Layered Design of KBS from Specification to Hardware

Dave Robertson, Nam Seog Park
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh

Jaume Agusti
Institut d'Investigacio en
Intelligencia Artificial
Cami de Santa Barbara, Blanes
Spain

Abstract

Knowledge based systems are now being embedded within the hardware of household items, such as cameras and washing machines. These systems demand a high degree of reliability, both in terms of the components used in implementation and in ensuring that the initial specification of the system is accurately implemented. We examine this problem in the context of a standard KBS specification language: first order predicate calculus (FOPC). Two key problems emerge. The first is to provide a reliable bridge between the axioms of FOPC and the design required for hardware implementation. If our FOPC specification can be translated to an equivalent propositional axiom set then we can provide an automatic transtation to existing design languages (*e.g.* functional logics). However, most KBS specifications cannot be simplified in this way. By drawing on work from a hitherto unrelated area we demonstrate, by example, how a significant class of non-propositional FOPC axioms can be translated automatically to an implementation–level design language. The second problem is in keeping track of the gradual process of refinement necessary when moving from an initial, underspecified description to the restricted class of FOPC appropriate for automatic translation to an implementation design. We describe a layered system of formal specification which permits incremental specification of the design and enables designers to converge on the appropriate class of FOPC axioms.

1 Why KBS Design for Hardware is Different from Traditional KBS Design

Traditional KBS design normally involves the definition of two separate pieces of software: an interpreter for performing the search necessary to solve problems, and a set of axioms (expressed in some convenient language) which supply the interpreter with appropriate information. It is rare to find this type of architecture used for hardware implementations, for a variety of reasons:

- The implementation must be compact and trimmed down to its essential components so that it can be tested more easily. Ideally, each component should have a simple local behaviour which can easily be tested and the connections between components should be explicit (and therefore easy to check for damage in operation). Notice that the use of a traditional interpreter and knowledge base does not provide this because the connections between axioms are made dynamically at run time.

- Often, the system must operate in real time, where responses must be guaranteed within fractions of a second. This places restrictions on the types of knowledge and inference mechanisms we can use, since we are disallowing infinite paths in the search space.

- Given the constraints above, it is preferable to implement as much as possible of the system as hardware - using a small range of components with simple behaviours.

- However, we must also retain the ability to represent the knowledge used by the system in a form which can be understood easily; which is modular and which behaves predictably at run time.

$$
\begin{array}{l}
p(X)\ \&\ q(X,Y) \rightarrow r(Y) \\
r(X)\ \&\ s(X) \rightarrow t(Y) \\
p(a)\ \&\ q(a,b)\ \&\ q(a,c)\ \&\ s(b)\ \&\ s(c)
\end{array}
$$

Figure 1: First Order Predicate Calculus axioms

$$
\begin{array}{l}
p(a)\ \&\ q(a,b) \rightarrow r(b) \\
p(a)\ \&\ q(a,c) \rightarrow r(c) \\
r(b)\ \&\ s(b) \rightarrow t(b) \\
r(c)\ \&\ s(c) \rightarrow t(c) \\
p(a)\ \&\ q(a,b)\ \&\ q(a,c)\ \&\ s(b)\ \&\ s(c)
\end{array}
$$

Figure 2: Ground version of Figure 1

A common manner of satisfying these requirements is to employ a propositional logic to represent our knowledge and then to convert the propositional axioms into a network of appropriate logic gates. For example, if we have the FOPC axiom set shown in Figure 1 then we can "flatten" this to the propositional axiom set shown in Figure 2.

From the propositional axioms of Figure 2 we can then construct the functional logic diagrams[1] shown in Figure 3. The arcs in these diagrams can be thought of as "wires" linking components of the network. The '&' connectives operate by producing an output signal if both input signals are active, othewise they produce no output. It is easy to see that if we activate the nodes representing $p(a)$, $q(a,b)$ and $s(b)$ then we will eventually activate the node for $t(b)$. This resembles the type of forward chaining inference which we might have obtained by applying an appropriate interpreter to the original FOPC axioms from Figure 2.

The example above demonstrates the essential features required of a hardware design: there are only two types of component (& connectives and propositional nodes) with simple local behaviours; the connections between components are explicit (via the arcs in the diagram); inference is potentially fast and the networks contain no infinite paths, so propagation of activation will terminate. Despite

[1]Functional logic diagrams are commonly used in the specification of safety-critical circuits (*e.g.* for automatic shutdown systems on oil production platforms). In Figure 3 we show a simplified version. In full functional logic diagrams it is possible to include other components such as invertors and time delays.

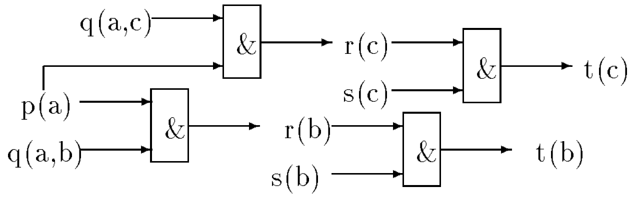

Figure 3: Representation of propositional axioms from Figure 2 using functional logic diagrams

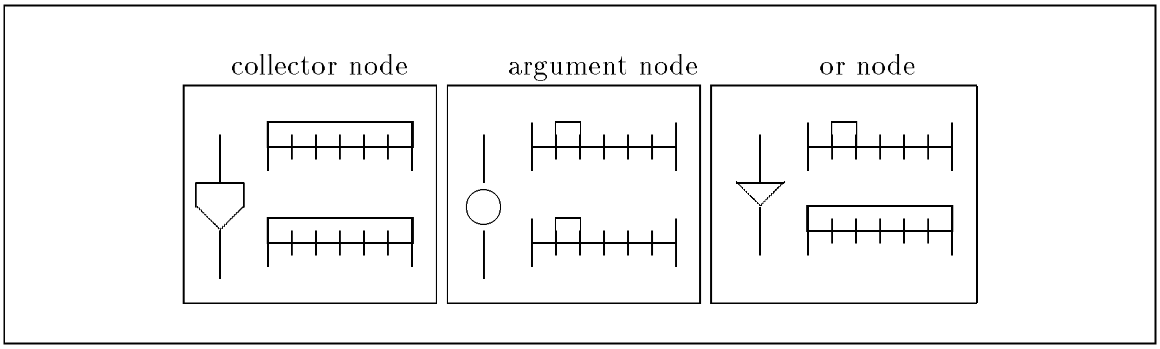

Figure 4: Node types in a Shastri-style network

these advantages, the functional logic is not ideal for representing FOPC because it provides no means of representing variable bindings explicitly. The functional logic is, effectively, a record of the entire search space for any potential query of the FOPC axioms, with a separate propositional node required for each possible variable binding. This not only increases the complexity of the network but also obscures the relationship between components of the network and the FOPC axioms from which it was derived.

Shastri & Ajjanagadde ([4],[5]) have described a form of network representation which avoids some of the problems of the propositional approach described above. The original motivation for their work was to provide a biologically plausible model for some types of reflexive reasoning. We do not consider here (nor do we necessarily subscribe to) such notions of biological plausibility. Our interest is only in the technical details of the notation. To simplify our explanation we use a simplified version of the full system in the discussion which follows, although extended versions now exist (see for example [3]). Essentially, we shall represent first order predicate calculus formulae as networks of three types of nodes, with spreading activation between the nodes providing an equivalent of either forward or backward chaining (depending on the initial network configuration). The identity of constants in the network is represented by allocating each to a separate phase of an oscillatory cycle which is propagated through the network according to the temporal behaviour of each node. In Figure 4 we show the temporal behaviour of the types of node used in this paper. The scales to the right of each node show the division into phases of input an output oscillatory cycles. Collector nodes are used to signal the selection of appropriate predicate names during inference. They are activated by input across all phases and produce as output in the next cycle a signal across all phases. Argument nodes are used to keep track of the identity of variable bindings. These nodes receive input in a particular phase and produce output on the same phase on the next cycle. “Or” nodes are used to ensure that the same variable binding is applied to multiple occurrences of a variable in the precondition of a rule. These receive input in a particular phase and produce output across all phases in the next cycle.

Applying the style of network described above to the rules from Figure 1 gives the network shown in Figure 5. We have created collector nodes for each of the predicate names (p, q, r, s, t) and argument nodes for the each predicate's arguments. For example, $q(X, Y)$ is represented by the collector node, q, and argument nodes $a2$ and $a3$. Having created the nodes for each predicate name and its arguments, we then “wire” together those nodes which interact. For example, the rule requires the node for predicate name r to become active if the nodes for p and q are active and if the identity of the first argument of q is the same as the argument of p. To ensure this, we connect p and q to r and also connect $a1$ and $a2$ to r, via an '*or*' node which will produce an output across the entire bandwidth if it receives two inputs in the same phase. We also require that the argument of r should be the same as the second argument of q so we connect $a3$ to $a4$. The propagation of variable bindings and predicate

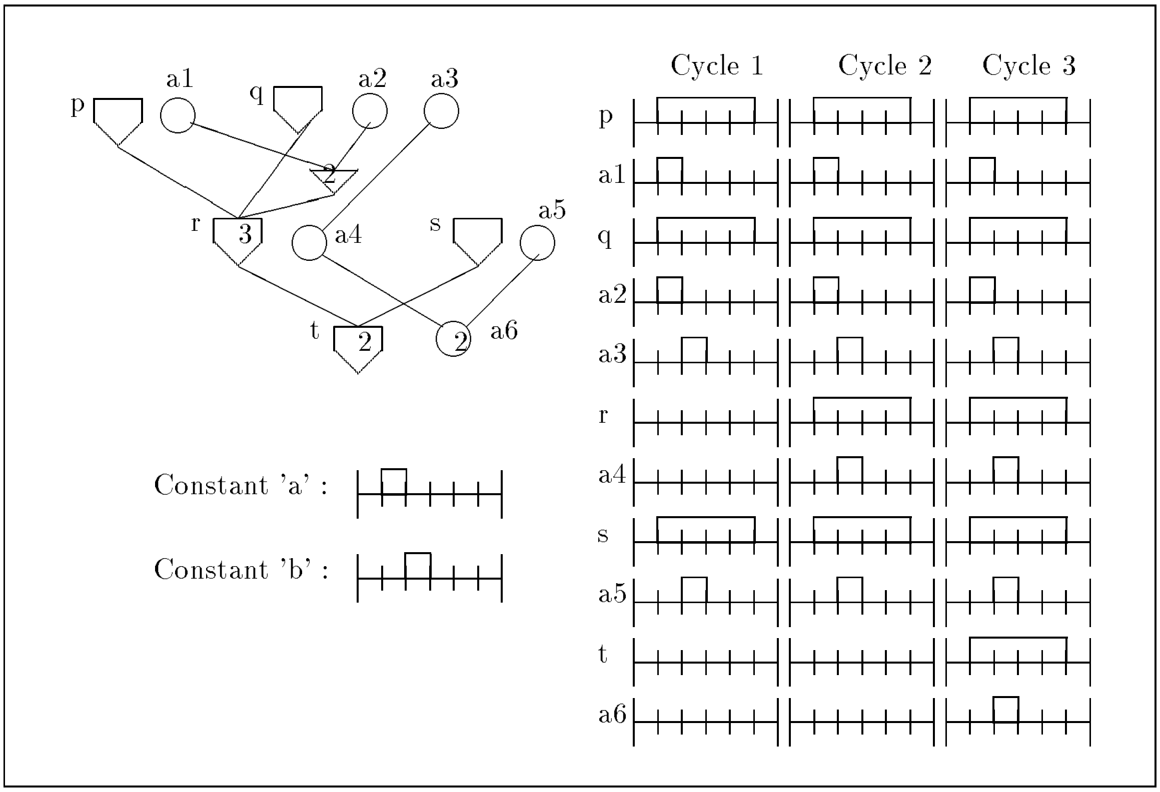

Figure 5: Encoding of the rules from Figure 1 in a Shastri-style network

activation during inference is illustrated in the sequence of cycles shown on the right in Figure 5. Assume, in this simplified example, that each cycle is divided into four phases with the first phase being used to represent the constant a and the second phase representing the constant b. We denote the facts $p(a)$, $q(a,b)$ and $s(b)$ by activating in the first cycle nodes p,q and s across all four phases; activating nodes $a1$ and $a2$ in the first phase; and activating nodes $a3$ and $a5$ in the second phase. In the second cycle, node r becomes active because it has received input across the entire bandwidth from all three of its input channels. Node $a4$ is also activated in phase with node $a3$. In the third cycle, the activations from nodes r and s propogate to node t and the activations of nodes $a4$ and $a5$ (which must be in the same phase) propagate to node $a6$. Since we have used the second phase to denote the constant b, we can now conclude $t(b)$, based on the facts $p(a)$, $q(a,b)$ and $s(b)$ which we presented to the network at the start of our sequence of cycles.

The reader will notice that the network in Figure 5 is no more compact than the propositional network of Figure 3. However, if we were to add to our axioms from Figure 1 the facts $q(a,d)$ and $s(d)$ then this requires us to add a complete new propositional network, as shown in Figure 6. The same new facts can be accommodated into the Shastri–style diagram from Figure 5 with no alterations, provided that we can find free phases to allocate to the new constant, d.

Algorithms exist for translating a restricted subset of first-order predicate calculus (without negation and standard recursion) into forms of representation similar to those used by Shastri (see for example [2]). We can then obtain very fast responses to queries – with an answer being guaranteed within a time proportional to the maximum diameter of the network. This form of representation is therefore compact, fast, predictable and seems suited to implementation in hardware (although we are not aware of any existing hardware implementations). Its major inadequacy is that, although individual components are simple, they need to be combined in complex ways for any non–trivial knowledge

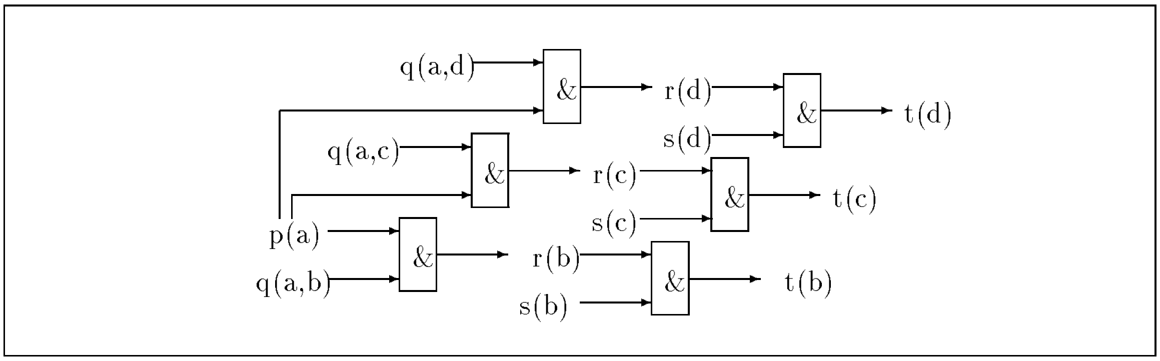

Figure 6: Network from Figure 2 accommodating the addition of $q(a, d)$ and $s(d)$

representation task. This means that it is difficult, simply by looking at a completed network, to distinguish how each part of it relates to the FOPC axioms from which it was derived. The purpose of this paper is to demonstrate how it is possible to provide an incremental description of this type of network, starting from general set–based descriptions; then to first order predicate calculus; and finally to hardware design. In Section 3 we introduce a simple boiler monitoring problem as a running example. As a precursor to this, we describe in the next section our view of layered specification in this context.

2 Layered Specification for Hardware Design

In attempting to make hardware and software more reliable, the vast majority of research effort has gone into verifying the behaviour of existing specifications. Recent analyses of system failures are now providing evidence that the majority of failures in some types of system may not be caused by verification problems but by misconceptions about what the system was designed to do. For example, in an analysis of 387 software errors discovered during testing of the Voyager and Galileo spacecraft ([1]), it is claimed that:

> "...the primary cause of safety-related functional faults is errors in recognising (understanding) the requirements."

Since these types of requirements errors could crop up anywhere from the first inception of the system to its finaal implementation, it is necessary to provide a framework for incremental development from general descriptions to implementation.

Our view of specification of knowledge based systems is that it can be divided into a number of distinct levels, with a fixed set of transformations relating higher level specifications to those below them. This view is quite conventional, being expressed informally in methodologies such as KADS ([7]) and more formally in languages such as $(ML)^2$ ([6]). The levels of specification described in following sections are shown diagrammatically in Figure 7. At the highest level are definitions of sorts and functions relevant to the system. We can then add more detail to these by defining interactions between functions. The language we have chosen for both these layers is set based because this helps ease the transition from sort definitions to function specification. At the next level, these are translated into equivalent expressions in FOPC. However, the restrictions we place on the implementation will only permit a subset of full FOPC so we have a further transition to this lowest level of specification before final translation to the hardware design language. All of these steps are controlled, implicitly, by the informal requirements of system designers.

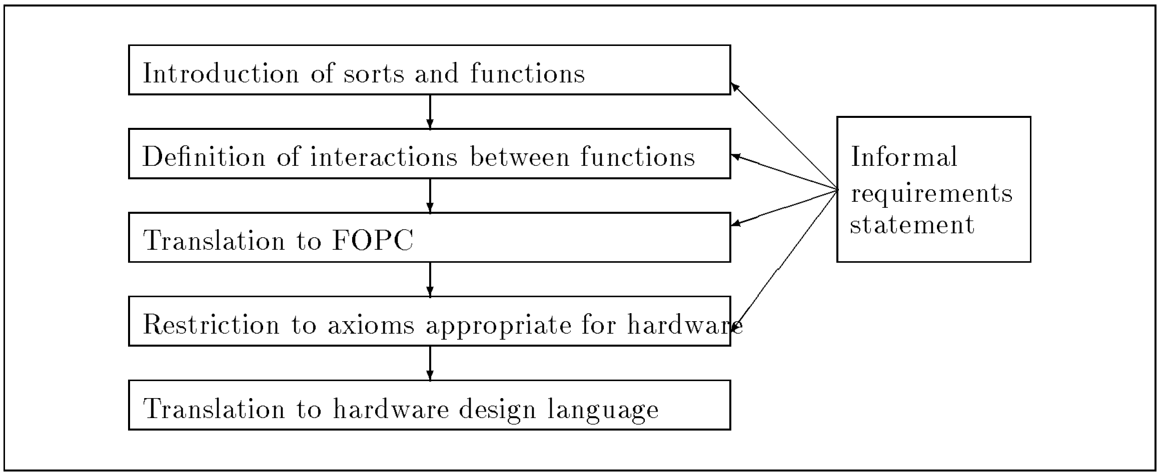

Figure 7: Overview of layers of design

3 Requirements Statement for a Simple KBS for Boiler Monitoring

The problem we have chosen is that of defining a monitoring unit[2] for a boiler system, as shown in Figure 8. Our boiler system consists of a number of boilers, each of which has pumped into it a water supply and releases as output a flow of steam. The rates of flow of water and steam are measured by meters and each boiler also has a meter to measure its content level. The readings from meters are converted from continuous numerical measures to discrete level assignments (*e.g.* on the scale low, medium, high) before being passed into the monitoring unit. Each pump is supplied with a monitoring device which indicates whether or not it is active. The monitoring unit should maintain a light on an operator console when the boiler is within safe operational levels. It should also switch on a warning light if a failure of meters or monitors occurs such that the system's estimate of metered levels is implausible.

This problem statement is underspecified – for example, we have said nothing about how safe levels or implausible metered levels are determined. In the following sections we shall work through the refinement of this initial specification, gaining detail on the way through, and ending up with an appropriate implementation using a Shastri–style network. We demonstrate how some simple interface tools can help to ease the description of the problem using formal languages.

4 Introduction of Sorts and Functions

We begin our description by defining the primitive sorts over which our initial functional descriptions will range. These are shown as arcs in the diagram below, in which the universal set is the symbol top. Within this are defined sets representing the components of the system (boilers, pumps, monitors); the types of measurement we might want to make (levels, rates) and the possible statuses of these components (confirmed, unknown, disconfirmed). All of the definitions of this section and of Section 5 will be expressed in terms of set inclusions. We use terms of the form $A \subseteq B$ to denote that some set, A, is included within the set B. For example, in the complete sort hierarchy of Figure 9 we have

[2] In some safety-critical systems it is necessary to have a system of multiple monitoring to ensure that failure in a single monitoring unit does not leave a fault undetected. We do not address this issue in the current paper but one could easily imagine replicating our solution for multiple monitoring units.

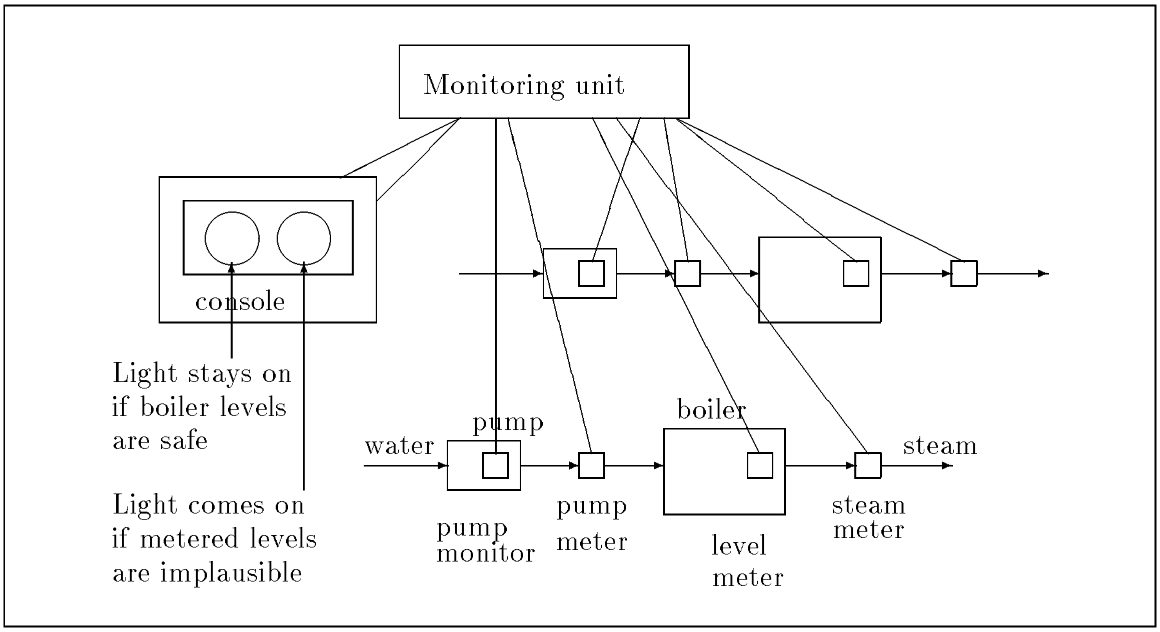

Figure 8: Schematic diagram for boiler system example

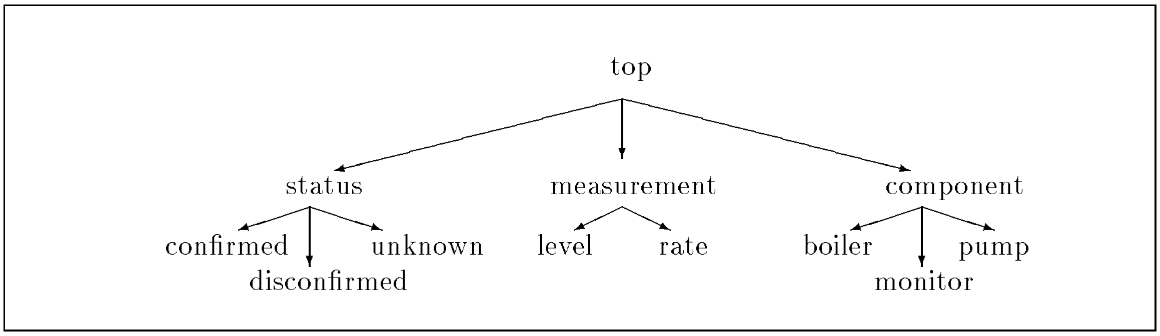

Figure 9: Basic sort hierarchy for the boiler example

an arc corresponding to the formula $disconfirmed \subseteq status$.

In our prototype system, implementd in LPA MacProlog, the construction of the basic sort lattice is performed using a standard menu–driven interface with edit fields to allow definition of new sorts. A complete set of these for our boiler example appears in Appendix A. Having defined the basic sorts in our monitoring system, we then introduce functions over these sorts. Initially, these are defined in a general way but in the next step we shall strengthen these preliminary definitions. Note that we are still using our inclusion operator, $\subseteq$, to relate these functions to the basic sorts which we had defined earlier.

5 Defining Interactions Between Functions

Our next step in system definition is to provide details of the interactions between the sets which we introduced in the previous section. To demonstrate how this is done, we use $safe\_level(boiler)$ as an example. Our aim is to refine the description of this function so that some subset of its results are defined over some subset of boilers. As a starting point for specifying these subsets, the user is provided with the template statement:

$$A \subseteq safe\_level(B) \leftarrow B \subseteq boiler$$

The system has introduced a new variable, B, to represent the subset of boilers and has introduced the variable, A, to represent the corresponding subset of safe levels. In our prototype implementation for manipulating these formulae, the screen display is divided into three areas:

- The top area gives the subset relation to be defined, in this case $A \subseteq safe\_level(B)$.
- The middle area is used to allocate any additional conditions to the definition (this would be empty of the expression above).
- The lower area is used to supply restrictions on all variables in the current formula. If new variables are introduced then extra edit fields are added appropriately, acting as a prompt for the user. In the current formula we have yet to supply a restriction on A but B is restricted to boilers.

If we wish to restrict A to be within both metered levels and safe operational levels then we add the restriction that $A \subseteq (metered\_level(B) \cap safe\_op\_level(B))$ in the edit field for the variable A. We may also wish to ensure that system consistency is always within the set of confirmed statuses. To do this, we add the restriction $consistency(B) \subseteq confirmed$ in the condition field. The resulting formula is:

$$\begin{aligned} &A \subseteq safe\_level(B) \leftarrow \\ &\qquad B \subseteq boiler \;\& \\ &\qquad A \subseteq (metered\_level(B) \cap safe\_op\_level(B)) \;\& \\ &\qquad consistency(B) \subseteq confirmed \end{aligned}$$

By a similar procedure we can create definitions of the other functions in our system. These are listed in Appendix B.

6 Translation to First Order Predicate Calculus

We now demonstrate how the formulae from the previous section can be translated into first order predicate calculus. However, we must bear in mind that we wish, ultimately, to be able to produce an implementation in Shastri & Ajjanagadde's notation. Since their notation permits only a limited subset of predicate calculus we must ensure that the formulae we produce at this stage fit into that subset. In particular, we wish to limit ourselves to Horn clauses (represented in the standard Prolog style) without negation. We demonstrate this translation by example.

Consider the formula for safe boiler levels, which we defined in the previous section:

$$\begin{aligned} &A \subseteq safe\_level(B) \leftarrow \\ &\qquad B \subseteq boiler \;\& \\ &\qquad A \subseteq (metered\_level(B) \cap safe\_op\_level(B)) \;\& \\ &\qquad consistency(B) \subseteq confirmed \end{aligned}$$

We first remove the intersection operator by replacing it with a conjunction, giving the formula:

$A \subseteq safe\_level(B) \leftarrow$
$\quad B \subseteq boiler\ \&$
$\quad A \subseteq metered\_level(B)\ \&$
$\quad A \subseteq safe\_op\_level(B)\ \&$
$\quad consistency(B) \subseteq confirmed$

We then translate every expression of the form $V \subseteq T$, where V is a variable and T is a non–variable, into a term T' which is identical to T but with the addition of V as an extra argument (this can become more complicated if T contains nested sets, as we shall demonstrate later). This gives us the formula:

$safe\_level(B, A) \leftarrow$
$\quad boiler(B)\ \&$
$\quad metered\_level(B, A)\ \&$
$\quad safe\_op\_level(B, A\ \&$
$\quad consistency(B) \subseteq confirmed$

There remains only to translate the term: $consistency(B) \subseteq confirmed$. The straightforward translation of this subset relation would be: $confirmed(X) \leftarrow consistency(B, X)$. Unfortunately, this translation would transgress our requirement for Horn clauses without negation. One way around this problem is to enumerate the elements of the set of confirmed statuses, $\{S_1, \cdots, S_n\}$, then add the conjunction of terms: $consistency(B, S1) \& \cdots \& consistency(B, Sn)$. In this case we assume that a confirmed status can be represented by a single constant, of the same name. This allows us to produce the final formula:

$safe\_level(B, A) \leftarrow$
$\quad boiler(B)\ \&$
$\quad metered\_level(B, A)\ \&$
$\quad safe\_op\_level(B, A)\ \&$
$\quad consistency(B, confirmed)$

Most of the other formulae from Appendix B can be translated in a similar way, with the exception of the definition for consistent pump operation:

$c\_pump(B) \subseteq confirmed \leftarrow$
$\quad B \subseteq boiler\ \&$
$\quad active(monitor(pump(B))) \subseteq confirmed$

The problem here is that we want to test for activity of all the monitors of all the pumps in the system. Given that we cannot tell (from the formula alone) how many pumps, monitors or boilers there are, we would like to use nested negation in order to ensure that the test for confirmation was performed over however many of each component there was. This would give us the translated formula:

$c\_pump(B, confirmed) \leftarrow$
$\quad boiler(B)\ \&$
$\quad \neg(pump(B, P)\ \&\ monitor(P, M)\ \&\ \neg active(M, confirmed))$

Unfortunately, this transgresses our requirement that negation should not appear in translated formulae. One way to avoid this is to enumerate sets of boilers, pumps and monitors. However, in our example we know that there is a single monitor on the pump and a single pump on the boiler so we can, in this case, simply dispense with the negation, giving the final translation as:

$$
\begin{aligned}
&c\_pump(B, confirmed) \leftarrow \\
&\qquad boiler(B)\ \& \\
&\qquad pump(B, P)\ \& \\
&\qquad monitor(P, M)\ \& \\
&\qquad active(M, confirmed)
\end{aligned}
$$

The rest of the translated formulae appear in Appendix C. The only additional transformation used in these is that unions between sets have been translated into disjunctions (represented using the operator 'or').

7 Translation to an Implementation Design

The FOPC axioms of the previous section are now in a form suitable for automatic translation into the network representation. Space limitations prevent us from drawing out the completed diagram, but the algorithm (and Prolog code) for generating such diagrams is described in detail in [2]. The only extra structure required in addition to those given in Figure 4 an inhibitory structure which ensures that a signal is inhibited on a link unless it conforms to the phase of a specified constant. The resulting diagram is sufficiently complex that it is difficult to imagine sizeable implementations being constructed directly at the network level. This underlines the desirability of specifying such systems in terms of more easily understandable notations. As we have demonstrated, these notations need not be identical throughout the specification. However, it is necessary that we can identify formally the connections between the different levels of specification and that the means by which we achieve these connections can readily be communicated to system designers. By integrating high and low level specification styles in this way, we retain the ability to identify the components of the network which correspond to the axioms of our high level specification. This makes it possible to identify the parts of the network which might be affected by changes to the high level specification and, equally importantly, it allows us to detect which parts of the high level specification might be invalidated by alterations to the network implementation.

8 Conclusions

KBS applications do not always require highly expressive implementation languages. We have described a class of applications for which we are prepared to sacrifice language features such as negation and general recursion in return for fast response times and finite search spaces. The problems faced by formal specification in these types of system are threefold:

- The notation required for implementation is too structurally complex to be practicable as a language for direct specification by users, since it is necessary to have pairwise connections between each all potentially unifying constants and variables. This makes higher level specification a necessity.
- The original problem description is often underspecified (see Section 3) with key details emerging in the process of refinement to the final design.
- As we proceed towards implementation it is necessary to constrain users in their expression of specification details, since requirements expressed at higher specification levels may only be realisable in limited forms because of the restrictions on the implementation language (We see this happening in Section 6).

We have described a number of techniques for tackling these problems:

- We can adopt more expressive implementation languages which are particularly suited to translation from commonly used specification languages. In Section 1 we introduced one example of this type of language. The advantage of using more expressive languages at this level is that we can afford to be less constrained during high level specification. However, this desire for expressiveness must always be tempered with an understanding of the constraints of the domain.
- The use of a layered approach (see Section 3) enables us to divide the task of specification into more manageable portions and imposes some order on the proceedings. This is not to imply that a strict ordering is always necessary – on the contrary, we found when developing the example in this paper that we frequently jumped between layers to work at greater or lesser detail as the need arose. The key benefit of a layered approach is to provide staging points when moving through the specification in this way.
- By using closely related formal languages for each layer it was possible to define comparatively straightforward translations between them. For example, the translation from our set–based language to full FOPC is automatic, although we impose additional restrictions on the class of allowed FOPC formulae for which we require some user interaction (see Section 6).

Our current prototype system for supporting this activity is rudimentary and fragmented. The set–based specification interface (shown in Sections 4 and 5) is able to produce appropriate FOPC translations but these currently have to be translated by hand into the restricted form needed by our automatic translator to the Shastri- style network. Forming this final connection is not, in theory, complicated. What remains unclear is the extent to which we can provide automated guidance for users in the articulation of high level specifications of this type. Part of this control may be provided by linking the levels of specification more closely, feeding back to higher specification decisions the implications for low level design. Other forms of control may be supplied by embedding within the system standard strategies for specification design. Formal frameworks, like the one in this paper, provide the skeleton upon which these more sophisticated systems may be attached.

References

[1] R.R. Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE Symposium on Requirements Engineering (RE93), IEEE Computer Society Press, ISBN 0-8186-3120-1*, pages 126–133, 1993.

[2] N.S. Park. A connectionist representation of rules, facts and dynamic bindings of variables. Technical Report TP-15, Department of Artificial Intelligence, University of Edinburgh, 1992.

[3] N.S. Park, D. Robertson, and K. Stenning. An extension of the temporal synchrony solution to dynamic variable bindings in a connectionist system. Research Report RP-666, Department of Artificial Intelligence, University of Edinburgh, 1993.

[4] L. Shastri and V. Ajjanagadde. Rules and variables in neural nets. *Neural Computation*, 3(1), 1991.

[5] L. Shastri and V. Ajjanagadde. Representation of rules, variables and dynamic bindings using temporal synchrony. *Behavioural and Brain Sciences*, 16(4), 1993.

[6] F. van Harmelen and J.R. Balder. (ml)2: A formal language for kads models of expertise. *Knowledge Acquisition*, 4(1), 1992.

[7] B.J. Wielinga, A.T. Schreiber, and J.A. Breuker. Kads: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–53, 1992. Reprinted in Buchanan, B and Wilkins, D. (eds) 1992, Readings in Knowledge Acquisition and Learning, pp 92-116, Morgan Kaufmann.

A Initial Sort Lattice for Boiler Example

Levels	
The level limits of boilers are within the set of levels.	$limit\_boiler\_level(boiler) \subseteq level$
The minimum levels of boilers are within the set of levels.	$minimum\_boiler\_level(boiler) \subseteq level$
Implausible metered levels of the boiler are within the set of levels.	$implausible\_metered\_level(boiler) \subseteq level$
Safe operational levels of the boiler are within the set of levels.	$safe\_op\_level(boiler) \subseteq level$
Metered boiler levels are within the set of levels.	$metered\_level(boiler) \subseteq level$
Safe boiler levels are within the set of levels.	$safe\_level(boiler) \subseteq level$

Rates	
The metered steam rates of boilers are within the set of rates.	$metered\_steam\_rate(boiler) \subseteq rate$
The limits of steam rates of boilers are within the set of rates.	$limit\_steam\_rate(boiler) \subseteq rate$

Tests	
Consistency tests of boiler systems are within the set of statuses.	$consistency(boiler) \subseteq status$
Consistency tests of the steam monitoring systems are within the set of statuses.	$c\_steam(boiler) \subseteq status$
Consistency tests of the pump monitors are within the set of statuses.	$c\_pump(boiler) \subseteq status$
Consistency tests of the boiler level monitoring system are within statuses.	$c\_level(boiler) \subseteq status$
Consistency tests between steam and pump readings are within the set of statuses.	$c\_steam\_and\_pump(boiler) \subseteq status$
Consistency tests between steam and boiler level readings are within the set of statuses.	$c\_steam\_and\_level(boiler) \subseteq status$
Consistency tests between pump and boiler level readings are within the set of statuses.	$c\_pump\_and\_level(boiler) \subseteq status$
Tests for the activity of monitors are within the set of statuses.	$active(monitor) \subseteq status$
Tests for the value of a measurement being below another are within the set of statuses.	$below(measurement, measurement) \subseteq status$
Tests for inactive pump monitors are within the set of statuses.	$inactive\_pump\_monitor(boiler) \subseteq status$
Tests for missing signs of activity of pump monitors are within the set of statuses.	$missing\_active\_pump\_monitor(boiler) \subseteq status$

B Function Interactions for Boiler Example

All results for consistency of B are confirmed if B is in the set of boilers and all of the consistency tests of components of the system give at least one result which is confirmed.

$$consistency(B) \subseteq confirmed \leftarrow$$
$$B \subseteq boiler \ \& $$
$$(c\_steam(B) \cap c\_pump(B) \cap c\_level(B) \cap c\_steam\_and\_pump(B) \cap$$
$$c\_steam\_and\_level(B) \cap c\_pump\_and\_level(B)) \subseteq confirmed$$

Unknown statuses are included in the reports for consistency of B if B is in the set of boilers and at least one of the consistency tests of components of the system gives a result which is of status unknown.

$$unknown \subseteq consistency(B) \leftarrow$$
$$B \subseteq boiler \ \&$$
$$(c\_steam(B) \cup c\_pump(B) \cup c\_level(B) \cup c\_steam\_and\_pump(B) \cup$$
$$c\_steam\_and\_level(B) \cup c\_pump\_and\_level(B)) \subseteq unknown$$

Disconfirmed statuses are in the consistency of B if B is in the set of boilers and at least one of the consistency tests of components of the system gives a result which is disconfirmed.

$$disconfirmed \subseteq consistency(B) \leftarrow$$
$$B \subseteq boiler \ \&$$
$$(c\_steam(B) \cup c\_pump(B) \cup c\_level(B) \cup c\_steam\_and\_pump(B) \cup$$
$$c\_steam\_and\_level(B) \cup c\_pump\_and\_level(B)) \subseteq disconfirmed$$

Consistent steam readings of B are confirmed if B is in the set of boilers and M is in the set of metered steam rates of B and L is in the set of limits to steam rates of B and all values in M are below those of L.

$$c\_steam(B) \subseteq confirmed \leftarrow$$
$$B \subseteq boiler \ \& \ M \subseteq metered\_steam\_rate(B) \ \& \ L \subseteq limit\_steam\_rate(B) \ \& \ below(M, L) \subseteq confirmed$$

Consistent steam readings of B are disconfirmed if B is in the set of boilers and M is in the set of metered steam rates of B and L is in the set of limits to steam rates of B and the tests for L below M include signals of confirmation.

$$c\_steam(B) \subseteq disconfirmed \leftarrow$$
$$B \subseteq boiler \ \& \ M \subseteq metered\_steam\_rate(B) \ \& \ L \subseteq limit\_steam\_rate(B) \ \& \ confirmed \subseteq below(L, M)$$

Consistent pump readings of B are confirmed if B is in the set of boilers and tests for all active pump monitors of B are confirmed.

$$c\_pump(B) \subseteq confirmed \leftarrow$$
$$B \subseteq boiler \ \& \ active(monitor(pump(B))) \subseteq confirmed$$

Consistent pump readings of B are disconfirmed if B is in the set of boilers and P is a subset of pumps of B and M is a subset of the monitors on P and the results of tests for activity on M are disconfirmed.

$$c\_pump(B) \subseteq disconfirmed \leftarrow$$
$$B \subseteq boiler \ \& \ P \subseteq pump(B) \ \& \ M \subseteq monitor(P) \ \& \ active(M) \subseteq disconfirmed$$

L is included in safe operational levels of B if B is in the set of boilers and M is in the set of metered levels of B and L is in the set of levels of B and all values in M are below those of L.

$$L \subseteq safe\_op\_level(B) \leftarrow$$
$$B \subseteq boiler \ \& \ M \subseteq metered\_level(B) \ \& \ L \subseteq limit\_boiler\_level(B) \ \& \ below(M, L) \subseteq confirmed$$

L is included in the set of implausible metered levels if B is in the set of boilers and M is a subset of metered levels of B and L is a subset of the limits of levels of B and all values in L are below those of M.

$$L \subseteq implausible\_metered\_level(B) \leftarrow$$
$$B \subseteq boiler \ \& \ M \subseteq metered\_level(B) \ \& \ L \subseteq limit\_boiler\_level(B) \& \ below(L, M) \subseteq confirmed$$

L is included in the set of implausible metered levels if B is in the set of boilers and M is a subset of metered levels of B and L is a subset of the minimum levels of B and L is a subset of the minimum levels of B and all values in M are below those of L.

$L \subseteq implausible\_metered\_level(B) \leftarrow$
$\quad B \subseteq boiler \;\&\; M \subseteq metered\_level(B) \;\&\; L \subseteq minimum\_boiler\_level(B) \;\&\; below(M, L) \subseteq confirmed$

M is a subset of pump monitors not known to be active if B is a subset o the set of boilers and P is a subset of pumps of B and M is a subset of monitors on P and the activity state of M is unknown.

$M \subseteq missing\_active\_pump\_monitor(B) \leftarrow$
$\quad B \subseteq boiler \;\&\; P \subseteq pump(B) \;\&\; M \subseteq monitor(P) \;\&\; active(M) \subseteq unknown$

M is a subset of inactive pump monitors if B is a subset of the set of boilers and P is a subset of pumps of B and M is a subset of monitors on P and the activity state of M is disconfirmed.

$M \subseteq inactive\_pump\_monitor(B) \leftarrow$
$\quad B \subseteq boiler \;\&\; P \subseteq pump(B) \;\&\; M \subseteq monitor(P) \;\&\; active(M) \subseteq disconfirmed$

C FOPC Clauses for the Boiler Example

The consistency of B is confirmed if B is a boiler and consistency of steam monitoring of B is confirmed and consistency of pump monitoring of B is confirmed and consistency of level monitoring of B is confirmed and consistency of steam and pump is confirmed and consistency of steam and level is confirmed and consistency of pump and level is confirmed.

$consistency(B, confirmed) \leftarrow$
$\quad boiler(B) \;\&$
$\quad c\_steam(B, confirmed) \;\&$
$\quad c\_pump(B, confirmed) \;\&$
$\quad c\_level(B, confirmed) \;\&$
$\quad c\_steam\_and\_pump(B, confirmed) \;\&$
$\quad c\_steam\_and\_level(B, confirmed) \;\&$
$\quad c\_pump\_and\_level(B, confirmed)$

The consistency of B is unknown if B is a boiler and consistency of steam monitoring of B is unknown or consistency of pump monitoring of B is unknown or consistency of level monitoring of B is unknown or consistency of steam and pump is unknown or consistency of steam and level is unknown or consistency of pump and level is unknown.

$consistency(B, unknown) \leftarrow$
$\quad boiler(B) \;\&$
$\quad (\; c\_steam(B, unknown) \; or$
$\quad\; c\_pump(B, unknown) \; or$
$\quad\; c\_level(B, unknown) \; or$
$\quad\; c\_steam\_and\_pump(B, unknown) \; or$
$\quad\; c\_steam\_and\_level(B, unknown) \; or$
$\quad\; c\_pump\_and\_level(B, unknown))$

The consistency of B is disconfirmed if B is a boiler and consistency of steam monitoring of B is disconfirmed or consistency of pump monitoring of B is disconfirmed or consistency of pump monitoring of B is disconfirmed or consistency of level monitoring of B is disconfirmed or consistency of steam and pump is disconfirmed or consistency of steam and level is disconfirmed or consistency of pump and level is disconfirmed.

$consistency(B, disconfirmed) \leftarrow$
$\quad boiler(B) \;\&$
$\quad (\; c\_steam(B, disconfirmed) \; or$
$\quad\; c\_pump(B, disconfirmed) \; or$
$\quad\; c\_level(B, disconfirmed) \; or$
$\quad\; c\_steam\_and\_pump(B, disconfirmed) \; or$
$\quad\; c\_steam\_and\_level(B, disconfirmed) \; or$
$\quad\; c\_pump\_and\_level(B, disconfirmed))$

Consistency of steam monitoring of B is confirmed if B is a boiler and the metered steam rate of B is M and the limit to steam rate of B is L and the test for M being below L is confirmed.

$c\_steam(B, confirmed) \leftarrow$
$$boiler(B) \ \& \ metered\_steam\_rate(B, M) \ \& \ limit\_steam\_rate(B, L) \ \& \ below(M, L, confirmed)$$

Consistency of steam monitoring of B is disconfirmed if B is a boiler and the metered steam rate of B is M and the limit to steam rate of B is L and the test for L being below M is confirmed.

$c\_steam(B, disconfirmed) \leftarrow$
$$boiler(B) \ \& \ metered\_steam\_rate(B, M) \ \& \ limit\_steam\_rate(B, L) \ \& \ below(L, M, confirmed)$$

Consistency of pump monitoring of B is disconfirmed if B is a boiler and P is the pump for B and M is the monitor on P and the test for activity of M is disconfirmed.

$c\_pump(B, disconfirmed) \leftarrow$
$$boiler(B) \ \& \ pump(B, P) \ \& \ monitor(P, M) \ \& \ active(M, disconfirmed)$$

M is a safe operating level for B if B is a boiler and M is a metered level of B and L is the limit to the level of B and the test for M being below L is confirmed.

$safe\_op\_level(B, M) \leftarrow$
$$boiler(B) \ \& \ metered\_level(B, M) \ \& \ limit\_boiler\_level(B, L) \ \& \ below(M, L, confirmed)$$

M is an implausible metered level for B if B is a boiler and M is a metered level of B and L is the limit to the level of B and the test for L being below M is confirmed.

$implausible\_metered\_level(B, M) \leftarrow$
$$boiler(B) \ \& \ metered\_level(B, M) \ \& \ limit\_boiler\_level(B, L) \ \& \ below(L, M, confirmed)$$

M is an implausible metered level for B if B is a boiler and M is a metered level of B and L is the minimum level of B and the test for M being below L is confirmed.

$implausible\_metered\_level(B, M) \leftarrow$
$$boiler(B) \ \& \ metered\_level(B, M) \ \& \ minimum\_boiler\_level(B, L) \ \& \ below(M, L, confirmed)$$

M is a pump monitor which is not known to be active if B is a boiler and P is a pump for B and M is a monitor on P and the test for activity of M is unknown.

$missing\_active\_pump\_monitor(B, M) \leftarrow$
$$boiler(B) \ \& \ pump(B, P) \ \& \ monitor(P, M) \ \& \ active(M, unknown)$$

M is a pump monitor which is known to be inactive if B is a boiler and P is a pump for B and M is a monitor on P and the test for activity of M is unknown.

$inactive\_pump\_monitor(B, M) \leftarrow$
$$boiler(B) \ \& \ pump(B, P) \ \& \ monitor(P, M) \ \& \ active(M, disconfirmed)$$