

A Functional Schematic Interpreter: an environment for Model Design

T. R. Addis¹ and J. J. Townsend Addis²

Abstract

There are two major reasons for the value of a functional schematic interpreter as a new programming language for constructing models. These reasons are the capacity of a functional language to express knowledge that has a formal representation on a computer and the rich informal semantics that can be attached to a set of diagrams (a schema). It is this informal semantics that help designers manage the complexity of model design. The combination of formal and informal semantics provides an elicitation, a specification and design language that can be run. The effect is the improved performance of system design through a path of minimal error. The design of complex models is shown to be further managed by two other strategies: the introduction of conceptual levels and the segmentation of design into functional domains. The notion of normalised objects is shown to be useful. However, the usefulness of this approach can be limited for some modelling problems. This limitation is illustrated through the design of a Roman Numeral calculator. Finally, the usefulness of a functional schematic interpreter as a specification and project management tool is also proposed.

Introduction

Diagrams as human input to computers are a representation that ought not to hold any particular significance in the range of possible notational schemes; yet they do. Why this is the case cannot be argued from the nature of computers since the sentential form of representation can be shown to express all the relationships necessary for defining computer models. This certainty we have concerning the expressibility of the sentential form is because computers are based upon the Von Neuman architecture; an architecture that is rooted in propositional calculus.

Diagrams cannot even be justified by their accessibility to computational machinery since computer theory and practice presumes that all commands and control arrive as a stream (or possibly a set of parallel streams); neither of which map comfortably into diagrams. The processing of the bit streams from graphical inputs has only become practical because of the advances in technology which provided us with sufficient computer power to compensate for the mismatch. It has been believed from the beginning of computer development by an active minority of engineers that a diagrammatic interface is

¹ Department of Information Science, University of Portsmouth.

² Visiting Research Officer at the University of Portsmouth.

needed. This belief has been justified by the current popularity by computer users of windows technology.

Before diagrams could be easily translated into computer input the use of diagrams for design was accepted. Traditionally, engineers were familiar with diagrams as a product of design producing such things as circuit diagrams, engineering drawings and graphs. Computer programmers and system analysts adopted this tradition and created diagrams to express their own kinds of designs. For computer systems, the initial design is often through representations such as Systemic Networks, Semantic Nets, Bond Graphs (Crowther et al 1995), Sfd graphs (Addis 1985), Petri-Nets, Data Flow Charts, Function Diagrams, Project Data Models, DAPLEX Diagrams, KADS Diagrams and Repertory Grids. These all express models (of different persuasion) within a two dimensional space.

Designs of computer systems that are drawn on paper can be considered as models. The nature of a model is that not only should it provide an abstraction of the significant components of a system but it should show how those components relate. John Casti (Casti 1997) has identified four types of models: experimental, logical, mathematical/computational and theoretical. As for him, we are concerned with the design of the last two types. These last two fulfill at least three roles:

- The *predictive* models such as Newton's laws of motion that describe the motion of particles interacting through gravitational forces. It is through this that the motion of the planets can be predicted.
- The *explanatory* models provide a framework in which observations can be understood; it is a structure that can summarize a range of detail. An example of this is the explanation of how modern languages originated from an ancient language through the movement of people. It explains but cannot give any kind of prediction as to how a language might change in the future.
- The *prescriptive* models are systems of interacting elements where each element represents some observable concept. The purpose of such a model is that scenarios can be played out under different conditions and such intervention suggests what actions that could be taken with the real thing to produce certain effects. An example of this is a model of the national economy where different elements such as investment, incomes and unemployment influence each other.

The advantage of drawing models directly into a computer system is that the coherence of the model can be checked and possibly run (e.g. see Addis et al 1991, 1993). Relational analysis, in particular, uses a diagrammatic scheme to show the elementary items (attributes, objects and entities) that are recognized by (say) a business. CASE systems attempt to translate these (and other)

diagrams into a working database. However, many (but not all) of these diagrams are too informal to translate directly into a formal structure that can be implemented as a program. For these, where the target system forces a representational bias, an extra stage in modeling is required that leads to an implementation that is consistent with the computational paradigm.

The diagram offers the designer or engineer a representation that seems more comprehensible than a linguistic description. It's not that the linguistic description necessarily lacks anything but a simple diagram can be much easier to grasp. The consequences of the interacting components in a diagram can, for many people but not all, be instantly obvious; a clarity that is not available without the application of inference procedures applied to a sentential representation. One can speculate that this insight stimulated by a diagram is using inherent abilities of the human mind; the ability, for example, to infer, at speed, from spatial structures. The main problem is to keep the diagrams simple and yet still able to define completely a complex system.

An approach to retaining simplicity is through a perspective that falls naturally with the separation of different manifestations of a system. These manifestations may be separate design domains. For example the distinction between the engineering of a car as a working transport system, an ergonomic device and a work of art.

Complex designs such as that required for computer 'chips' (wafer fabrication) demands several stages (about 11 in this case). Each stage considers a different aspect of a design. For example, the last three stages of wafer fabrication involve details of the electronics (circuit diagrams) and an intermediate stage (stick diagrams) that translates the circuit into a form that allows an easy transformation into areas of silicon (real estate diagrams). Each stage in this design provides an ontology of constructs that helps a designer towards an artifact that is consistent with the target system. Each ontology is concerned with different aspects of design (electronic -> electronic/area -> area). The quality of the final design depends upon these diagrams. For example, before the introduction of stick diagrams the size of wafers were an order of magnitude larger than the same functions using the same technology with the consequence of poorer performance (Mead and Conway 1980). This is an example of the diagram being the key to the explosion of computer power. Unfortunately, it takes a long time to forge ontologies of this kind, their associated representations and methods.

Another approach to retaining simplicity in design is to consider the system as a set of interfaces. An example is the Open Systems Interconnection (OSI) used to define the communication between computers. This is specified as seven layers, each layer concerned with details of communication within a predefined domain; domains that are described in terms of a range of 'services'. For example the transport layer insulates the upper layers from any changes in

the network hardware by defining a set of operations that are independent of the mechanisms involved in communication.

It is a characteristic of the design of complex systems in that there are different layers of abstraction. These layers of abstraction may not be so well defined or isolated as the OSI structure. However, each layer may be considered to be a formal language consisting of a set of services, functions or procedures. The exact nature of these layers cannot always be predefined and may only emerge as the system develops. Thus one of the major stages in design is a process of identifying these layers (insight), reorganizing the internal structure of the system to reflect the layers and complete each layer with a full set of operations. This will often lead to greater simplicity and generality of the system. Such structures are akin to the *explanation* form of model.

Such layering is only required to reduce the complexity of the system for the designer (or maintainer) and it may bring with it a cost of inefficiency. Other ways of reducing the perceived complexity may be needed that do not effect the system design. This is where a representation rich in informal description potential is useful. Diagrams, for example, are particularly rich in extra dimensions of representations that are not all used in formal specifications. Examples are the spatial layout, the size and colour of the set of tokens. All these dimensions can be used to provide informal meaning to the formal structure.

In summary, a complex system can be managed through at least three possible techniques. These are:

- The introduction of conceptual levels
- The segmentation into functional domains (aspects)
- The introduction of informal semantics

All these techniques can be used together in order to establish some kind of overall grasp of a complex system.

Stages of System Design

An argument can be put forward that supports the approach of designing complex systems through the use of a functional schematic environment. Such an environment is embodied in Clarity³. Clarity is a *functional schematic programming language* and provides a programming environment that allows a user to draw a program as a set of directed graphs. The term *schema* has been used to avoid confusion with languages that

³ A Beta test version of the Clarity schematic environment is available on request from the authors or via the web site <http://www.sis.port.uk/research/clarity/index.html>.

generate pictures rather than use graphs to represent a program (or model). The term *schematic* is drawn from the traditions of engineering where the diagrams that represent electronic circuits or those of physical objects are often referred to as *schematic drawings*. We will thus take *schema* as a set of pictures or graphs that represent a program or working model. A *schematic* is taken as a system of tokens and structuring rules that expresses a program, model or concept; its a graphical language.

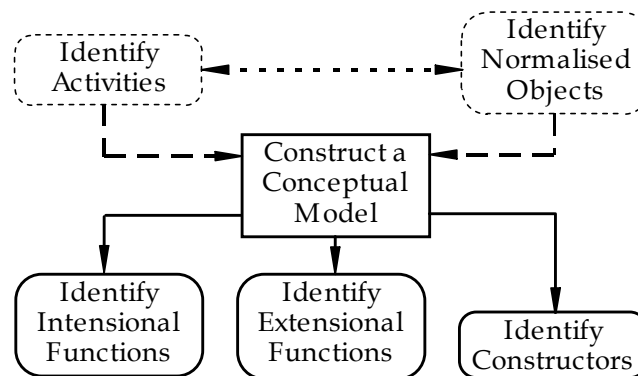


Figure 1. Approaches to analysis

Computer system's design has been well served by relational analysis; an approach that can be harnessed from an object orientated perspective. The notion of normalised objects (see **figure 1**), always in the context of activities and the objects that are recognised, provides the key to the construction of a model and in particular it is referred to as a *conceptual* model. A normalised object is characterised by having a set of attributes that identify it from any other object of its type. These attributes are usually artificially assigned by the analyst. The most convenient means of assigning identifiers that are unique is to use numbers - such as part number. The other attributes if they exist are specific to the description of the object and are expected to be 'simple' values. However, an attribute may be an identifying attribute to another object. It is through these 'foreign' identifiers that the dependencies between objects can be recorded.

Such an approach leads to coherent system design. A less common approach and one that is not well documented is through the analysis of activities. These activities require a description of what they act upon and what the actions produce. Such an analysis leads to the emergence of normalised objects.

The analysis provides a framework in the form of this conceptual model and this conceptual model can be translated into a functional language Faith and in our case this is generated via the schematic interpreter Clarity. The important decisions in a functional representation of such a model are what elements of this model become functions and what elements are

constructors. Some of these functions will naturally form extensional functions that may or may not have a general component (see figure 1).

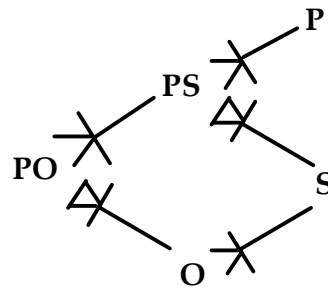


Figure 2. A Conceptual Model of a purchasing system (An Sfd graph)

Figure 2. is an example of a conceptual model where **P**, **S**, **PS**, **O** and **PO** are normalised objects (i.e. conform to normalised relations). The conceptual model shows a set of dependencies or constraints that must exist between the elements of the represented sets. Thus, **P** represents a set of parts where each part p_i consists of a set of attributes (an n-tuple) that describe that part, **S** represents a set of suppliers in the same way. So we can say that $p_i \in P$ and $s_k \in S$. where **PS** is the link between parts and suppliers such that $p_i s_k \in PS$ recording which parts are obtainable from which supplier. This link can also be taken as a normalised object in its own right and as such will have attributes specific to the linking. In this case it might be the attribute 'price'. **O** are the orders to suppliers such that $o_j \in O$ for the parts given in **PO**. Thus $p_i o_j \rightarrow o_j$.

The symbols that connect them show the mappings zero/many to one and the one/many to one. Thus an order (o_i) uniquely identifies a supplier (s_i) although a supplier may have zero, one or many orders. However, an order has to refer to at least one part otherwise it is not an order.

Faith is a direct derivative of lambda calculus and can be represented (formally) as a set of diagrams that have a rich informal interpretation potential; a potential that can be used to aid understanding and hence improve the design or help support more complex systems. The relationship between the diagrams supported by Clarity and the formal language Faith is exact. Further, the diagrams command the creation of a multi-layered abstraction of virtual mechanisms and because diagrams encourage simplicity (since too many items can confuse but a simple drawing is insightful) each mechanism is easy to understand.

Consider a the normalised object:

Part[PNo : Description, Size]

which represents parts in a database and records the parts available in store. PNo is the identifier attribute (indicated by being to the left of the colon :) and

is a number. This relation can be represented relationally as a function 'part'. The extensional component of a functional description⁴ in Faith of a set containing only two parts would appear as:

```
part #45672 ::= Part_Own (Description "Nut") (Size #32);
part #3214 ::= Part_Own (Description "Screw") (Size #24);
```

and this could be stored as a **part** relation with the value:

<i>part_no</i>	<i>part_own</i>
#45672	Part_Own (Description "Nut") (Size #32)
#3214	Part_Own (Description "Screw") (Size #24)

where **bold** all lower case represents a function, **Bold** initial upper case a constructor and *bold* italic a type. Values are signified in normal case and are type marked with # for integer and " " for string. ?0 represents a variable. This representation is consistent with representing an intensional function that has an extensional component (the terminating condition) such as **factorial**:

```
factorial ?0 ::= multiply ?0 (factorial (sub1 ?0)) ;
factorial (#1) ::= (#1) ;
```

It is thus possible to have many pattern sensitive components of a function that override the general case. The patterns are chosen by a 'best' fit algorithm. Moreover, since a functional language is used, it is possible to model at any level of abstraction, combining different levels of complexity in the same representation. Further description of this language is given latter in the paper.

A function in extension can be considered as a relation. For extensional functions the equivalencies are shown in the table below:

Functional Data Language	Relational Database
function name	relation name
parameters (domain)	key domain
output (co-domain)	own domain
constructor or function	domain or domain group (e.g. own) generator

Equivalent representations between a Functional Data Language and a Relational Database

A function can be tested with examples by 'querying' thus:

```
QUERY> part #3214
      (Part_Own (Description "Screw") (Size #24))
```

⁴ The language used here is FAITH which is similar to MIRANDA and ML.

QUERY> part #4444
(Unknown_Part #4444)

The complete Faith code will be:

```
tdec
part_no ::= typeop #0 ;
size ::= typeop #0 ;
description ::= typeop #0 ;
part_own ::= typeop #0 ;
;
cdec
Size ::= int->size ;
Description ::= str->description ;
Part_No ::= int->part_no ;
Unknown_Part ::= part_no ->part_own ;
Part_Own ::= description ->size ->part_own ;
;

fdec
part ::= part_no ->part_own ;
;
fdef
part ?0 ::= Unknown_Part ?0 ;
part #45672 ::= Part_Own (Description "Nut") (Size #32) ;
part #3214 ::= Part_Own (Description "Screw") (Size #24) ;
;
```

Specifying the Ideal

The term 'designing a system' used in this paper means the creation of a model of a system that has one or more abstracted (conceptual) layers and that this model will lead directly to an implementation of a full scale version of that system. The model of such a system must be represented in such a form that there are well established procedures that convert any well formed model to a physical realisation. Six further ideals to which the model should comply are that:

- there should be automatic methods that ensure the well-formed character of a model so that syntactic and structural errors of the modelling representation are at least minimised and preferably eliminated. Type checking is a good example of such methods.
- the model should be based upon a well established formal calculus: well established so that the extensive experience of practitioners can be drawn upon; a calculus because it is complete and hence all the operations that are necessary and sufficient are defined. A further advantage of a formal calculus is that it provides inference protocols that open up the opportunity for proofs and theorems. Function based languages are a good candidate since they are founded on clear principles; principles that underlie mathematics and the construction of calculus in general.
- the model should be capable of being run. This ensures that the dynamics of the design can be checked against a wide range of situations and in a way that can be understood by a client or user of a final system. The language should incorporate the semantics of the computational mechanism on which it is to be interpreted.

- the model should provide the possibility of components that can be interchanged with their functional equivalence. Thus if there is a more efficient or a more general version of a particular function then there should be no difficulty in its replacement. For example, an extensionally defined function should be easily replaced by its intensional equivalence⁵.
- the representation of a model should provide a flexibility or additional degrees of freedom to encompass an informal (i.e. human interpretable) semantics. Examples of this are the use of indentations and annotations to link symbols with meaningful concepts (external signs).
- the representation should encourage ordered structure and clear organisation. In particular, it should provide mechanisms for 'conceptual layering' (see below).

Such a model for design is considered to have conceptual layers because its basic components at some single comprehensive view must have a clear interpretation with reference to the system to be designed; the components represent concepts. It is layered because each view deals with one of a series of independent aspects that lead from the outward appearance of a system to the inner structure and finally the physical implementation (cf. OSI). What makes it a model rather than a specification is that the concepts at each level are mutually constrained in a way that reflects the final system. Models of this kind are an embodiment of the designer's and client's (e.g. user's or associated expert's) knowledge of a part of the world that describes a construct. This knowledge can be used to predict events or provide a basis for action.

Abstracting to a Representation.

The above notion of modelling ignores much of the human activity of design. Design is not an isolated activity nor does it usually proceed in an orderly fashion. Further, many design activities involve a considerable amount of visual sketching and diagrammatic constructions. The design of a knowledge system for example usually starts with the investigation of the experts' knowledge. It is a process that starts with elicitation protocols and terminates with computational procedures.

Figure 3 shows the general processes linking elicitation, modeling and formal representation (modified from Shaw and Woodward 1990); it illustrates

⁵ An extensional function is one that is expressed as a look up table. Examples are 'times tables', 'logs' and railway timetables. An intensional function is one that involves the use of other functions. Examples are the formula for calculating the hypotenuse of a right angle triangle or the procedure for using logs for a particular calculation.

experience most of the techniques and methods of these stages have evolved empirically in response to the needs of existing and now outdated technology.

The stage of creating a conceptual model from the different kinds of objects identified is a process of adding further constraints between objects (Semantic Functional Dependencies -Sfds). The first stage is to determine the ontology of the world and populate it with normalised objects. The second stage is to infer hypotheses about the behaviour of these objects with respect to each other. Thus the existence of one kind of object should predict or constrain the existence of another kind. However, there is another route to design by taking the activities as primary and inferring the normalised objects from them (see figure 1).

The conceptual model needs to include many more types of constraints than simple mutual existence functions. More complex functions that may involve formulae or dependencies that involve calculations need also to be expressed. The functional language can provide this but functional calculus, in its current form, has not been found easy to learn except by those who are already skilled at abstract mathematics.

It is proposed that diagrams can be used as a more comprehensive and perhaps a more universally acceptable representation for the functional calculus to express the third stage of design. This does not mean that the diagrammatic representation is obvious to a naive observer. The purpose behind the use of the diagrams is to provide a rich representation that has the potential for presenting information beyond that of the formal semantics of the functional language. This extra dimension of informal semantics provides a human interpretable form that clothes the formal representation with greater meaning; it gives sense to the symbols.

Formal Diagrams and Clarity

It is the practice of most engineers, systems analysts etc. to use sketches and diagrams as aids to implementation. However, accounts assume that all diagrammatic modeling reduces to a linguistic representation of some kind.

Formal representation is essential to computation, nevertheless formal notation may prematurely displace informal diagrammatic working during the elicitation process (Shaw & Woodward's summary 1990 of the process suggests that this is the case). Addis et al give the diagrammatic representations used in knowledge elicitation a more prominent and more enduring role by showing that it is possible to combine the accessibility, flexibility and exploratory capacity of diagrams with the disciplines of formal representation (Addis et al 1993). It is conceivable both to be formal and informal at the same time because it is now possible to generate program code directly from diagrams. These diagrams are the Clarity schema and can be interpreted directly into the functional language Faith.

Such a representation we have referred to as a schematic. Schema diagrams of this kind are designed to generate computer interpretable code. If the code is a functional database language then this generated code enables the coherence and consistency of the diagrammatic representations to be checked. Further, the code can be used to create the dynamic components of knowledge in the form of procedures, working models and programs. This is a functional schematic programming environment that represents a point where knowledge representation meets its implementation as a program.

The graphs of the schematic Clarity are constructed from a small set of tokens (e.g. oblongs and lozenges) linked together with arrows. The schema components (called graphs) in this case are constructed in windows (currently under an Apple Mac operating system). The 'function window' is one of three windows that uses the tokens oblong and lozenge each in their three different forms. Each of the forms of a token has a specific meaning as shown in the following table:

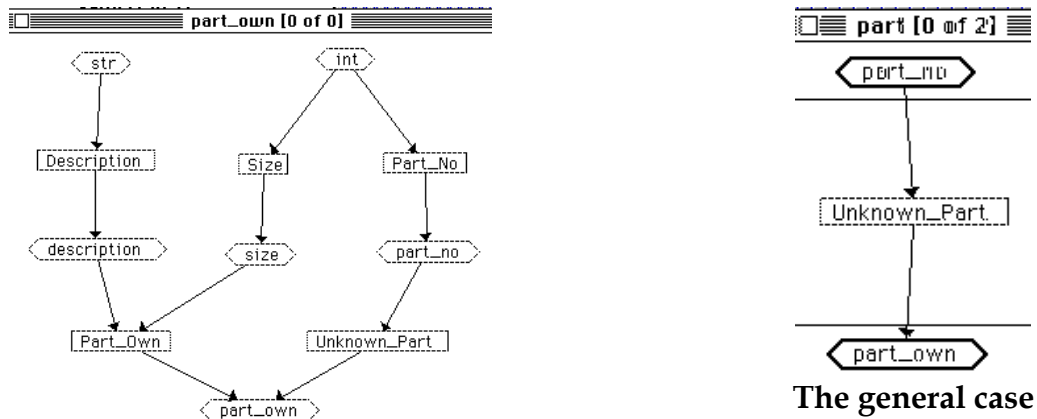
Tokens	Dotted Line	Plain Line	Bold Line
Oblong	Constructor	Function	Recursive Function
Lozenge	Type	Type Held (HOLDN) ⁷ .	Parameter and Constants

Table 1. The range of tokens for Clarity

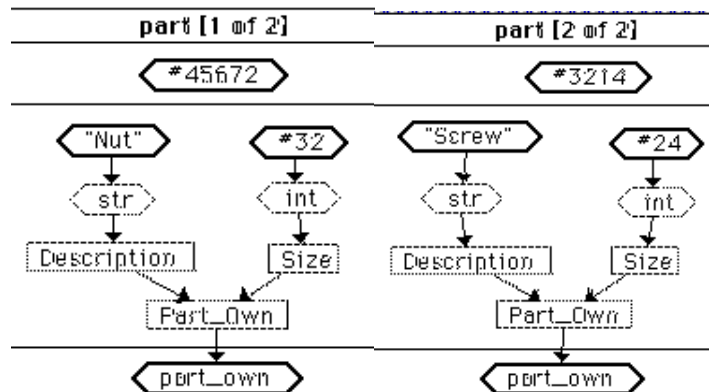
The simple principle behind a function declaration and definition is that every function needed for a definition can be represented by an oblong with the function's name inside; the name is put in by the programmer. There are currently about 250 library functions (see Clarity Built-In and System Functions with Faith Examples (Addis and Addis, 1996)).

There are three different kinds of schema: *declaration*, *function* and *network*. In general, a declaration schema introduces new types and constructors, a function schema declares and defines a function and a network schema displays the dependencies between functions (i.e. a 'called by' graph). The declaration and function schema are illustrated below showing how the 'part' example may be represented in Clarity.

⁷ Equivalent to the function *let*



And the specific cases thus:



Faith the underlying interpreted language is based upon a *typed functional database language*. The form of Faith is similar to Miranda or ML. It depends upon the principle of 'best match' for a function component to be activated rather than the order it was defined. This 'best match' allows interactive development of a program because the components of functions can be defined at any time and in any order. Faith functions are normally stored on disc as a database and retrieved and run in main memory. Currently, all of the database is loaded into main memory thus avoiding retrieval problems at the cost of maximum program size.

Clarity generates Faith code from the schema and conversely the schema can be generated from the Faith code. Programs can be written as a mixture of both. It is the Faith code that is interpreted into computer instructions; it is Clarity that does most of the type checking and programmer support.

The main purpose of the design was to provide a supportive environment for a programmer that matches an interactive approach to modelling or programming. It was constructed to support, with on-line assistance, all the methods of an experienced functional programmer without restrictions. The restrictions are only imposed against fatal programming decisions otherwise only warnings are given for bad practice.

Three Levels of Memory

The Clarity environment consists of three levels of memory and to understand what is happening depends upon keeping these three levels in mind. The program or model is normally stored in main memory but before this can be done the program or model has to be created or loaded into main memory from a database of such programs or models.

The initial stage of creation is to construct a model in one or more of the windows in which the different types of schema are constructed: Faith, Declarations, Function and/or Network. The information placed in these windows is temporary and not linked with the model in any way at this stage other than through consistency checks. The 'running' model is kept in main memory ('Store' in figure 4). To include the construct created in one of the windows into the model is the act of 'commit'. This action will translate the Faith Code, or schema into the store to be integrated into the model. However, even this is not permanent and to ensure that the model is not lost it has to be 'saved' to the database. Once this is done the model is secure.

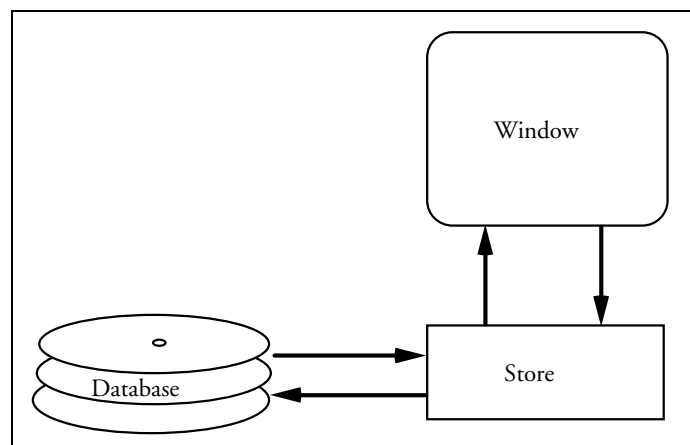


Figure 4. The Three Types of Memory

The Network window is unusual in that it is principally concerned with describing the state of the model as expressed on the database only and not in the main memory⁸. The Faith window describes the state of the database in the first instance and after every 'save'. It also maintains a record of the state of the functions committed through the schema windows (Function and Declarations) in between one 'save' and the next.

For the creation of a program or model there are three classes of window that can be identified:

1. *Model* ... Includes all those windows that provides mechanisms for creating a model.

⁸ However, it can be used in reverse to place function frames into main memory.

2. *Control* ... The window that informs about the design environment and provides input to the model.
3. *Side Effects* ... The windows that provide output of the model.

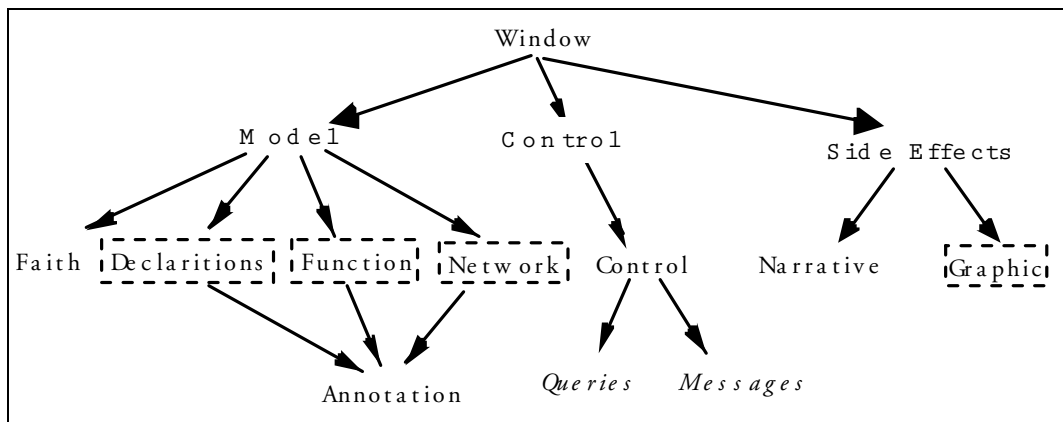


Figure 5. Windows and their role

The above figure 5 shows the complete set of windows separated into their roles. The dotted boxes indicate those windows that are essentially graphical and the others are text. The program or model is normally formed in the declarations and function windows. There is an annotation window that contains text that is tied to a specific window and also the tokens in that window. This provides a means of keeping detailed comments, descriptions and explanations on the model. The annotation window is the key to using Clarity as a specification language and project management tool (see later).

Data Types.

There are nine basic data types defined for modelling and these types are distinguished from user defined data types in that their predefined constructors are integrated as part of the value. Thus the integer constructor # abuts to a number thus: #3462 or #-7144 for positive and negative numbers. Boolean values are just an enumerated type and hence the values are constructors. Strings and characters have the common syntactic form found in most languages. List uses the constructor : which is sometimes mistaken for a function by users who have Lisp as their first language. The normal way in which a user may write a list is within square brackets (e.g. ['a' 'b' 'c']).

A variable on the other hand fulfils many roles. In the first instance it can be used to indicate within the body of a function definition a parameter position (e.g. the first parameter would be referenced as ?0 and the second as ?1 etc.). Alternatively, a variable can be used as a generic type (e.g. List ?3) showing where different but unspecified types should match. It is also used to indicate a general element in a pattern within the input parameters of a function (e.g. (: ?2 ?3) where ?2 is the head of a list and ?3 is the tail).

DOMAIN	TYPE	CONSTRUCTOR	SYNTAX	EXAMPLES
Integer	int	#'	#n where n is a number	#4 #125 #-36
Real	real	#r'	#rn where n is a number	#r4.3 #r46.32 #r-14.73
Boolean	bool	True, False	True, False	True, False
String	str		" "	"All good men"
Character	char		' '	'A' 'a' 'x'
List	list	:'	(: (: .. (: nil)..) or [] where .. means repeat pattern for each item in list.	(: 'A' (: 'B' .. (: 'c' nil) ..)), ['A' 'B' 'C' 'c'] [#3 #45] Note that items in a list must be all the same type.
Variable	Var	?	?n where n is a number	?0, ?1, ?2 Note that the arguments of a function can be referred to by a variable. The first argument is always ?0. Sometimes a referential confusion can occur between variables used within an argument pattern and an argument. This can be avoided by using a ?n for a variable such that n is > number of arguments.

Table 2. The main preset data types of Faith & Clarity

Using the Functions window.

The function window is a palette on which a function can be declared and defined. The generic input and output parameters declare the function and the body of the function is its definition. All functions have to be declared and it is possible to declare a function without commitment to its generic form.

A generic form of a function is the component of a function (if it exists) that has all its input parameters specified as a type (e.g. as int or str etc.) and there are no actual values such as #3 or "Fred". Output parameters can only be specified as a type.

The input of a function are the arrows from either the output of a function or a parameter and the output of a function is a single arrow directed outwards to another function, constructor or output parameter.

- *The order of the parameters of a function or constructor is strictly ordered from the single output arrow (as reference start) in a clockwise direction.*

Although a function may only have a single output it can provide many inputs via a lozenge. Once a function has been declared and defined it may then be used actively in the declaration or definition of other functions. In practice it is possible to use functions before they have been declared or defined but only as place holders.

There are other tokens or aids that can be used to make a function or type declaration and definition clear. These do not provide any further functionality but do provide a means of assigning informal semantics to the

schema (the graphs that represent a program). These include labelled join nodes and colour.

The Roman Numeral Calculator: An exception?

The normalized object approach does not always fit easily into the design of some models. For example the design of a Roman Numeral calculator where we have imposed the following restriction on the design in that no arithmetic library functions will be used. We will only use the most primitive library functions that will allow us to find equality, manipulate strings and the conditional function *if*.

The problem here is “what are the normalized objects?” There are only two objects that can be identified and these are the symbols used to express a number and the number itself (as an abstract idea). The important link between the numbers is that they are ordered and ordered precisely by the simple separation of the digit “I”. The concatenation of symbols that identify each number develop from this notion of order and produce a family of compound symbols. However, the generation of the sequence of symbols pass several ‘threshold’ forms. These threshold forms show how the symbol changes as the numbers increase. The first change for example (see figure 12) is that “IIII” maps to “IV”. Most of the system is to do with the mechanism for progressing the order and relating this mechanism to the numerical identifiers. The expression of order and the access to the way in which the objects are generated lie outside the comfortable descriptive power of normalized objects.

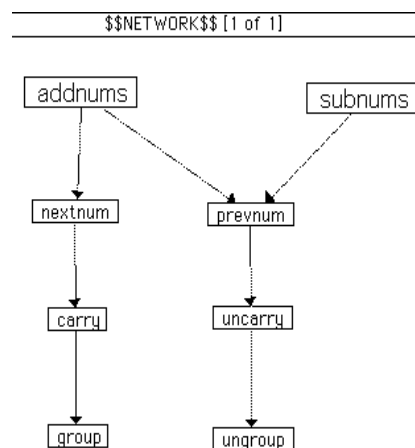


Figure 6 The Overview of the Roman Numeral Calculator

Figure 6 shows a filtered overview as a network of ‘used by’ links of the final model. In particular it shows at the top level functions *addnums* and *subnums* which add and subtract two Roman Numerals respectively. We will only consider the recursive function *addnums*.

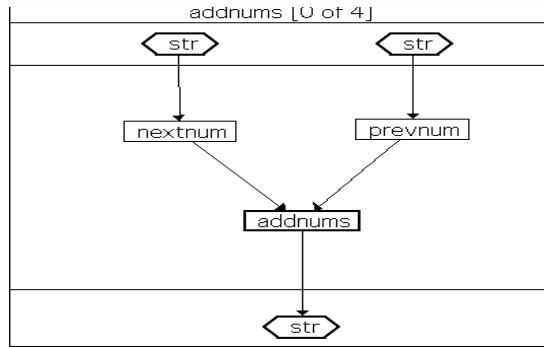
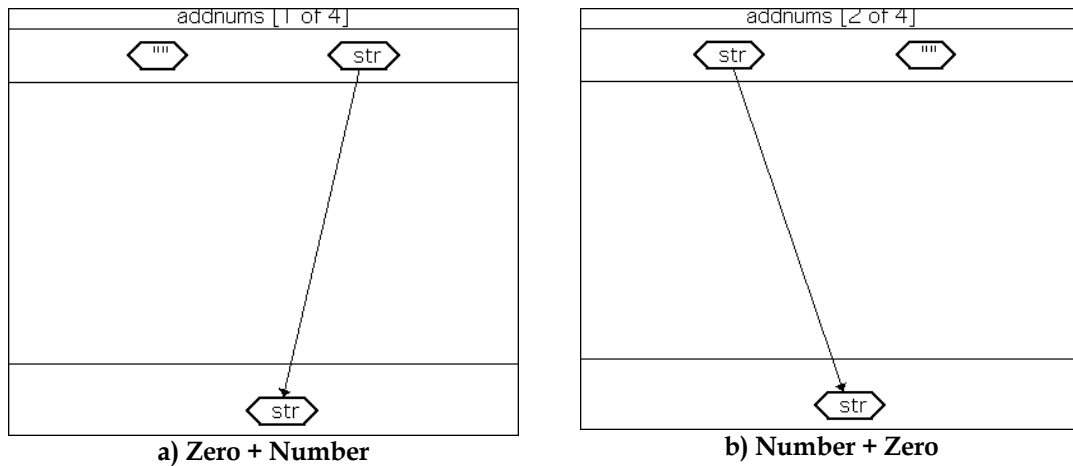


Figure 7. Adding two numbers together

Figure 7 indicates the very simple approach to adding two numbers together. Here we simply subtract one from one number and at the same time add one to the other number. This continues because of the recursion until one of the numbers reaches zero (the empty string "") and the other number will be the answer.

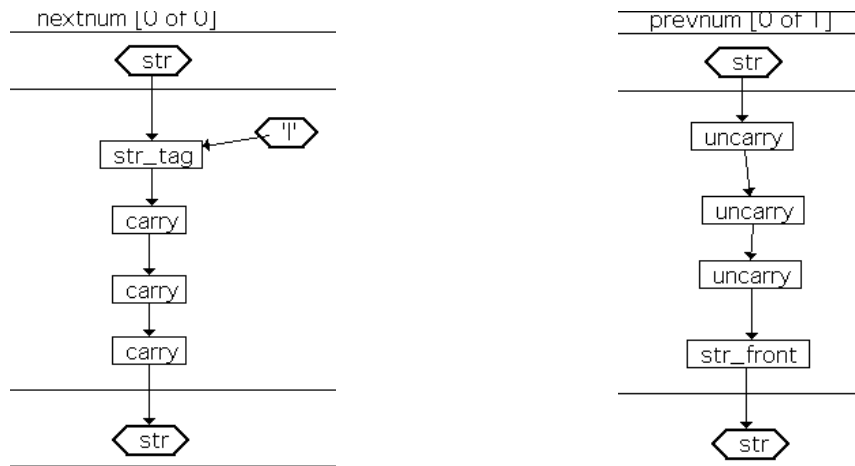


a) Zero + Number

b) Number + Zero

Figure 8. The two special cases of addition

The figure 8b shows the stopping condition for *addnums* (figure 7) whereas figure 8a saves the calculation if the first number is zero anyway.

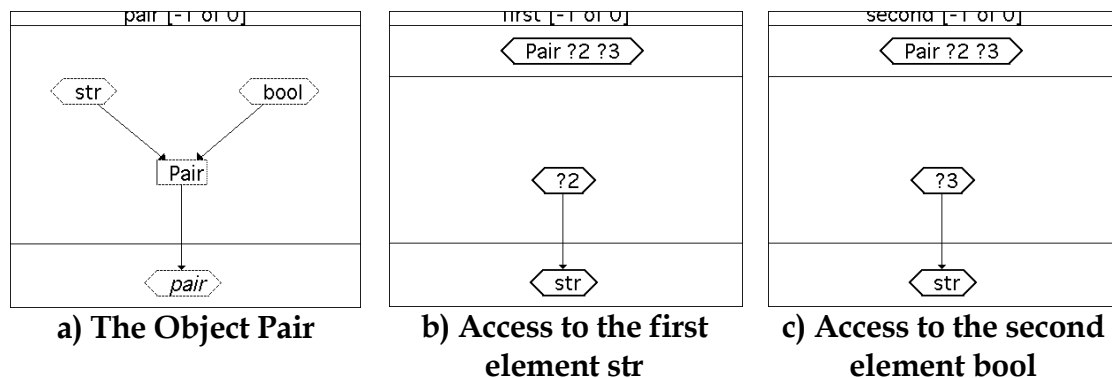


a) Adding One and rippling through the changes

b) Removing One and rippling through the changes

Figure 9. Adding and removing a unit and maintaining the correct form

The process of adding one depends upon maintaining the correct form at all times. The function *carry* is used for rippling through the addition of one and the inverse function *uncarry* is used for the subtraction of one. The library function *str_tag* adds the unit “I” onto the end of a string. The *carry* ensures that the form is correct and it is carried out three times because each deals with units, tens and hundreds. There are no more reduction forms after “M”. The general approach would be to make *carry* recursive at this level.



a) The Object Pair

b) Access to the first element *str*

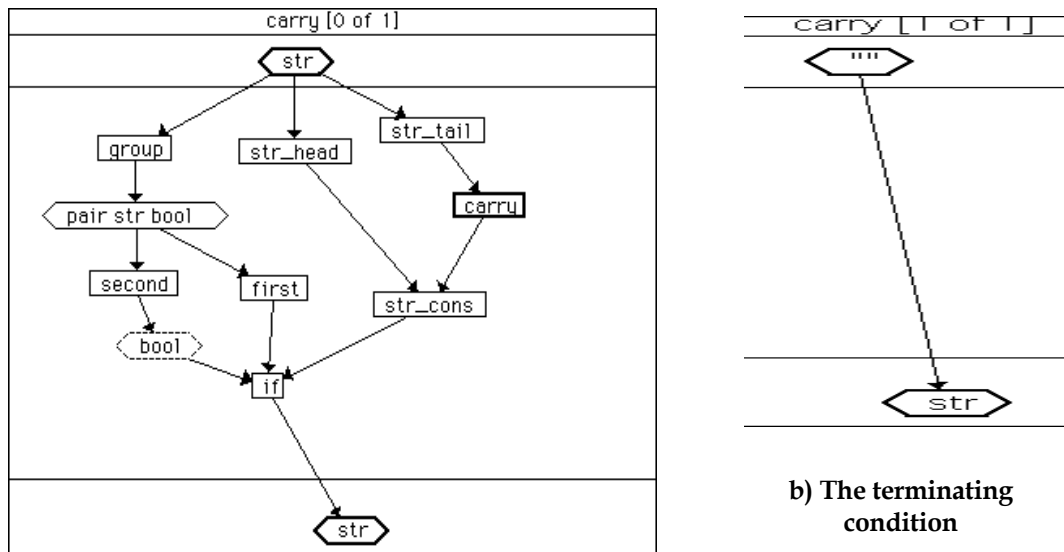
c) Access to the second element *bool*

Figure 10. Declaring an the Object Pair

The form of a Roman Numeral has to conform to a simple syntax. In order to show that a number conforms we need to mark it in some way. This we can do by creating the constructor *Pair*⁹ that produces an object type *pair*. With *Pair* two other functions are needed to access the information in the construct; these are *first* and *second*. Note how they are defined by a simple reference to the pattern of the input parameter and that the **Pair** in conjunction with the access functions (*first* and *last*) is an object in the object orientated sense. This will be used by the function *group* which rewrites the **Pair** such that if it needed changing to conform to a correct Roman Numeral it would indicate

⁹ Pair is in fact part of the predefined constructors. However, we are using it here to illustrate the process of declaring a type.

this change by True. Otherwise if there is no change then the Boolean part would be False.



a) The recursive function that ripples changes through the number

Figure 11. Transforming a number into the correct form

The function *carry* illustrated in figure 11a and 11b is a recursive function. This function uses the test *group* to determine if the number is in the correct form. The result is a **pair** that will contain True or False as the second element. Note the automatic use of a HOLDN lozenge thus ensuring that the calling of the function *group* occurs only once. The *if* function acts as a switch that is sensitive to Boolean values. On the first parameter of *if* being True then the second parameter is the output of *if* otherwise the third parameter is the output. Thus, in this case, the well formed numbers will pass through without further processing.

Those numbers that are not well formed are open to further processing by stripping off the first character (*str_head*) and processing the remainder by *carry*. This will continue until the empty string as shown in figure 8b where it simply returns the empty string. The function *carry* thus uses the function *group* to rewrite the string. If it is rewritten then it is passed on otherwise there is an attempt to rewrite the tail. For example the process can be illustrated thus:

XVIII -> *group* -> (XVIII, False)
 -> VIII -> *group* -> (IX, True)
 -> X, IX -> *str_cons* -> XIX

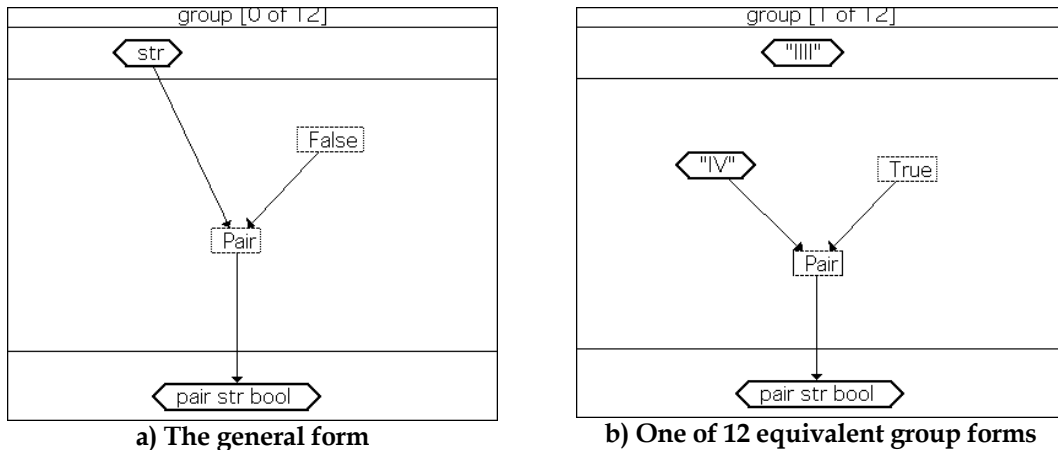


Figure 12. Transforming a small group into its equivalent form

Two examples of the different patterns of group is shown in figure 12. Figure 12a is the general condition and figure 12b is one example of the twelve different patterns that are recognized as well formed. In this case "IIII" becomes "IV".

Some tests of the Roman Numeral calculator are given below:

```

QUERY> addnums "MCMXCIX" "I"
"MM"

QUERY> subnums "MM" "III"
"MCMXCVII"

QUERY> map (addnums "XXXVII") ["I" "II" "III" "IV" "V"]
["XXXVIII" "XXXIX" "XL" "XLI" "XLII" ]

QUERY> map (subnums "LII") ["I" "II" "III" "IV" "V"]
["LI" "L" "XLIX" "XLVIII" "XLVII" ]

QUERY> foldr addnums "" ["I" "II" "III" "IV"]
"X"

QUERY> addnums "XXV" "XXV"
"L"

QUERY> subnums "MDCLXVI" "VII"
"MDCLIX"

```

The choice of names for the functions used in this example are meaningful primarily to the designer where in this case the layout of the diagram suggests the temporal flow of events. The graphs (the schema) are not always obvious to the casual observer since the choice of how the informal features (e.g. names, position and placement) are interpreted depends upon the designer's culture and habits. However, the connectivity and the order of the arrows around a function box defines precisely the formal interpretation; it is this interpretation that is converted into a working program.

The Functional Data Model

The functional data model first emerged in the early 70's (see Poulouvasilis A. and King P., 1990 for an account) at the time when the relational model¹⁰ was becoming an acceptable alternative view based upon a formal calculus to that of the storage architecture model (files, records and access paths). The functional data model considers the world in terms of entities or normalised objects and functions where the functions map entity domains into entity domains. Both normalised objects and functions can be defined intensionally (nested), recursively and/or extensionally (enumerated).

People do not naturally conceive their own knowledge in either of the extreme forms of data and processing (D. Michie 1979). Representation of people's knowledge is best understood in terms of some intermediate form that combines data and rule. Many practical skills use such a mixture: navigators, statisticians, architects and electronic engineers all use tables of data and precalculated results as well as rules. The recognition that processing and the objects to be processed are intimately related is evident by the growing interest in object orientated analysis and its associated systems. However, it should be noted that the distinction between process and storage becomes less distinct and finally vanishes altogether when object orientation is replaced by functions.

Experiments with a functional database language such as Faith have shown that it possesses all of the properties necessary for expressing knowledge-based processes in a clear and succinct manner (Poulouvasilis, 1988). In particular, its uniformity of expression of data and processes ensures that no artificial distinctions are introduced into the system design; it provides a language that is unconstrained by the distinctions between data and its manipulation.

It is the functional database approach that has the representational power to model any declarative knowledge. The functional database retains both the important unification property (pattern matching) of Prolog and the explicit control over the processing strategy without loss of generalisation. Further, it is the function in the form of functional dependency that underlies the important analysis tool of object normalisation and modelling. Consequently this technique can be included in the elicitation process. On the other hand, functional languages are compatible with Prolog but are more primitive; they are more primitive because they are not committed to the full 'logic' paradigm. However, it is this non-commitment that allows the functional language to readily accept a wide range of 'knowledge' representation schemes including the relational representation.

¹⁰ where a relation represents a set of tuples drawn from the cross-product of domains that reflect the possibilities in the world

Specification and Project Management

Functional languages can be used for the specification of systems. The principle difference between a formal specification language and a functional language is that a specification language description is not usually run only 'proved'. Since proof is the major role of a specification language the language has to be confined to a form that is drawn from predicate calculus. Such restrictions are not required for a functional language and as a result the language tends to step away from expressions that are open to normal inference procedures. These maverick expressions are introduced to fully engage the underlying computational system as well as accepting that the world cannot be fully expressed. So, for example, the possibility of redefining functions dynamically acknowledges the fact that we cannot completely specify all the sets and that we live in an open world.

When it comes to complex systems the 'proof' of such systems is still problematic in that only simple systems can be proved correct in some complete sense. Currently, complex systems can be proved in this formal way only in part and the practical 'proof' is in a barrage of tests and lengthy trials.

Clarity, as such, does not provide any formal help other than it embodies type checking; a process that could equally well be associated with Faith. Where Clarity serves the user is in the diagram form of expression; a form that reduces the possibility of syntax errors and encourages simple structures.

Clarity can help in specification by providing a simple framework in which a natural language description (e.g. English) can be associated with every token in a drawing. Linked with the function declarations where input and output types are defined this provides a unique way to partition and specify the work required to create a complex system to a team of people.

The other element in this partitioning of work is that each sub-team, having been given their interfaces as a function declaration, can work independently. The dependency of relying upon another sub-team to finish some component of the system before another sub-team can proceed is avoided to some extent by providing extensional functions that 'stand in' for the component. The extensional functions can 'stand in' for a component since it is the equivalent of a look-up table that provides a sample output given a particular input. For example, if you are waiting for a sub-team to produce a means of multiplying any two numbers then you can use a simple look-up table for a range of number combinations instead. When the multiplication component is available then it just simply replaces the extensional function.

In this way, there is no need to create (and often recreate during the life time of a project) such management aids as a Pert Chart to control a project. The function declaration allows each sub-team to work at their tasks independently of each other.

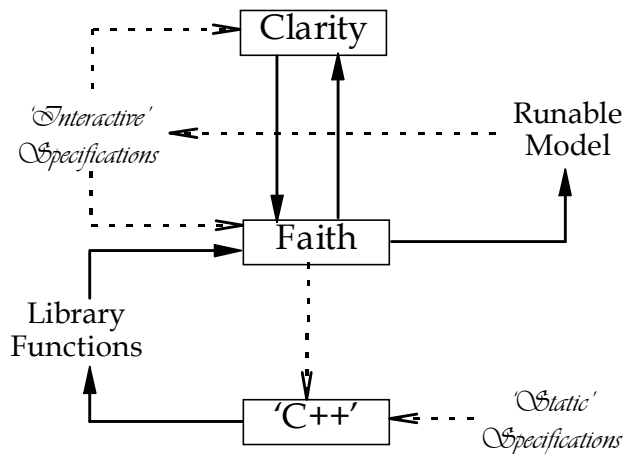


Figure 13. The major Components of the Design Process

Figure 13 shows the next stage in that a working model of a complex system can be tried and tested as a Faith program. However, in most case it will be appropriate to encode the system in an imperative language such as C or C++. This can be done by the Clarity/Faith specification being translated into C or C++ code and incorporated into the Faith library. In this way, the C or C++ code can be tested within the system model and the model itself slowly changes into the final system (when all the system is written in C or C++).

Conclusions

Whenever a new language is proposed the question arises as to what advantages the new has over the current languages. It cannot be argued that the languages in themselves will transform the underlying machinery so that this machinery will perform better or do things that could not be done before. However, if there are advantages then these advantages should relate to the improvement of performance of the language user.

The first advantage that lends itself to the improvement of user performance of an on-line schematic elicitation tool is that it provides the conditions for group design. In particular it allows an expert from another discipline to be actively involved in the creation of a model that represents his knowledge or understanding of a particular domain. Group design, in this case, means the sharing of a common visual field that forms the focus of discussions. Diagrams can be easily altered and substructures of functions can be quickly identified. By 'querying' the code the user can experiment with the model or components of the model at anytime in its construction. If this does not behave as expected, modifications can be made on-screen.

A second advantage for user improvement is that the schematic method provides a disciplined structure of group dynamics. Through this discipline and the controlled manipulation of the common diagrams,

elements of a design may be explored and activated, a construct may be agreed upon and tested, and a consensus may be captured and displayed.

A third advantage of a schema is that it contains a rich informal semantics. The layout of a picture in conjunction with the names, size, shape and colour of the tokens can provide a description that releases the meaning of the model to a user. These 'accidental' features (as opposed to 'essential') of the language extend the formal semantics to an interpretation within the user's world. A related advantage to that given by a rich informal semantics is that complexity can also be controlled through the capacity of a functional language to represent both different conceptual levels as well as different functional domains.

One of our purposes in developing a functional schematic is to provide a design environment that is sympathetic to people and the actual methods they employ in design. It provides a multidimensional space in which concepts may be arranged, linked, organised and analysed. The environment allows a design to proceed from top down to bottom up or anywhere in between. The method acknowledges the iterative approach to a design and allows elements of a design to be only partially defined. This gives an opportunity to create, to explore and to modify total design structures through trial and error. It also provides a powerful specification language and project management tool where the system emerges through a cycle of conversions of Clarity → Faith → C++.

The sympathetic design of a schema will accommodate an environment which minimises the most frequent errors of the kind that normally plague the occasional and the professional programmer. One way this is achieved is through 'typing'. A functional database language is a 'strongly typed' language which means that the objects to be manipulated are classified (typed) and this classification is used to ensure that sensible manipulation occurs within each constructed model. For example, it prevents any attempt by the user to construct a model that might at some stage multiply together two characters (belonging to the class or type 'char') instead of two integers (belonging to the class of type 'int'). This check can be made by a functional schema system before that component of the model is compiled into a working system. Most programming errors have been shown to stem from this problem.

Syntactic errors in programming languages are the second major source of errors. Another way that potential errors are minimised is through the diagrams themselves. Many of these are picked up by compilers but the exact nature of the error is often uncertain because of the compiler centred error messages. The act of constructing diagrams inhibit even the possibility of a syntactic error since diagrams cannot fail to be correct except in the most obvious way.

Because a functional database language is an interactive language each element of a model may be checked independently before being installed. Functions may be defined and tested outside their normal environment and thus their behaviour checked against expectations. This avoids logical errors caused by faulty design.

Usually, those skilled within a particular linguistic paradigm will find that a new language often cuts across their expertise and places them in the unsettling position of a novice. In some cases the skilled student may be worse off than a novice because of their need to unlearn habits that are no longer appropriate. The divide between declarative and imperative languages is an excellent example of how one method of thinking inhibits another. In the end, it is the tacit knowledge gained from experience of a language that is the final judge of its utility.

Acknowledgements

Much of the work on CLARITY has been supported through the projects *Knowledge Design from Natural Science Experiments*. MRC (No SPG 9107137) and *Graphic Programming Methods for Modelling Cognitive & Social Process of Science*, ESRC (No R000235286), investigators: Gooding D. and Addis T. R.

References

- Addis T. R. and Townsend Addis J. J.** (1996) '*Clarity: Built-In and System Functions with FAITH Examples*', Version 3.6.5, May 1996. [for http://www.sis.port.ac.uk/research/clarity/index.html](http://www.sis.port.ac.uk/research/clarity/index.html).
- 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](http://www.sis.port.ac.uk/research/clarity/index.html).
- Addis T. R. and Townsend Addis J. J.** (1996b) '*A Functional Schema Interpreter as an aid to Knowledge System Design*' IEE Colloquium on Thinking with Diagrams, Digest No 96/010, pp13/1 - 13/10
- Addis T. R. and Townsend-Addis J. J.** (1995) '*Diagrams for Design: A Schema Interpreter For Knowledge Systems*', Proceedings of the 15th Annual Conference of the British Computer Specialists Group on Expert Systems (ES95), pp231 - 247.
- Addis T. R., Pretlove A. J. and Townsend J. J.** (1994) '*A Functional Approach to Creating Evolutional Models for Engineering Design Illustrated by a Bridge Design*', presented at Expert Systems 94, pub in Research & Development in Expert Systems XI, BHR Group Ltd. and BCS, ISBN 1-899621-01-6. pp. 275-284
- Addis T. R.,** (1985) *Designing Knowledge-Based Systems*, Originally Kogan Page, now Chapman & Hall. published October. Hardback ISBN 0 85038 859 7. Soft back ISBN 1 85091 251 3., .
- Addis T. R.,** (1989) '*The Science of Knowledge: a research programme for Knowledge Engineering*'. Proceedings of the Third European Workshop on Knowledge Acquisition for Knowledge-Based Systems. Paris (France), July, pp. 49-59.

- Addis T. R.**, (1993) '*Knowledge Science: A Pragmatic Approach to Research in Expert Systems*', presented at Expert Systems 93, pub in Research & Development in Expert Systems X, BHR Group Ltd. and BCS, ISBN 1-85598-020-7.
- Addis T. R., Gooding D. C. and Townsend J. J.** (1991) '*Modeling Faraday's Discovery of the Electric Motor: An investigation of the application of a functional database language*'. Published in the proceedings of the Fifth European Knowledge Acquisition for Knowledge-Based Systems Workshop, Crieff Hydro, Scotland, 20-24 May.
- Addis T. R., Gooding D. C. and Townsend J. J.** (1993). '*Knowledge Acquisition with Visual Functional Programming*'. Knowledge Acquisition for Knowledge-Based Systems, 7th European Workshop, EKAW '93, Toulouse and Caylus, France. September 6-10. Pub Springer-Verlag, ISBN 3-540-57253-8, ISBN 0-387-57253-8. Lecture Notes in AI 723.
- Addis T. R. and Nowell M. C. C.** (1990a) '*Scaling Up Knowledge Systems: An architecture for the GigaKnowledge-base*'. Proceedings of the BCS Specialist Group on Expert Systems, London, September, pp 238-251.
- Addis T. R. and Nowell M. C. C.** (1990b) '*Knowledge and The Structure of Machines.*' Symbols versus Neurons? Stender J. and Addis T. R. (Editors), Pub IOS Press, pp 10-31.
- Casti J. L.** (1997) '*Would-be Worlds: How simulation is changing the frontiers of science*' pub J. Wiley. ISBN 0 471 12308 0.
- 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.
- Jaques E.** (1978) '*Levels of Abstraction in Logic and Human Action*' pub Heinemann, ISBN 0 435 82280 2.
- Mead C. & Conway L.** (1980) '*Introduction to VLSI Systems*' pub Addison Wesley, ISBN 0-201-04358-0.
- Michie D.** (1979) '*Problems of the 'Human Window*'. Presented at the AISB Summer School on Expert Systems, Edinburgh, July .
- Poulovassilis, A.** (1988) FDL: *An Integration of the Functional Data Model and the Functional Computational Model*, BNCOD6, CUP, pp 215-236.
- Poulovassilis, A and King P.** (1990) '*Extending the Functional Data Model to Computational Completeness* EDBT-90 (ref. Dept of Computer Science, Birkbeck College, University of London).
- Shaw L. G. & Woodward J. B.** (1990) "*Modeling Expert Knowledge*", Knowledge Acquisition ,vol. 2, pp. 179-206 .