Lifecycle Calculus

Dave Robertson
Division of Informatics, University of Edinburgh
24th October 2000

Abstract

The notion of "lifecycle" is central to AKT. We envisage a high level of automation in our lifecycles, which means that there must at least be some basic, uniform level of formality to support automation. Getting this right is essential not only in providing the technical glue to link different techniques but also to provide a broader, theoretical framework within which we can understand methods for individual lifecycle stages. This note provides some preliminary ideas on a calculus for describing lifecycles.

AKT is founded on the idea that we can connect the diverse systems which hitherto were used disconnectedly in knowledge engineering and, by doing this, produce technologies, methods and theories which are more than the sum of their parts. This is ambitious because of the large number of of steps involved in a lifecycle; the size and complexity of the software involved at each step; and the high level of automation which we wish to introduce into the process. It is problematic because there is, as far as I am aware, little research into the ways we might understand this new sort of lifecycle. One way of beginning to understand is by describing formally the essentials of the activity.

A key issue is the level of description. Too shallow and we have no link to the technologies we wish to support. Too deep and we become swamped by the task of formally describing details of numerous individual techniques. A concept which may help resolve this tension is that of capabilities, which I interpret as a form of specification of a problem solver ("problem solver" her being used broadly to mean anything you care to write which could be used to compute the answer to some problem). Section 1 enlarges a little on this topic, sufficient to understand the rest of the note.

Section 3 defines a basic calculus for describing lifecycles. This has been written in ignorance of related work, since it seems to me to be a higher priority right now to establish a direction to this area of AKT activity rather than to conduct a meticulous survey. The calculus has not been considered in detail from a formal point of view so it will almost certainly change once I have given it a little more thought. Nevertheless, it seems in the spirit of AKT to share early ideas, rather than waiting for a polished result.

Section 5 describes how the calculus can be used to support a simple, useful form of inference, relevant to high level knowledge management, which would not be possible in practice without the calculus (or something similar to it).

Sections 2, 4 and 6 give a running example which demonstrates the calculus in use. I do not claim that the example demonstrates practicality. We remain a long way from that. It is, however, encouraging to be able to describe reasonably complex chains of construction with a comparatively simple formal system.

1 Capability Descriptions, Specifications and Problem Solvers

This note does not prescribe a particular style of capability description, except where concrete examples are needed for illustration. Nevertheless, to understand the demands on a calculus for KBS lifecycles we must have a view of what capabilities should be.

The purpose of a capability description is to describe the computation which a problem solver may do, without having to invoke the computation itself, so that we may decide whether to use it for a given task. This is similar to the use of specifications in programming, where the specification describes what we want the program to do and the program is designed to do it. There are, however, several ways of writing a traditional formal specification:

- Non-executable, total specifications describe precisely all that is required of the program but in a style which does not reveal how to compute the program. The specification $X = \sqrt{X^2}$ is a total specification of what it means to be the square root of a number in terms of the square of that number, but reveals little about the computation of square roots.
- Executable, total specifications are as above but are described in a style which allows direct computation. The specification, as a logic program, for appending two lists together to form a third: $append(A, B, C) \leftrightarrow ((A = [] \wedge B = C) \vee (A = [H|T] \wedge B = [H|R] \wedge append(T, B, R)))$ is computable as a logic program.
- Partial specifications describe constraints on the requirements for input or output of a program. The specification: $append(A, B, C) \leftarrow list(A) \wedge list(B)$ constrains the input types of *append* but does not constrain its output.

Although total specifications are an ideal, in the sense that they describe exactly what an appropriately constructed problem solver will do, it seems impractical to build total specifications for realistically sized problem solvers. Constructing such specifications is (arguably) more time consuming and error prone than programming. Partial specifications are much easier to write; may be more stable over the lifetime of a program (if we require only the partial specification to be preserved during program adaptation); and may be sufficient in practice to allow appropriate choices of problem solvers for given tasks. This last point is speculation, relying on ingenuity in being "just partial enough" to have small, easily written[1], capability descriptions while also providing sufficient information to discriminate between problem solvers.

2 Scenario: Part 1

We introduce, as a demonstration, a simple example in which the lifecycle of development is described. Since it is small we are able to show the problem solver and capability descriptions at each stage. This is for expository purposes only. In practice we may never look at the problem solvers when constructing lifecycle descriptions, so the problem solvers may be complex and in any language. In this example, for simplicity we use the same language (Horn clauses) for both capability and problem solver description.
Suppose that we have available, through some Web publishing mechanism, three problem solvers. We describe each below.
The first, call it s_e, is a collection of rules for classifying objects as safe or dangerous according to physical features such as colour and texture. The rules are as follows:

$$\begin{array}{rcl} hazard(X, yes) & \leftarrow & colour(X, red) \wedge texture(X, spiky) \\ hazard(X, yes) & \leftarrow & colour(X, red) \wedge texture(X, smooth) \\ hazard(X, yes) & \leftarrow & colour(X, blue) \wedge texture(X, spiky) \\ hazard(X, no) & \leftarrow & colour(X, blue) \wedge texture(X, smooth) \end{array}$$

The capability we assign to this problem solver is that if we supply it with an object colour and texture then it will (if it can) tell us whether the object is dangerous. We write this formally as the capability description (call it c_e):

[1] I assume an engineer would write a capability description from scratch. It would be interesting to see whether they could be synthesised in some way but that is not a pre-requisite for this note.

$$hazard(X,H) \leftarrow colour(X,C) \wedge texture(X,T)$$

The second problem solver, call it s_r, simply provides a single rule for assessment of safety, only judging things to be safe if they are not known to be hazardous and are known to be safe. Notice that we are uniformly assuming Prolog-style closed world negation so it makes sense to perform this check.

$$safe(X) \leftarrow hazard(X,no) \wedge not(hazard(X,yes))$$

The capability we assign to this problem solver is that it will judge safety if we tell it about a hazard. We write this formally as the capability description (call it c_r):

$$safe(X) \leftarrow hazard(X,Y)$$

The third problem solver, call it s_i, is a basic Prolog meta-interpreter, augmented to construct an explanation in its second argument.

$$\begin{array}{rcl} solve(true,true) & & \\ solve((A \wedge B), and(A_e,B_e)) & \leftarrow & solve(A,A_e) \wedge solve(B,B_e) \\ solve((A \vee B), A_e) & \leftarrow & solve(A,A_e) \\ solve((A \vee B), B_e) & \leftarrow & solve(B,B_e) \\ solve(A,B_e) & \leftarrow & clause(A,B) \wedge solve(B,B_e) \end{array}$$

The capability we assign to this problem solver is that it will solve a query given at least one definition for *clause*. We write this formally as the capability description (call it c_i):

$$solve(X,Y) \leftarrow clause(X,Z)$$

We now start to build a new problem solver using the sources s_e, s_r and s_i. First we choose s_e and apply a rule induction algorithm to generalise the examples. Since the capability description for s_e is already generalised this doesn't change it. The new version of the problem solver is:

$$\begin{array}{rcl} hazard(X,yes) & \leftarrow & colour(X,red) \\ hazard(X,yes) & \leftarrow & colour(X,blue) \wedge texture(X,spiky) \\ hazard(X,no) & \leftarrow & colour(X,blue) \wedge texture(X,smooth) \end{array}$$

We then combine this problem solver with the safety decision rule from s_r. This gives us the combined problem solver:

$$\begin{array}{rcl} hazard(X,yes) & \leftarrow & colour(X,red) \\ hazard(X,yes) & \leftarrow & colour(X,blue) \wedge texture(X,spiky) \\ hazard(X,no) & \leftarrow & colour(X,blue) \wedge texture(X,smooth) \\ & & \\ safe(X) & \leftarrow & hazard(X,no) \wedge not(hazard(X,yes)) \end{array}$$

and the capabilities of this problem solver is the union of the capabilities c_e and c_r:

$$\begin{array}{rcl} hazard(X,H) & \leftarrow & colour(X,C) \wedge texture(X,T) \\ safe(X) & \leftarrow & hazard(X,Y) \end{array}$$

We now wish to build a meta-interpreter which will work with the problem solver we currently possess but adding to it an explanation facility. To do this we choose the Prolog meta-interpreter at s_i and restrict it (since we don't have to deal with disjunction); then change the *clause* predicate to be the '$\leftarrow$' operator to suit the ontology we are using. This gives us the problem solver:

$$\begin{array}{rcl} solve(true, true) & & \\ solve((A \wedge B), and(A_e, B_e)) & \leftarrow & solve(A, A_e) \wedge solve(B, B_e) \\ solve(A, B_e) & \leftarrow & A \leftarrow B \wedge solve(B, B_e) \end{array}$$

which has the same capabilities as the original, given the ontological change:

$$solve(X, Y) \quad \leftarrow \quad (X \leftarrow Z)$$

Finally, we combine the new interpreter derived from s_i with the problem solving knowledge from s_e and s_r:

$$\begin{array}{rcl} hazard(X, yes) & \leftarrow & colour(X, red) \\ hazard(X, yes) & \leftarrow & colour(X, blue) \wedge texture(X, spiky) \\ hazard(X, no) & \leftarrow & colour(X, blue) \wedge texture(X, smooth) \\ & & \\ safe(X) & \leftarrow & hazard(X, no) \wedge not(hazard(X, yes)) \\ & & \\ solve(true, true) & & \\ solve((A \wedge B), and(A_e, B_e)) & \leftarrow & solve(A, A_e) \wedge solve(B, B_e) \\ solve(A, B_e) & \leftarrow & A \leftarrow B \wedge solve(B, B_e) \end{array}$$

The capabilities of this problem solver include those of the earlier problem solvers and we might even want to add a more specific capability relating the explanation-giving solver to the attributes of the example rules.

$$\begin{array}{rcl} hazard(X, H) & \leftarrow & colour(X, C) \wedge texture(X, T) \\ safe(X) & \leftarrow & hazard(X, Y) \\ solve(X, Y) & \leftarrow & (X \leftarrow Z) \\ solve(safe(X), Y) & \leftarrow & colour(X, C) \wedge texture(X, T) \end{array}$$

This completes our example. We return to it in Sections 4and 6 but first we introduce a basic calculus for describing this sort of process.

3 Basic Calculus

This section introduces a minimal calculus for describing the sort of lifecycle which we introduced in the previous section. If such a calculus were to be used in practice then much more thought would need to be given to reconciling the steps described by our formal rules with the steps which knowledge engineers actually find convenient. This could be done in a number of ways: by inventing different rules in a similar style; by extending the existing rules; or by defining additional rules as refinements of the basic ones. The aim of this note is not to perform this reconciliation but to describe the essential problem which must be tackled. Surprisingly, even the minimal calculus has its uses, as we demonstrate by example in Section 6.

We shall describe the calculus as a system of eight rules, given as expressions 1 to 8 below. Each of these is of the traditional form $\frac{P}{C}$ where C is a conclusion we are allowed to draw given the preconditions P. Each of the rules provides a transition from one lifecycle stage to another. These are described informally below.

1. Any capability which is available in the initial pool of published capability descriptions can be acquired as the initial step in a lifecycle.
2. If we have two lifecycles leading to capability descriptions which are consistent with each other then we can form a lifecycle leading to the union of these capability descriptions.
3. If we have two lifecycles leading to non-empty intersections of capabilities then we can form a lifecycle to the intersection of those capabilities.
4. If we can find a generalisation of the capabilities resulting from any lifecycle then we can add those generalised capabilities as the next step in that lifecycle.
5. If we can find a specialisation of the capabilities resulting from any lifecycle then we can add those specialised capabilities as the next step in that lifecycle.
6. If we can change the ontology used at a given stage of the lifecycle into a new ontology formed strictly by changing the names of symbols in the old ontology then we can use the new ontology as the basis for the next step in the lifecycle.
7. The capability description produced at the end of any lifecycle can be published (along with the appropriate record of its lifecycle and the ontology it uses).
8. A capability description published as a consequence of one lifecycle may be used as the basis for another lifecycle, yielding a combined lifecycle formed from the concatenation of the two originating lifecycles.

The calculus is described by rules 1 to 8 below, the numbering of the formal rules corresponding to the numbering of the informal descriptions above. The following formal expressions are used:

- Capability descriptions are denoted by the (possibly subscripted) variable C. The union of two capability descriptions, written $C_1 \cup C_2$, has any capability available in either of the original descriptions. The intersection of capability descriptions, written $C_1 \cap C_2$, has only those capabilities which are available in both of the original descriptions. $consistent(C)$ is true if there are no conflicting capabilities in C.
- Ontology descriptions are denoted by the (possibly subscripted) variable O. When one capability description, C in ontology, O, can be rewritten to another description, C', in ontology O' by renaming of symbols we write $C \overset{(O,O')}{\rightsquigarrow} C'$.
- Lifecycle descriptions are denoted by the (possibly subscripted) variable L. Instances of this variable can be in any of the following forms: a constant giving the name of a basic source of capabilities or any of the following terms representing a step in the lifecycle: acquisition $a(L)$; union $u(L_1, L_2)$; intersection $i(L_1, L_2)$; generalisation $g(L)$; specialisation $s(L)$; rewriting $r(L, O, O')$ from ontology O to ontology O'; publishing $p(L)$; and concatenation $c(L_1, L_2)$.

- A lifecycle is represented by the expression $C_1 \xrightarrow{(L,O)} C_2$, where C_1 is the capability available at the start of the lifecycle; C_2 is the capability available at the end of the lifecycle; L is the lifecycle description produced in constructing the lifecycle; and O is the ontology within which C_2 is described.
- The expression $\odot(L, C, O)$ denotes that a capability description, C, described within the constraints of ontology, O, has been published as a consequence of lifecycle, L.
- Generalisation occurs when the range of capabilities are extended. We write $C_1 \prec C_2$ when C_1 is generalised to C_2.
- Specialisation occurs when the range of capabilities are reduced. We write $C_1 \succ C_2$ when C_1 is specialised to C_2.

Given this formal syntax, we now give (in order) the formal versions of the informal lifecycle transitions we introduced earlier.

$$\frac{\odot(L, C, O) \in \mathcal{I}}{\odot(L, C, O) \xrightarrow{(a(L),O)} C} \tag{1}$$

$$\frac{C_1 \xrightarrow{(L_1,O)} C_1', \quad C_2 \xrightarrow{(L_2,O)} C_2'}{C_1 \xrightarrow{(u(L_1,L_2),O)} C_1' \cup C_2'} \quad \text{if } \mathit{consistent}(C_1' \cup C_2') \tag{2}$$

$$\frac{C_1 \xrightarrow{(L_1,O)} C_1', \quad C_2 \xrightarrow{(L_2,O)} C_2'}{C_1 \xrightarrow{(i(L_1,L_2),O)} C_1' \cap C_2'} \quad \text{if } C_1' \cap C_2' \neq \emptyset \tag{3}$$

$$\frac{C \xrightarrow{(L,O)} C', \quad C' \prec C_g}{C \xrightarrow{(g(L),O)} C_g} \tag{4}$$

$$\frac{C \xrightarrow{(L,O)} C', \quad C' \succ C_s}{C \xrightarrow{(s(L),O)} C_s} \tag{5}$$

$$\frac{C \xrightarrow{(L,O)} C', \quad C' \overset{(O,O')}{\rightsquigarrow} C''}{C \xrightarrow{(r(L,O,O'),O')} C''} \tag{6}$$

$$\frac{C \xrightarrow{(L,O)} C'}{C \xrightarrow{(p(L),O)} \odot(L, C', O)} \tag{7}$$

$$\frac{C \xrightarrow{(L_1,O_1)} \odot(L_1, C', O_1), \quad \odot(L_1, C', O_1) \xrightarrow{(L_2,O_2)} C''}{C \xrightarrow{(c(L_1,L_2),O_2)} C''} \tag{8}$$

This completes the basic calculus. In the next section we apply it to the example scenario of Section 2.

4 Scenario: Part 2

In Section 2 we gave an example of combination of three different sources of problem solving knowledge. In applying the lifecycle calculus to it we use the same identification symbols, where possible, to make it easier to see how we are following the same development steps. The basic idea is: starting from a selection from a pool of published sources of problem solvers, construct a lifecycle which gives us a new published problem solver. Expressed in terms of the lifecycle calculus:

Given that $\odot(S,C,O) \in \mathcal{I}$
Show that $\exists L, S', C', O'.(\odot(S,C,O) \xrightarrow{L} \odot(S',C',O'))$

The remainder of this section shows how this is done for our example.
The initial pool of capability descriptions contains our table of examples (source name s_e); rulebase for deciding on safe eating of fruit (source name s_r); and a meta-interpreter for Horn clauses (source name s_i). We assume these are all in the same ontology, $o1$. Therefore:

$$\mathcal{I} = \{\odot(s_e, c_e, o1), \odot(s_r, c_r, o1), \odot(s_i, c_i, o2)\} \tag{9}$$

First, we acquire the tabular examples from its published source. This is described by lifecycle rule 1:

$$\frac{\odot(s_e, c_e, o1) \in \mathcal{I}}{\odot(s_e, c_e, o1) \xrightarrow{(a(s_e),o1)} c_e} \tag{10}$$

Then we apply an induction tool to generalise our tabular examples. This uses lifecycle rule 4 on the result of step 10 to give.

$$\frac{\odot(s_e, c_e, o1) \xrightarrow{(a(s_e),o1)} c_e, \quad c_e \prec c_g}{\odot(s_e, c_e, o1) \xrightarrow{(g(a(s_e)),o1)} c_g} \tag{11}$$

We then combine our generalised rule set with the rulebase (source name s_r) for deciding on safe eating of fruit. This requires two steps: the introduction of s_r into the lifecycle (using lifecycle rule 1) and its combination with the result of step 11 (using lifecycle rule 2). The capability description c_4 in step 13 below is $c_g \cup c_r$.

$$\frac{\odot(s_r, c_r, o1) \in \mathcal{I}}{\odot(s_r, c_r, o1) \xrightarrow{(a(s_r),o1)} c_r} \tag{12}$$

$$\frac{\odot(s_e, c_e, o1) \xrightarrow{(g(a(s_e)),o1)} c_g, \quad \odot(s_r, c_r, o1) \xrightarrow{(a(s_r),o1)} c_r}{\odot(s_e, c_e, o1) \xrightarrow{(u(g(a(s_e)),a(s_r)),o1)} c_4} \quad \text{if } \mathit{consistent}(c_4) \tag{13}$$

Our next step is to acquire the meta-interpreter (source name s_i)using lifecycle rule 1; specialise it using lifecycle rule 5; and change its ontology to that of our rule base using lifecycle rule 6.

$$\frac{\odot(s_i, c_i, o2) \in \mathcal{I}}{\odot(s_i, c_i, o2) \xrightarrow{(a(s_i),o2)} c_i} \tag{14}$$

$$\frac{\odot(s_i, c_i, o2) \xrightarrow{(a(s_i),o2)} c_i, \quad c_i \succ c_s}{\odot(s_i, c_i, o2) \xrightarrow{(s(a(s_i)),o2)} c_s} \tag{15}$$

$$\frac{\odot(s_i, c_i, o2) \stackrel{(s(a(s_i)),o2)}{\longrightarrow} c_s, \quad c_s \stackrel{(o2,o1)}{\leadsto} c_n}{\odot(s_i, c_i, o2) \stackrel{(r(s(a(s_i)),o2,o1),o1)}{\longrightarrow} c_n} \tag{16}$$

Finally, we combine our new interpreter from step 16 and the rulebase from step 13 (using lifecycle rule 2) and publish the result (using lifecycle rule 7). The capability description c_5 in step 17 below is $c_4 \cup c_n$.

$$\frac{\odot(s_e, c_e, o1) \stackrel{(u(g(a(s_e)),a(s_r)),o1)}{\longrightarrow} c_4, \quad \odot(s_i, c_i, o2) \stackrel{(r(s(a(s_i)),o2,o1),o1)}{\longrightarrow} c_n}{\odot(s_e, c_e, o1) \stackrel{(u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)),o1)}{\longrightarrow} c_5} \quad \text{if } consistent(c_5) \tag{17}$$

$$\frac{\odot(s_e, c_e, o1) \stackrel{(u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)),o1)}{\longrightarrow} c_5}{\odot(s_e, c_e, o1) \stackrel{(p(u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1))),o1)}{\longrightarrow} \odot(u(u(g(a(s_e)), a(s_r)), r(s(a(s_i)), o2, o1)), c_5, o1)} \tag{18}$$

This completes our example of lifecycle construction. In Section 6 we show how the lifecycle description produced at the end of step 18 can be used to infer relationships between capabilities without having to look at the capabilities themselves. First, we must define a language for inferring this information.

5 Inferring Capabilities from Lifecycle Descriptions

An advantage of publishing lifecycle descriptions along with capabilities for problem solvers is that the capabilities of different problem solvers may be compared in limited ways by looking only at the lifecycle descriptions, rather than performing inference on the capability descriptions. For example, if we have a problem solver which was built by generalisation from another problem solver then we know (just by our knowledge of the lifecycle) that the capabilities of the former are included in the latter. We can apply this style of reasoning to all of the permitted expressions in a lifecycle description to relate the range capabilities associated with outer terms of lifecycle descriptions to those terms immediately inside them. The full set of relationships are given below. We use the following relations between lifecycle descriptions:

- $L_1 = L_2$ when L_1 and L_2 have the same capabilities.
- $L_1 \geq L_2$ when L_1 includes the capabilities of L_2.
- $L_1 \leq L_2$ when L_2 includes the capabilities of L_1.
- $L_1 \approx L_2$ when the capabilities of L_1 and L_2 cannot be related purely from the lifecycle description.

Notice that in only one case is the inclusion relation between capabilities impossible to infer from the lifecycle description. This happens when the sequencing rule (lifecycle rule 8) has been used to splice together two lifecycles but the combination of these involved steps which separately generalised and specialised the capabilities.

$$
\begin{array}{lcll}
a(L) & = & L & \\
r(L,O,O') & = & L & \\
g(L) & \geq & L & \\
u(L_1,L_2) & \geq & L_1 & \\
u(L_1,L_2) & \geq & L_2 & \\
c(L_1,L_2) & \geq & L_1 & \text{if } L_2 \text{ contains neither } i(L',L'') \text{ nor } s(L''') \\
s(L) & \leq & L & \\
i(L_1,L_2) & \leq & L_1 & \\
i(L_1,L_2) & \leq & L_2 & \\
c(L_1,L_2) & \leq & L_1 & \text{if } L_2 \text{ contains neither } u(L',L'') \text{ nor } g(L''') \\
c(L_1,L_2) & \approx & L_1 & \text{if} \neg((c(L_1,L_2) \leq L_1) \vee (c(L_1,L_2) \geq L_1))
\end{array}
$$

6 Scenario: Part 3

We can now return to the running example from Section 6. Its conclusion was the following published description:

$$\odot(u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)),c_5,o1)$$

Without looking at the actual capabilities in c_5 what can we say about these? By applying the lifecycle description relations of Section 5 we can test the connection between c_5 and the capabilities of its basic sources: s_e, s_r and s_i.
The chain of relations to s_e is given below. These all grow larger as the lifecycle description is constructed so c_5 includes all the capabilities of s_e.

$$
\begin{array}{lcl}
u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)) & \geq & u(g(a(s_e)),a(s_r) \\
u(g(a(s_e)),a(s_r) & \geq & g(a(s_e)) \\
g(a(s_e)) & \geq & a(s_e) \\
a(s_e) & = & s_e
\end{array}
$$

The same is true of the link to s_r - the capabilities in c_5 include all its capabilities.

$$
\begin{array}{lcl}
u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)) & \geq & u(g(a(s_e)),a(s_r) \\
u(g(a(s_e)),a(s_r) & \geq & a(s_r) \\
a(s_r) & = & s_r
\end{array}
$$

For s_i, however, the direction of the inequalities changes during the lifecycle, from initial specialisation to subsequent generalisation (when the interpreter was merged with the knowledge base). Therefore, we know that c_5 includes some but not necessarily all of the capabilities of s_i.

$$
\begin{array}{lcl}
u(u(g(a(s_e)),a(s_r)),r(s(a(s_i)),o2,o1)) & \geq & r(s(a(s_i)),o2,o1) \\
r(s(a(s_i)),o2,o1) & = & s(a(s_i)) \\
s(a(s_i)) & \leq & a(s_i) \\
a(s_i) & = & s_i
\end{array}
$$

7 Conclusion

I have given a preliminary description of a calculus for describing lifecycles. Its rules are compact and simple, yet it can be used to derive information which appears useful in managing lifecycles of problem solvers, where we are particularly interested in performing automated inference based on the previous design stages of a system. An important feature of the calculus is that it makes minimal assumptions about the way in which problem solvers and their capabilities are described. It can be used without making explicit what the actual description of the problem solver is, so the problem solvers may be in different, procedural languages. It does make some commitments to the language for capability description - in particular, it must be possible to test for (or attest to, if human endorsement is required) the consistency of combined capabilities. These commitments, however, are the sort we would expect in a capability language in any case.

There are many points for extension of the basic idea. Here are a few:

- The sequencing rule (8) allows us to splice together two lifecycles which share a common, published, capability description. We finessed the problem of deciding when we have "the same" capability description by making this test be identity. The comparison could be much more sophisticated.
- Inequality between capabilities need not be the only information we infer from lifecycle descriptions. There could be many other types of information which we associate with lifecycle stages and might analyse automatically from an appropriate description.
- The inference we present in Section 6 is performed purely on the structure recording lifecycle stages. This was done to demonstrate the most extreme position, where someone looking for information about a problem solver does not need to look at its code or its capabilities - only at its lifecycle. It would, however, be possible to include inference over capabilities (perhaps also accumulated in some structure during lifecycle stages) as well as the more basic form of inference shown in this note.
- The choice of lifecycle steps has been made without considering what designers really require. As noted in Section 3, there is an issue of how to adapt it to be more ergonomic for design.