Knowledge Representation in Systems Modelling

Denham, Michael J.

1987

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7365)/1-022/(1987)

Knowledge Representation in Systems Modelling

M.J. Denham

Department of Automatic Control
Lund Institute of Technology
August 1987

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
1987-08-20

Document Number
CODEN:LUTFD2/(TFRT-7365)/1-022/(1987)

Author(s)
M.J. Denham

Supervisor

Sponsoring organisation
The National Swedish Board of Technical Development (STU contract 86-4047)

Title and subtitle
Knowledge Representation in Systems Modelling

Abstract

This report discusses a knowledge based approach to Computer Aided Control Engineering (CACE). The functionality of a language for communicating system engineering concepts between the user and the knowledge base is discussed. Such a language must have both the flexibility of natural languages and the exactness of formal languages. The importance of modularity is pointed out, and for that reason the knowledge base is divided into three parts: structural knowledge, behaviour knowledge and functional knowledge. A frame based representation of the knowledge base is considered and some experiments using KEE (Knowledge Engineering Environment, trademark of Intellicorp) is reported. Finally, a rulebased approach to how behaviour of an interconnected system structure can be derived from the behaviour of the components is sugested.

Key words
Computer Aided Control Engineering; Computer Aided Control System Design; Software; System Representations; Knowledge Engineering

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
22

Recipient's notes

Security classification

Knowledge Representation in Systems Modelling

1 Introduction

In this report, some ideas are explored concerning the use of knowledge-based methods in the computer-based modelling of dynamic systems. Systems modelling is regarded as the process of creating and reasoning about a computer-based model of a real-world system. The creation of such a model requires the existence of a language for expressing the objects and relationships from which the model is composed. The process of reasoning about the model requires that the objects and relationships in the model are expressed in a form in which some deductive reasoning process is possible, e.g. logic. Therefore statements in the modelling language must be used to create a knowledge base (or bases) which contains the knowledge which is required for the reasoning process. Development of a systems modelling language and an associated knowledge base is part of an overall requirement to build a computer aided control engineering system, the components of which support the total engineering process, including system modelling, stmulation, analysis, design and implementation. These activities are closely integrated and therefore the systems modelling language, viewed as a tool for the modelling activity, must be closely integrated with the other tools which are to be developed, or already exist, to support the total engineering process [1]. In the view of many control system engineers, however, the modelling activity is seen as the most time-consuming and difficult task and the one in which user interaction is at its most intense. It is also the means by which the engineer forms the foundation for the other tasks of simulation, analysis, etc, and the main method of communication of the results of these activities with other participants in the engineering process, eg the customer.

It can therefore be strongly argued that a major part of the time in designing a new computer aided control engineering system should be devoted to the design of the systems modelling language and the associated knowledge base. In this report we will not be directly concerned with the form that this language takes, but with a specification of its functionality. Its implementation, e.g. as a graphical-based language using windows, icons, etc, is an issue which is not discussed here, although in specifying the language we are essentially describing the user interface in an abstract way and therefore the functional specification of the language is driven by our understanding and view of how the user goes about the process of creating and reasoning about a system model.

2 Knowledge requirements for systems modelling

We assume tht the user of system modelling language has knowledge about, or the means of acquiring knowledge about, a system and requires a language in which to express this knowledge and a knowledge base in which to store it. The knowledge about a system can be decomposed into three typoes of knowledge which are closely related:

* structural knowledge
* behavioural knowledge
* functional knowledge

Structural knowledge is concerned with how the system is made up from its various components and subsystems and the way in which these are interconnected, e.g. a table has a top and a number of legs, the legs being connected to the top in a variety of ways, but generally obeying certain rules, such as all the legs are connected to only one side of the table top. In some cases, geometrical or spatial knowledge can affect the way in which the

components or subsystems interact, e.g. a robot moving parts between machines, the legs of a table being connected to the top in each corner, the sensors in a system being geographically distributed.

Behavioural knowledge describes how the system will behave, in an undisturbed situation, and how it will respoomd to external stimulus. The knowledge may be explicit, in the case of components, or implicit, in the case of subsystems and systems built up of components. Here the behavioour is in general inferred from the behaviour of the components and their intercconnections. Thus behavioural knowledge depends on structural knowledge. For example, in the case of a table, we may explicitly state the behaviour of the table top and legs as solid bodies subject to gravitational forces, with a certain rigidity, etc. The behaviour of the table, i.e. its action on being placed on the floor, is determined from the behavioural properties of the components, knowledge about their interconnections and about the connection of the system to the external world, in this case the floor, This behaviour could be inferred and expressed qualitatively, i.e. "the table rests on its legs in stable equilibrium with the table top horizontal", or it might be expressed quantitatively, in terms of forces, positions, angles, etc.

Knowledge about behaviour might also be expressed in either a declarative form, i.e. as a set of axioms which the system must always satisfy, or in a procedural form, i.e. as a set of algorithms from which the behaviour can be computed. In the first case, we might refer to the set of axioms, and the entities and operations refered to within the axioms, as a theory of behaviour. A model is then regarded as satisfying a theory when we create a mapping between entities and operations in the model and those in the theory, and assert that the axioms hold. We will explore this concept further at a later stage.

Functional knowledge is the means by which we will relate the model to the real world system which it describes. For example, when we talk of a table, we naturally have knowledge of what such an object is in the real world. But if we were presented with a model of a table, i.e. a description, perhaps in the form a photograph, drawing or miniature cardboard model, we would not necessarily be able to determine that it described a table. It might be a stool, or a device for punching holes, for instance. We only relate it to the class of objects we call tables when we have knowledge of the function of the system described by the model. By function we mean its purpose, as expressed by its relationship with the environment in which it exists, i.e. other objects, situations, etc. Hence, we know our model is a model of a real world object from the class of tables when we are told that its purpose is for standing cups, plates and other eating utensils on, for example. Of course, a table might also be used for sitting on, in which case we must distinguish between primary or normal behaviour, and secondary or abnormal behaviour. Also in order to use the functional knowledge it must refer to other systems of which we already have knowledge. It is no use knowing that a table is for standing a cup on if we have knowledge of what a cup is.

In some cases, we will be able to verify the consistency between the functional knowledge about a model and its behavioural knowledge. For example, if we know about a cup in terms of its function and behaviour, e.g. that its purpose is to hold liquid and that the liquid remains in the cup only if the latter is within a given angle from the vertical, then we can determine whether the behaviour of our table, e.g. in maintaining a horizontal top, is consistent with its function of having a cup standing on it.

We see therefore that knowledge about structure, behaviour and function are closely related. The purpose of separating the knowledge into these three categories is not only to make it easier for the user to express knowledge about the model and the real world system which it describes, but also to allow the knowledge in one category to be changed without altering the knowledge in another, This is particularly important in the process of creating a system model, during which we often have only partial or estimated knowledge about structure, behaviour or function. As our knowledge becomes greater we may wish to update any of these categories, eg the behaviour of a component, without altering its other characteristics, eg its structure. In separating our knowledge into these categories, we therefore hope to achieve knowledge modularity.

The relationships between the three categories of knowledge which are established during the modelling proces, can however be used to test for consistency betwen knowledge in each category, so that changes, e.g. in structure, which contradict other knowledge about the system, e.g. behaviour, can be detected and reported. The knowledge base creation process must therefore ensure that the relationships between the knowledge categories are stated explicitly or inferred automatically. The latter implies that the modelling process must be based on some fundamental stored knowledge about such relationships, e.g. the procedureto compute the behaviour of two components of a model according to their individual behaviours and their structural interconnections.

3 Language requirements for systems modelling

The major requirements for a systems modelling language are that it should be:

* capable of expressing knowledge about the system, i.e. structural, behavioural and functional knowledge and the relationships between these categories, in a way which is natural to the user, i.e. by referring to entities and operations which are the naturally occuring objects in the vocabulary of the user,

* capable of modifying and adding to existing knowledge, by being able to refer to such knowledge and to new knowledge and carry out such operations as deletion, replacement. addition, combination, etc.

* able to be used for reasoning about the model, including the derivation of properties of the model, consistency checking, etc, for speculating about the model, i.e. adding speculative knowledge and then reasoning about its relationship with existing knowledge, with the ability to return easily to the original state.

* capable of being realized in an executable form, i.e. the model should be capable of being animated and thus reproducing or simulating the characteristics and behaviour of the system which it models.

From these requirements, it can be seen that a system modelling language must have some of the characteristics of both ordinary (natural) language, in relation to its expressive power, and computer programming language, in relation to the need to formally reason, using a precisely defined syntax and semantics, about knowledge expressed in the language and to permit a realization in an executable form. In the former, we accept the possibility of ambigious interpretation and the consequent need for an extensive knowledge of the world and situation in which the language statements are made, in recognition of the ability of the language to express a rich variety of entities and relationships. In the latter, we achieve precision and formality, and require only limited domain knowledge, at the expense of descriptive power. In ordinary language we are able to refer to real world objects and situations based on the possession of a consistent and adequate knowledge base by the

recipient of statement, and to describe structure, behaviour and function in this world. In a computer language, we can refer only to the limited set of objects and relations which can be interpreted in the context of an abstract, computational machine, and describe only the structure and behaviour of objects in this machine, in general under the assumption that we are able to exactly measure the state of the machine at any moment of time.

Since we require some of the properties of both forms of language, it is appropriate to study the relevant concepts and methods which have been considered in the areas of both ordinary language understanding and computer language design. For example, there have been several recent developments in ordinary language understanding based on the study of situation theory and situation semantics [2]. This theoretical basis is also being used to study computer language semantics [3], and there is also a considerable amount of work in progress on the design of system description languages, e.g. [4], [5]. The concepts on which the design of this latter class of languages is based has much in common with those underlying our notion of a system modelling language. In most cases, however, the systems at which these languages are being directed display discrete-event behaviour, whereas we are concerned also with continuous and discrete-time behaviour. In the discrete-event case, for example, the main design issues relating to the behaviour of intercconnected components and subsystems include those of synchronization, fair access to shared resources and the absence of blocking or deadlock conditions. In the continuous or discrete time case we are more concerned with stability, time and frequency response, disturbance attenuation, regulation, etc, and the semantics of the behavioural description of the system must allow us to reason about such basic properties. However at a certain abstract level the distinction is not so marked and much could be learnt from studying the design of the wide range of system description languages currently being developed. The close relationship between a systems modelling language and an ordinary language or a computer language raises the common issue of coping with complexity. In the study of ordinary language understanding, in order to cope with the complexity of the underlying knowledge domain required, methods have been developed for dividing this knowledge into "chunks" of an acceptable size. The frame method of knowledge representation is a good example of this [6]. Here knowledge about a particular object or class of objects in the real world is gathered together into a single unit or frame. The items of knowledge to be stored in the frame are allocated slots, the value of each item being either known or unknown. The process of understanding then involves creating new units of knowledge or new slots in existing units, or simply giving values to previously unfilled slots.

Similarily in computer languages, complexity is handled by the notion of modularity. A module is in general a well defined operation or set of closely related operations which might be realized as a single function or procedure. The design of a program is then principally a process of specifying the constituent modules and their interfaces and interconnections in terms of control of execution and of data flow. Efficiency and productivity issues have also created the requirement for reuseable modules, i.e. modules which can be kept for future use in putting together other programs. In general, such modules have a set of parameters which can be used to fit the module to a specialized situation. A more recent innovation in this context is the notion of object-orientated languages, in which the modules contain not just procedures but also the data variables on which the procedures operate. At this point the notions of object-oriented languages and frame based languages largely coincide, if we consider a frame to contain both declarative knowledge (with explicit values) and procedural knowledge (which describes how to compute the value).

A system modelling language must have a similar capability of coping with the complexity of describing real world systems. It must therefore be able to create and manipulate well-defined units of knowledge about the system. Such units might be the components and

subsystems of the system, their interfaces, their interconnections, their behaviours and their function. We can therefore regard a systems modelling language as a mean of operating on such units, including:

* creating units which define structural, behavioural and functional knowledge about the system,

* combining such units of knowledge by creating relationships between them, and conversely, decomposing such combinations into their constituent units,

* enriching such units of knowledge by adding to the existing stored knowledge or specializing it in some way, and conversely, simplifying such enriched knowledge to a more basic or general form.

Each of these opérations involve the use of certain types of relations. In creating units of knowledge, we assume that the real world systems and the objects from which they are composed will have existing structural, behaviour and functional relationships, e.g. a dining table is a kind of table, a pressure release valve is a kind of valve, which can be used in their creation. That is, attributes of existing knowledge units can be inherited from units to which they are related by the is-a-kind-of relation. In combining units, we create a variety of relations depending on the form of combination and type of units being combined. For example, in a combination of subsystems to create a new subsystem or system, we might be creating both is-a-part-of and is-connected-to types of relations. The operation of enrichment is clearly defining an is-a-specialization-of relation. This is very similar to an is-a-kind-of relation, and we can probably regard them as identical without loss of functionality in the language. The frame method of knowledge representation directly supports the is-a-kind-of relation. However, other types of relations must be constructed explicitly by the inclusion of attributes which embody such relational knowledge.

The relations considered above are concerned mainly with objects within the same knowledge base, i.e. the structural, behavioural or functional knowledge bases. The language must also be capable of creating relations between units in different knowledge bases. These are primarily of the has-behaviour or is-for types, e.g. "a car has behaviour X", "a table is for putting a cup on". In a frame-based system, these relations between knowledge bases will also have to be handled explicitly as attributes.

For uniformity in the use of the language, it should be possible for the operations which the language defines to be used on any unit of knowledge in any of the three knowledge bases, e.g. behaviour X is a part of behaviour Y. We will need to investigate the semantics of these relations.

4 Knowledge about structure

We have defined structural knowledge as being about how the system is made up from its various components and subsystems. We will assume here that the most natural form for defining structure is a hierarchical nesting of subsystems. The top level object is called the system. A system can contain any number of interconnected subsystems. Each subsystem can in turn contain any number of further interconnected subsystems . A subsystem (or system) which containes no further subsystems is called a component. As we shall see later, a component must have an explicitly defined behaviour. Recursion is not allowed, i.e. a subsystem is not allowed to contain itself, although it may contain a copy of itself, regarded as a distinct entity.

Each system, subsystem or component has a number of local variables. Some of these variables are defined as terminal variables and some as parameter variables. Terminal variables have the property of being able to be connected to terminal variables of other subsystems or components. Parameter variables havethe property of being able to be connected to objects called settings and to parameter variables of other systems or subsystems. Any number of parameter variables, from more than one subsystem, can be connected to the same setting. A parameter cannot be connected to more than one setting. Any variable can be connected to an object called a gauge. A variable can be connected to more than one gauge and multiple variables can be connected to a single gauge. Like all other objects, settings and gauges will have behaviours which define the relationship between their internal variables and the subsystem variables to which they are connected, but their primary purpose is to create an interface with the user of the modelling language which will allow the values of parameter variables to be set and observed respectively.

The interconnection between subsystems and between subsystems and settings/gauges is by means of objects called connectors. A connector contains a list of lists of variables, these lists being called connections. A connection has a behaviour which defines the relation between listed variables. The default behaviour of a connection is to make the elements of the connection list equal. Variables are defined in a list by both their subsystems name and their variable name. Simple relationships (behaviours) can also be expressed directly in the connection list using the primitive operations of plus, minus, times, and, not, or. For example, to say that variable x from subsystem s1 is equal to the sum of variables y and z from subsystem s2, we would write the list defining this connection as

(s1.x plus(s2.y s2.z)).

A system unit contains knowledge of its subsystems, terminal variables, parameter variables and connectors. A subsytem unit contains knowledge of its parent system ("is-a-part-of" relation), subsystems and connectors. A component unit is identical except that it has no subsystems or connectors. A connector unit has knowledge of its parent system or subsystem and its connections.

5 Knowledge about behaviour

Behavioural knowledge has been defined as knowledge abut how the system is expected to behave, expressed either as a procedure for computing the values of variables which describe the behaviour or as a theory of behaviour, i.e. a declared set of variables, operators and axioms, which the system is said to satisfy. A procedure or a theory may be expressed either quantitatively or qualitatively.

There are very many possible types of behaviour and the essential feature of the behaviour knowledge base is that the modelling language operations of creation, combination and enrichment should act on the existing units of the knowledge base in order to create new types of behaviour, expressed as new units within the knowledge base. For example, a behaviour which is expressed in a purely symbolic form, e.g. a transfer function, could be enriched or specialized by assigning values to some or all of the symbolic variables (parameters).

A behaviour has a set of sorts, a set of functions and a set of relations which hold between the sorts and functions. A system is said to have a specific behaviour when variables local to the systems are associated with the sorts of the behaviour. In general, there will be a set of pre-defined functions stored in behaviour knoweldge base, but the user will be able to

specify new functions, for example, as specializations of existing functions. Therefore a behaviour might be defined in the following way:

sorts:	U X Y
functions:	f: (X U) -> X
	g: X -> Y
	D: X -> X
relations:	D(x) = f(x u)
	y = g(x)

Here f and g are user defined functions and D is a pre-defined operation. We say therefore that a behaviour is parameterized by a set of sorts and a set of functions. For behaviour to be valid for a component of the system, the functions must be defined for the variables after the association of the sorts with variables.

The operations which are user defined in a behaviour unit are stored as values of slots in that unit, for example as LISP functions, in the behaviour knowledge base. The relations are also stored as values of a slot and the functions are referenced by these relations, Thus the definition of an function can be changed without changing the relations which refer to it. Similarily, the relations can be changed without changing the definition of the functions.

The sorts of the behaviour consists of all those entities, other than function symbols, which are referred to in the definition of the functions and relations. Not all of these sorts will required in forming the association with a set of component variables. Therefore we define two distinguished subsets of the sorts, one called terminal sorts which must be associated with component terminal variables, and another called parameter sorts, which must be associated with component parameter variables. The remaining subset of sorts is known as internal sorts.

6 An experiment in KEE

To investigate the ideas and proposals described in the preceeding sections, we have initiated the development of a knowledge base within the KEE system. KEE is a knowlege engineering environment which uses the frame based method of knowledge representation, as described in section 3 above. The interactive interface to the system allows the user to define units of the knowledge base, the contents of slots and facets (values of slots) and the membership and subclass structure between units. The KEE system generates LISP code which describes the knowledge base and allows reasoning operations to be carried out using either a simple query language or a rule-based system, provided as a part of the KEE system. Procedural knowledge in a slot is known as a method, and is written as a LISP lambda expression. Inheritance of methods between units is according to fairly rigid rules, but there is a wide range of KEE system-defined functions which permit methods to be written to access any component, eg unit, slot, etc. of any knowledge base present in the system. This provides the flexibility necessary to carry out operations on the knowledge base which involve both structural knowledge and behavioural knowledge, e.g. consistency analysis, whilst still maintaining independence of these two categories of knowledge.

At the present time, only the structural knowledge base has been constructed . The form of this is shown in Figure 1. The unit SYSTEMS has declarative slots: connectors, parameter-variables, subsystems, terminal-variables, and procedural slots (methods): show. all.subsystems and show.all. connectors. The latter access items in the knowledge base to determine the subsystem interfaces of a given system as required. The unit SUBSYSTEMS inherits the above slots and has the additional declarative slot part-of,

which contains the name of the system or subsystem of which it forms a part. We have not included the notion of an indivisable part of a system, i.e. a component, in the knowledge base at this point in time.

The unit CONNECTORS has two declarative slots: connections and part-of. The former lists the connections between subsystems as a list of lists of variables which satisfy a simple equality relationship, e.g. if we have three subsystems P, Q and R, with terminal variables W, Y, Z in each subsystem respectively, we write the connection of these variables as the list (P.X Q.Y R.Z). The part of slot names the subsystem of which the connector is a part.

The unit TERMINALS has four declarative slots: description, which is a textual specification of the real-world entity to which the variable refers, e.g. voltage; part-of, the name of the subsystem containing the terminal; range, the set of values which the terminal can take; and type, which specifies the terminal as being either an input, an output or a two-way terminal.

The creation of a structural knowledge base for a specific system is carried out by first copying the fundamental units SYSTEMS, SUBSYTEMS, CONNECTORS and TERMINALS to the knowledge base. The constituent parts of the system to be modelled are then created as subclass units of one of these fundamental units, as in Figure 2. Each subclass unit is named and each of the inherited slots, e.g. subsystems, part-of is given a value. The slots terminal-variables and connectors in the units SYSTEMS and SUBSYTEMS have so-called active values connected to them. These are methods which are initiated whenever the slot is accessed in order to insert a new value. The function of these methods is to enter values into the units TERMINALS and CONNECTORS by requesting from the user the necessary values for the slots in these units. Thus the necessary knowledge for these parts of the knowledge base is elicited from the user as soon as any structural interconnection component in a subystem, i.e. a terminal variable or a connector, is created. The active values SETUP.TERMINALS and SETUP.CONNECTORS appear as units in the structural knowledge base (Figure 1). The simple system shown in Figure 3 is used as an example for the creation of a system specific structural knowledge base (see also Figure 2). The LISP listing describing the knowledge base is given in Appendix 1. The knowledge contained therein allow queries to be made about the structure, e.g. the subystem hierarchy, by accessing the methods contained in the units of the knowledge base, e.g. show.all.subystems. Additional methods could easily be created to provide further knowledge about the structure of the modelled system as required.

The results of the experiment so far show:

(a) that KEE is a suitable environment in which to construct a simple knowledge based modelling tool. It is fast and easy to use, once the initial learning phase is over. To construct the same tool in LISP would take considerably longer.

(b) that the knowledge based approach using frames provides a useful tool for representing and accessing information about the structure of a system. It would be a simple matter to extend the present version to include methods to modify the system structure, e.g. by creating subsystems of existing subsystems, by altering connections, etc, and to reason about the system structure, e.g. test for consistency of types in connections, determine feedback loops, etc.

Further development of the experimental tool in the short term might include the addition of a graphical interface for creating the structural knowledge base, i.e. a block diagram

manipulation interface. This would however be reasonably straightforward and, whilst improving the user interface, would not add to the functionality of the tool. A more important extension would be to add the second component of the modelling knowledge base, the behaviour knowledge base. Some tentative proposals on how this could be done are given in the following section.

7 Extension of the experiment to include behavioural knowledge

The relationship which we want to create between the structural knowledge and the behavioural knowledge is one of a has-behaviour type. This implies that

(a) the behavioural knowledge base contains units which represent the behaviour of systems, either in a procedural from or as a theory of behaviour, as discussed in section 5.

(b) these units can be related by specialization, i.e. an is-a-kind-of relation, which allows simple behavioural knowledge to be enriched during the modelling process.

(c) the units can be combined, according to the structure of the system, so that the behaviour of interconnected subsystems can be inferred from the behaviour of the subsystems themselves, as specified by the has-behaviour relationship value in the structural knowledge base units for the subsystems.

The has-behaviour relation must include a binding of the variables in the structural description of a subsystem to the variables, or sorts (see section 5), in the behaviour description. The operation of combination, through interconnection, of subsystems then implies a combination of behaviours. This is achieved by the addition of relationships between sorts in the individual behaviours, according to the interconnection relationships. For example, consider subsystems P and Q, each of which have terminal variables x and y, representing input and output respectively for each subsystem and parameter variable g. Assume that there exists a behaviour called CONSTANT-GAIN which has sorts A, B and K, function f: A->B and relations b=f(a), f(a)=k*a. A and B are designated terminal sorts and k a parameter sort The has-behaviour relation value for both P and Q specifies CONSTANT-GAIN as the behaviour and makes the bindings x to A, y to B and g to K in each case. Assume now that subsystems P and Q are connected,with a connection in which P.y=Q.x. This will have the effect of producing a combined behaviour for the pair of interconnected subsystems which has

sorts:	A, B, K, A', B', K'
functions:	$f: A \rightarrow B$ $f': A' \rightarrow B'$
relations:	$f(a) = k * a$ $b = f(a)$ $b = a'$ $f'(a') = k' * a'$ $b' = f'(a')$

This is the simple combination of the two constituent behaviours, together with the additional relation, $b=a'$, imposed by the interconnection equation P.y = Q.x.

Clearly, such a combined behaviour does not have to be explicitly constructed but can be inferred from the knowledge of the behaviours of the individual subsystems and their structural interconnections. In order to reason about the combined behaviour, we must have knowledge of the individual behaviours, of the additional relationships imposed by the interconnections, and of a set of rules for combining behaviours in more complex ways than in the example given below. This latter aspect is very important, and implies that the behavioural knowledge base must possess a fundamental component wihch consists of a set of rules and a set of methods, which together constitute a theory of combined subsystem behaviour. To combine behaviours, it will be necessary to access the rule base to determine whether the combination can be create, and the methods to create the necessary new sets of sorts, functions and relations for the combined behaviour.

The first steps therefore in the extension of the experimental modelling tool will be to create a set of simple behaviours, a rule base for determining possible combinations and a set of methods for creating the combinations. A set of analysis requirements must be specified to determine the kind of reasoning which will be needed about the behaviour of the interconnected subsystems. This will determine how the rules and methods are to be accessed and therefore how they should be stored in the knowledge base. One possibility might be to create a behavioural simulation using qualitative behaviours for subsystems and setting up a rule base and set of methods which permit the qualitiative behaviour of interconnected subsystems to be inferred from their individual behaviours, i.e. a qualitative theory of feedback control [7].

8 Conclusion

In this report, a knowledge based approach to system modelling has been described. Modelling is the most time-consuming, complex and costly of control systems engineering tasks and effective tools to support the activity are required. During the modelling process, knowledge about the system is often gained incrementally and thus any tool must support this form of knowledge acquisition. Also the problem of complexity must be overcome by allowing the model to be created as a set of components with precisely specified interfaces and by providing operations to combine such components , and by implication, knowledge about their properties, e.g. behaviour and function. The approach proposed in this report envisages three knowledge bases concerned with structural, behavioural and functional knowledge respectively, with the ability to create relationships between them, e.g. system X has-behaviour Y. This allows these three categories of knowledge about a system to be incrementally developed (or enriched) independently, whilst maintaining the required relationships.

A simple experimental tool has been developed using the KEE knowledge engineering system. At present, this contains only knowledge about system structure. Further development is required to introduce behavioural knowledge, but some initial work is required here to develop a rule base (and possibly a set of methods) which will allow behaviours of interconnections of subsystems to be inferred from individual subsystem behaviour. The means of storing and accessing the rule base needs to be studied, together with the way in which the combined behaviour should be represented and, if required, stored in the knowledge base. A way of representing behaviour, based on specifying sets of sorts, functions and relations has been proposed, which is aimed at meeting the need to provide a general framework for representing many different kinds of knowledge about behaviour, both procedural and axiomatic, quantitative and qualitative, e.g. theories about stable behaviour.

References

1 Denham, M J , "Design issues for CACSD systems", Proc IEEE, December 1984

2 Barwise, J and Perry, J , "Situations and Attitudes", MIT Press, 1983

3 Goguen, J , Unpublished report in the newsletter of the Center for the Study of Language and Information, Stanford, 1986

4 Yeh, K , "Constructing and analyzing specifications of real world systems", Rep STAN-CS-86-1090, Department of Computer Science, Stanford University, 1986

5 Winograd, T , "Aleph, a system specification language" ,Stanford technical report, in preparation.

6 Minsky, M , "A framework for representing knowledge", in Winston (ed), The Psychology of Computer Vision, McGraw-Hill, 1975

7 Kuipers, B , "Commonsense reasoning about causality; deriving behaviour from structure", Journal of Artificial Intelligence, vol.24,1984.

SYSTEMS --------- SUBSYSTEMS

CONNECTORS

TERMINALS

SETUP.TERMINALS

SETUP.CONNECTORS

Figure 1. The structural knowledge base

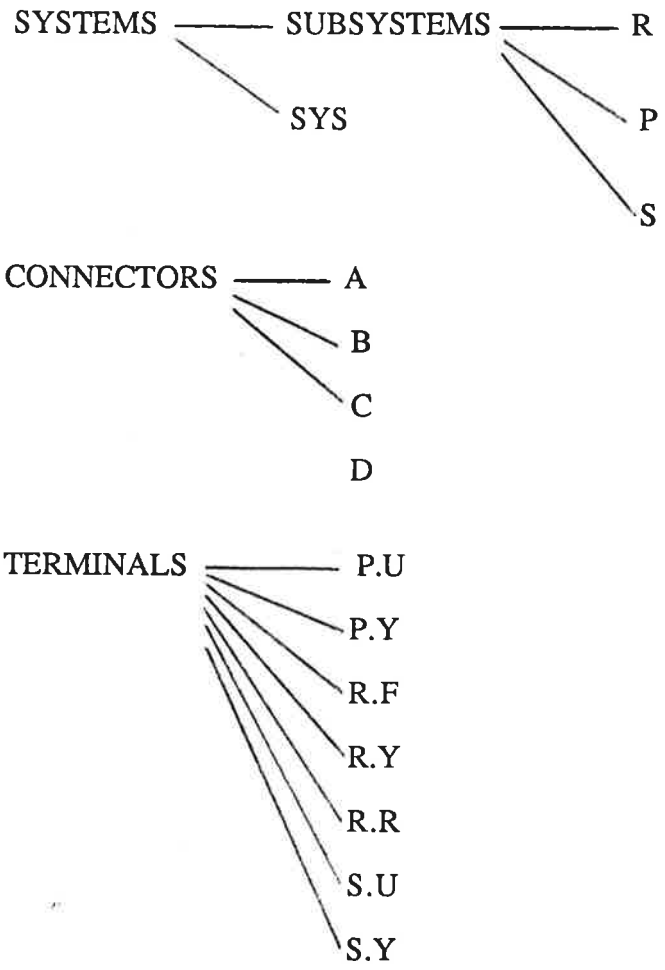

Figure 2. The structural knowledge base for the system of Figure 3.

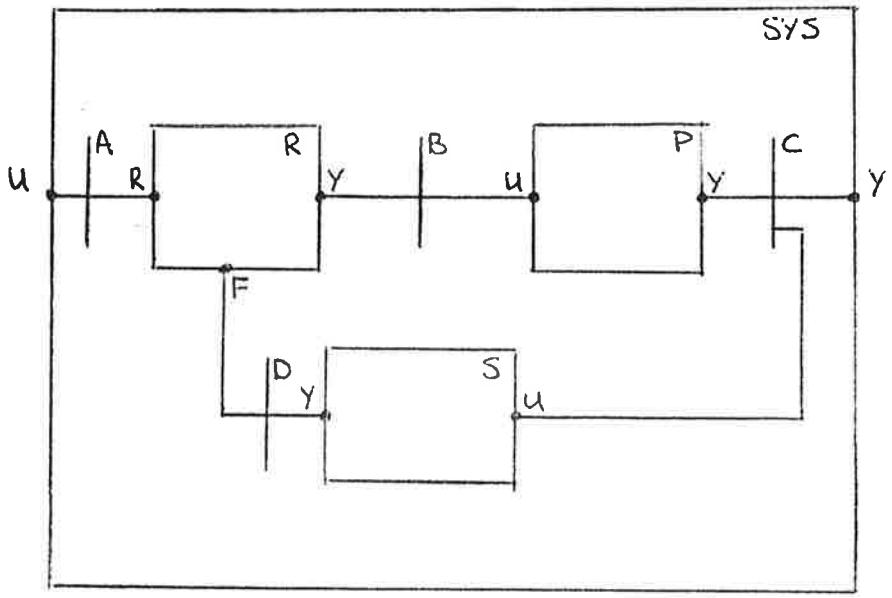

Figure 3. The example system

```
;;;  -*- Mode:LISP; Package:KEE; Base:10. -*-

(STRUCTURE
  ("" "29-Nov-1986 11:38:52" "MIKE" "5-Dec-1986 11:26:27")
  NIL
  (KNOWLEDGEBASES)
  NIL
  ()
  ((KBSIZE 23)
   (KEE.DEVELOPMENT.VERSION.NUMBER 0)
   (KEE.MAJOR.VERSION.NUMBER 2)
   (KEE.MINOR.VERSION.NUMBER 1)
   (KEE.PATCH.VERSION.NUMBER 64.1)
   (KEEVERSION KEE2.1)))


(CONNECTORS
  ("" "29-Nov-1986 11:44:55" "" "29-Nov-1986 11:45:43")
  ((ENTITIES GENERICUNITS))
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTIONS NIL NIL NIL NIL NIL)
   (PART-OF NIL NIL NIL NIL NIL))
  ())


(P
  ("" "29-Nov-1986 11:44:24" "MIKE" "5-Dec-1986 11:12:38")
  (SUBSYSTEMS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTORS (NONE))
   (PARAMETER-VARIABLES (NONE))
   (PART-OF (SYS))
   (SUBSYSTEMS (NONE))
   (TERMINAL-VARIABLES (U Y)))
  ())


(#:P.U
  ("MIKE" "5-Dec-1986 11:11:39" "MIKE" "5-Dec-1986 11:12:11")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (CURRENT))
   (PART-OF (#Unit (P STRUCTURE)))
   (RANGE ((0 10)))
   (TYPE (INPUT)))
  ())


(#:P.Y
  ("MIKE" "5-Dec-1986 11:12:38" "MIKE" "5-Dec-1986 11:13:07")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (TEMP))
   (PART-OF (#Unit (P STRUCTURE)))
   (RANGE ((0 100)))
   (TYPE (OUTPUT)))
```

```
  ())


(R
  ("" "29-Nov-1986 11:44:24" "MIKE" "5-Dec-1986 11:15:21")
  (SUBSYSTEMS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTORS (NONE))
   (PARAMETER-VARIABLES (NONE))
   (PART-OF (SYS))
   (SUBSYSTEMS (NONE))
   (TERMINAL-VARIABLES (R F Y)))
  ())


(#:R.F
  ("MIKE" "5-Dec-1986 11:14:49" "MIKE" "5-Dec-1986 11:15:05")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (VOLTAGE))
   (PART-OF (#Unit (R STRUCTURE)))
   (RANGE ((0 1)))
   (TYPE (INPUT)))
  ())


(#:R.R
  ("MIKE" "5-Dec-1986 11:13:50" "MIKE" "5-Dec-1986 11:14:28")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (VOLTAGE))
   (PART-OF (#Unit (R STRUCTURE)))
   (RANGE ((0 1)))
   (TYPE (INPUT)))
  ())


(#:R.Y
  ("MIKE" "5-Dec-1986 11:15:21" "MIKE" "5-Dec-1986 11:15:34")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (CURRENT))
   (PART-OF (#Unit (R STRUCTURE)))
   (RANGE ((0 10)))
   (TYPE (OUTPUT)))
  ())


(S
  ("" "29-Nov-1986 11:44:24" "MIKE" "5-Dec-1986 11:16:44")
  (SUBSYSTEMS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTORS (NONE))
   (PARAMETER-VARIABLES (NONE))
   (PART-OF (SYS))
```

```
    (SUBSYSTEMS (NONE))
    (TERMINAL-VARIABLES (U Y)))
   ())


(#:S.U
  ("MIKE" "5-Dec-1986 11:16:18" "MIKE" "5-Dec-1986 11:16:34")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (TEMP))
   (PART-OF (#Unit (S STRUCTURE)))
   (RANGE ((0 100)))
   (TYPE (INPUT)))
  ())


(#:S.Y
  ("MIKE" "5-Dec-1986 11:16:44" "MIKE" "5-Dec-1986 11:17:02")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (VOLTAGE))
   (PART-OF (#Unit (S STRUCTURE)))
   (RANGE ((0 1)))
   (TYPE (OUTPUT)))
  ())


(SETUP.CONNECTOR
  ("MIKE" "5-Dec-1986 10:32:58" "MIKE" "5-Dec-1986 11:00:39")
  ((ENTITIES GENERICUNITS))
  ((ACTIVEVALUE ACTIVEVALUES) (CLASSES GENERICUNITS))
  NIL
  ()
  ((AVPUT (LAMBDA (SELF SLOT NEWVALUE OLDVALUE UNIT SLOTTYPE)
            (COND ((> (LENGTH NEWVALUE) (LENGTH OLDVALUE))
                  (LET ((CNAME (MAKE-SYMBOL (STRING-APPEND (UNIT.NAME UNIT)
                                                           "."
                                                           (CAR (LAST NEWVALUE))))))
                    (UNITCREATE CNAME 'CONNECTORS NIL NIL NIL NIL)
                    (ADD.VALUE CNAME 'PART-OF UNIT)
                    (PRINC "Enter simple connector (a list of terminals): ")
                    (ADD.VALUE CNAME 'CONNECTIONS (READ)))))
            NEWVALUE)
          NIL
          NIL
          NIL
          ((FIRE.ON.KBLOAD NIL . UNIQUE)))))


(SETUP.TERMINALS
  ("MIKE" "3-Dec-1986 16:10:14" "MIKE" "5-Dec-1986 10:26:16")
  NIL
  ((ACTIVEVALUE ACTIVEVALUES))
  NIL
  ()
  ((AVPUT (LAMBDA (SELF SLOT NEWVALUE OLDVALUE UNIT SLOTTYPE)
            (COND ((> (LENGTH NEWVALUE) (LENGTH OLDVALUE))
                  (LET ((TNAME (MAKE-SYMBOL (STRING-APPEND (UNIT.NAME UNIT)
```

```
                                                              "."
                                                              (CAR (LAST NEWVALUE))))))
                        (UNITCREATE TNAME 'TERMINALS NIL NIL NIL NIL)
                        (ADD.VALUE TNAME 'PART-OF UNIT)
                        (PRINC "Terminal type (input,output,2-way): ")
                        (ADD.VALUE TNAME 'TYPE (READ))
                        (PRINC "Terminal description (any symbol): ")
                        (ADD.VALUE TNAME 'DESCRIPTION (READ))
                        (PRINC "Value range (lo high): ")
                        (ADD.VALUE TNAME 'RANGE (LIST (READ) (READ))))))
              NEWVALUE)
            NIL
            NIL
            NIL
            ((FIRE.ON.KBLOAD NIL . UNIQUE)))))


(SUBSYSTEMS
  ("" "29-Nov-1986 11:42:44" "" "29-Nov-1986 11:51:06")
  (SYSTEMS)
  ((CLASSES GENERICUNITS))
  NIL
  ((PART-OF NIL NIL NIL NIL NIL))
  ())


(SYS
  ("" "29-Nov-1986 11:43:20" "MIKE" "5-Dec-1986 11:23:47")
  (SYSTEMS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTORS (A B C D) NIL NIL NIL ((AVUNITS (#Unit (SETUP.CONNECTOR STRUCTURE)))))
   (PARAMETER-VARIABLES (NONE))
   (SUBSYSTEMS (R P S))
   (TERMINAL-VARIABLES (U Y)))
  ())


(#:SYS.A
  ("MIKE" "5-Dec-1986 11:21:39" "MIKE" "5-Dec-1986 11:21:59")
  (CONNECTORS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTIONS ((SYS.U R.R)))
   (PART-OF (#Unit (SYS STRUCTURE))))
  ())


(#:SYS.B
  ("MIKE" "5-Dec-1986 11:22:38" "MIKE" "5-Dec-1986 11:22:52")
  (CONNECTORS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTIONS ((R.Y P.U)))
   (PART-OF (#Unit (SYS STRUCTURE))))
  ())


(#:SYS.C
  ("MIKE" "5-Dec-1986 11:23:05" "MIKE" "5-Dec-1986 11:23:28")
```

```
  (CONNECTORS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTIONS ((SYS.Y P.Y S.U)))
   (PART-OF (#Unit (SYS STRUCTURE))))
  ())


(#:SYS.D
  ("MIKE" "5-Dec-1986 11:23:47" "MIKE" "5-Dec-1986 11:24:03")
  (CONNECTORS)
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTIONS ((S.Y R.F)))
   (PART-OF (#Unit (SYS STRUCTURE))))
  ())


(#:SYS.U
  ("MIKE" "5-Dec-1986 11:17:43" "MIKE" "5-Dec-1986 11:18:03")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (VOLTAGE))
   (PART-OF (#Unit (SYS STRUCTURE)))
   (RANGE ((0 1)))
   (TYPE (INPUT)))
  ())


(#:SYS.Y
  ("MIKE" "5-Dec-1986 11:20:36" "MIKE" "5-Dec-1986 11:20:52")
  (TERMINALS)
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION (TEMP))
   (PART-OF (#Unit (SYS STRUCTURE)))
   (RANGE ((0 100)))
   (TYPE (OUTPUT)))
  ())


(SYSTEMS
  ("" "29-Nov-1986 11:39:15" "MIKE" "3-Dec-1986 16:32:12")
  ((ENTITIES GENERICUNITS))
  ((CLASSES GENERICUNITS))
  NIL
  ((CONNECTORS NIL NIL NIL NIL NIL)
   (PARAMETER-VARIABLES NIL NIL NIL NIL NIL)
   (SHOW.ALL.CONNECTORS (LAMBDA (SELF)
                          (PRINT (UNIT.NAME SELF))
                          (PRINC ":::")
                          (LET ((X (GET.VALUES SELF 'SUBSYSTEMS)))
                            (PRINT (GET.VALUES SELF 'CONNECTORS))
                            (COND ((NOT (EQUAL '(NONE) X))
                                   (DO Y X (CDR Y) (NULL Y)
                                     (UNITMSG (CAR Y) 'SHOW.ALL.CONNECTORS))))))
                        METHOD
                        (METHOD))
   (SHOW.ALL.SUBSYSTEMS (LAMBDA (SELF)
```

```
                              (PRINT (UNIT.NAME SELF))
                              (PRINC ':')
                              (LET ((X (GET.VALUES SELF 'SUBSYSTEMS)))
                                (COND ((EQUAL '(NONE) X)
                                       (PRINT X)
                                       (TERPRI))
                                      (T
                                       (PRINT X)
                                       (TERPRI)
                                       (DO Y X (CDR Y) (NULL Y)
                                         (UNITMSG (CAR Y) 'SHOW.ALL.SUBSYSTEMS))))))
                            METHOD
                            (METHOD))
   (SUBSYSTEMS NIL NIL NIL NIL NIL)
   (TERMINAL-VARIABLES NIL NIL NIL NIL ((AVUNITS (#Unit (SETUP.TERMINALS STRUCTURE))))))
  ())


(TERMINALS
  ("MIKE" "3-Dec-1986 15:51:42" "MIKE" "3-Dec-1986 16:02:51")
  ((ENTITIES GENERICUNITS))
  ((CLASSES GENERICUNITS))
  NIL
  ((DESCRIPTION NIL NIL NIL NIL NIL)
   (PART-OF NIL NIL ((SUBCLASS.OF SYSTEMS)))
   (RANGE NIL NIL NIL NIL NIL)
   (TYPE NIL NIL ((ONE.OF INPUT OUTPUT 2-WAY)) NIL NIL))
  ())


KBEnd
```