (1990). 'Programming Languages as Information Structures' in Psychology of Programming, edited by Hoc et al, pub Academic Press, ISBN 0-12-350772-3.pp117 - 137.
15. **Larkin J. H. & Simon H. A.** (1995) 'Why a Diagram is (Sometimes) Worth Ten Thousand Words' in Diagrammatic Reasoning: Cognitive and Computational Perspectives. eds. Chandrasekaran B, Glasgow J. and Narayanan H. N. AAAI Press/ The MIT Press. pp. 69- 109
16. **Maier D.** (1983) '*The Theory of Relational Databases*', Pub. Computer Science Press. ISBN 0-914894-42-0
17. **Mead C., & Conway L.,** (1980), 'Introduction to VLSI systems', pub. Addison-Wesley, ISBN 0-201-04358-0
18. **O'Reilly T.** (1999) '*Lessons from Open-Source Software Development*' Comm' ACM, Vol. 42, No 4. pp 33 - 37.
19. **Petre M. and Green T. R. G.** (1993). 'Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill'. Journal of Visual Languages and Computing Vol. 4, pp55-70
20. **Poulovassilis A.** (1988), '*FDL: An Integration of the Functional Data Model and the Functional Computational Model*', BNCOD6, Cambridge University Press, pp215-236,
21. **Reade C.** (1989) 'Elements of Functional Programming' Addison Wesley, ISBN 0-201-12915-9.
22. **Sutcliffe A.** (1988) 'Jackson System Development' Prentice Hall ISBN 0-13-508136-X
23. **Wegner P., & Doyle J.** (1996) '*Editorial: Strategic Directions in Computing Research*', ACM Computing Surveys, Vol. 28, No 4. December.