Domain Specific Problem Description

David Robertson
Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Email: `dr@aisb.ed.ac.uk`

Abstract

Much of software engineering and knowledge engineering has concentrated on generic languages and methods which are supposed to be transferable between domains. By contrast, engineers working in real domains usually employ domain-specific methods and terminology which have evolved from their experience in getting the job done. This paper argues that we have paid too little attention to instantiating generic methods to the demands of specific engineering problems. We advocate narrow but deep studies of carefully chosen domains, with the aim of harnessing domain-specific problem descriptions to guide the construction of software specifications.

We have a problem, which Jackson neatly expresses in [7].

> Software engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed, including reusable components, formal methods, structured methods and architectural studies. These approaches chiefly emphasise the engineering product; the solution rather than the problem it solves.

The difficulty is that reliably engineered solutions require succinct descriptions of the problem which is to be solved, expressed in a form which is compatible with the engineering conventions of the domain of application. Luqi & Goguen [10], envisage a branch of formal methods occupying part of this territory:

> There is much more to formal methods than suggested by the themes dominant in the past, namely synthesis and correctness proofs for algorithms. ...A number of recent successful applications of formal methods seem to form a cluster suggesting a new paradigm for applying formal methods. These applications

> involve a tool having all or most of the following attributes: ... A narrow, well defined, and well understood problem domain ... a coherent user community ... [an] interface that is intuitive to the user community ... The tool takes a large grain approach ... Inside the tool is a powerful engine that encapsulates formal methods concepts ... [but] ... users do not have to know how it works or even that it is there. A name for this emerging paradigm might be *Domain Specific Formal Methods*, in recognition of the role played by the user community and their specific domain.

In other words, there may exist classes of problem for which it is worth studying, formally, how problems are described and used to constrain the design of solutions. In doing this we hope to understand more precisely where certain styles of specification are appropriate. We also hope to be able to supply tools which enable problems to be solved reliably in these domains without requiring a deep understanding of some general-purpose formal method.

1 The Relationship Between Problem and Solution

The distinction between problem and solution description is often blurred. High level specification languages allow us to prototype systems rapidly. This tempts software engineers to "go for the jugular" by suggesting a solution with only minimal awareness of the problem. This sometimes works and, as Jackson [7] points out, gives software engineers the comfort of working with notations which they know will lead to executable software. However, Jackson also identifies the weakness of this approach: "the developer is then easily persuaded that a description of the software *is* a description of the problem. But it is scarcely more so than a description of a car steering wheel is a description of the physiology of the driver's arm and hand." The terminology of some specification languages help to perpetuate this misconception. For example, a cornerstone of logic programming is that [15] "the knowledge about the problem and assumptions that are sufficient to solve it be stated explicitly, as logical axioms." It is easy to mistake this distinction (between declarative and imperative description) for the orthogonal distinction between problem and solution domains.

In practice, the main difference between the two domains is in how we discipline their engineering. This is best understood by considering the ideal case for constructing a solution, in which we have a formally expressed description of the assumptions we are making about the problem and the properties which our solution should have. The job of the engineer is to define a system which, based on the assumptions, possesses those properties. A good engineer will be guided in this task by standard practice in the specification language chosen for the job - *i.e.* the constraints on solution design come from experience with the specification language. By contrast, problem description requires us to decide what the assumptions and properties will be. A good engineer will be guided by standard practice within the domain of application - *i.e.* the constraints on problem description come from experience in the domain. This is a crucial distinction because it means that if we want systems of problem description which enforce good engineering principles then these are unlikely to come from abstract formal methods. We must instead harness our formal specification languages to describe standard engineering practice in target domains.

2 Sources of Problem Descriptions

The way in which we analyse problem domains effects the types of problem descriptions which we derive. There are a variety of strategies for doing this. We could perform an intensive study of a narrow domain and try to understand the whole process of problem description through to specification of solutions. Alternatively, we might make a more extensive classification of typical problem types. Each approach has both advantages and disadvantages, as we can appreciate by considering some examples.

The ECO project [14] involved formal problem description in a single domain for which the conceptual gap between problem description and solution is large. Our "clients" were chiefly animal population ecologists who, although they have strong intuitions about the key components and functional relationships in their ecological problems, have little expertise in constructing appropriate simulation models as solutions. The difference between language of problem and description in this domain is stark. Typical problem descriptions would include statements such as "Wolves prey on deer" and "The mortality of deer varies with their location" - simple phrases laden with ecological meaning. Typical solution descriptions would include spatial grids to represent animal location, with numbers of animals of different types associated with grid elements and transfer functions between them; procedures for aggregating and disaggregating populations into age and sex classes; and mechanisms for controlling the dynamics of the simulation over time. In addition to representing the domain specific components of problem and solution descriptions, we had to define methods for constructing solutions on the basis of problem descriptions. These were also (necessarily) domain specific because we were trying to encode a small part of the engineering abilities of a human modelling expert.

Part of the knowledge based systems community has attempted to classify problem description types across a broad range of domains commonly encountered in this line of work. Motivating efforts in this area is the belief that standard patterns of description are frequently re-used and that by recognising these we may help establish better engineering practices. A dominant system of classification is KADS [16], one result of which is the Common KADS library of problem solving methods. This is an hierarchical structure of concepts recognisable to those who construct this type of system (*e.g.* hypothesis generation, test selection), some of which are associated with standard patterns of implementation. The KADS method of system development is used commercially (see for example [8]) but development of a standard library has turned out to be problematic. A major obstacle, described by Breuker [2], is that there is not a clear mapping from KADS problem solving methods to types of problem in application domains: "It is not even clear how to describe problems in a generic way, in particular because the solutions may contain all kinds of elaborations ...This is a serious problem because it makes it difficult to state which PSM solves which type of problem." At the root of this difficulty is that KADS models are more a guide to standard architectures for solutions than they are a description of standard types of application problem. There is no reason to suspect that a picture of standard practice for constructing solutions will match in a simple way to a picture of standard types of problem, and much of the domain-specific content of our later ECO systems was concerned with controlling this type of matching.

Attempts have been made from within application domains to standardise description of problems without making commitments to forms of solution. One example is IBM's Business System Development Method which is a structured but informal scheme for describing the key components and processes of businesses. The aim of BSDM is to allow those in a business to produce more precise analyses of business requirements in order that [6] "...business decisions are made by business people. Technical decisions are still made, but only after business decisions." Like KADS, the developers of BSDM are attempting to construct a library of standard design patterns.

Unlike KADS, these are derived from categories of business operation - not from types of solution architecture. This gives BSDM a deficiency which is the reverse of that for KADS: whereas KADS has difficulty matching its solution frameworks to problems, BSDM has no mechanism for matching its problem descriptions to appropriate solutions. This issue resurfaces in Section 3.

The examples above follow a pattern. Those (like Breuker) who attempt to produce descriptions of a broad range of solution architectures independently of describing problem domains run into difficulty in connecting the diverse range of architectures to plausible problem descriptions. Conversely, general-purpose methods for describing problem domains (like BSDM) are difficult to realise as software architectures. Systems (like ECO) which attempt to make explicit the expertise required to link problem descriptions to solution specifications either require the languages of problem and solution to be close or demand a knowledge engineering effort in order to form the connection. This bridging expertise is difficult to elicit and express, yet it is the material which is required in order to understand how solutions are engineered for classes of problems.

3 Problem Descriptions Must (Eventually) be Executable

A luxury available in an ideal solution description, where assumptions and required properties of the specification are known, is that we can test that these properties hold of the solution. This is not true of problem descriptions, where we must rely on an ontology (a restricted language appropriate to the domain) to allow us to validate our descriptions against what we believe to be the real requirements. Having a standard ontology, however, does not guarantee a valid description because we can not be sure that those who use the ontology have the same interpretation of its symbols as those who developed it. One of the best examples of this is documented by Manfred Aben [1], who studied the proceedings of the second KADS user meeting and concludes that out of the nine papers using a specific ontological primitive only one used it as originally defined by its authors. Such ambiguities are not easily resolved by discussion - nor are they settled by appealing to the semantics of the problem description language. The most convincing test (although limited by computational constraints) is to explore what the problem description might imply about the behaviour of the system.

This form of exploration can occur either by inferring consequences within the problem description or by moving rapidly to an executable specification of the solution (see [9] and [4] for a discussion of the value of executable specifications in rapid prototyping).

The scope for inference within problem description languages is often limited and inference steps may not always be deductive. Our problem descriptions in the ECO project supported inferences such as the following.

The ecological knowledge base contains an axiom (in a sorted logic) stating that if any attribute of an object varies with its location then it is spatially represented. It also contains an axiom stating that if the spatial representation of an object is by grid squares then it is spatially represented.

$$\forall X : object, A : attribute.varies\_with(A, X, location, X) \rightarrow spatial\_rep(X)$$
$$\forall X : object.grid\_squares(X) \rightarrow spatial\_rep(X)$$

An ecologist might assert that the biomass of wolves varies with their location.

$$\forall X : wolf.varies\_with(biomass, X, location, X)$$

We can deduce (knowing that *biomass* is a type of *attribute*) that wolves are spatially represented.

$$\forall X : wolf.spatial\_rep(X)$$

We can also suggest (abductively) that the spatial representation of wolves might be based on grid squares.

$$\forall X : wolf.grid\_squares(X)$$

This form of simple inference, highly constrained, was used to suggest to ecologists the general consequences of the information they volunteered to the system. It gave little clue to the consequences of ecologists' assertions as far as the range of possible solutions was concerned.

However, the ECO program generator allowed us rapidly to produce executable models from those early descriptions, giving ecologists a fast route to finding out if the solutions ECO was capable of giving them were the ones they wanted to have. This was done by an interactive program generator which pieced together logic programs using schematic definitions of predicate groups - each group corresponding to a standard form of model description on this domain. An example of a schema is shown below. When instantiated by the program generator, this defines a method for calculating whether two objects, A and B are in proximity on a spatial grid at time T. The schema is one of a number of methods for constructing a program to satisfy this goal. It is distinguished from other methods by its preconditions, which reference the problem description - in this case, checking whether the problem stipulates the numbers of grid rows and columns, and collecting together all the objects in the problem description which are sensitive to location (*e.g.* by belonging to a set of objects which are spatially represented). If these apply then the actions associated with the schema construct any additional structure required in instantiating its clauses - in this case by constructing a grid square data structure with appropriate rows and columns, and prompting the ecologist for locations of objects on the grid.

Goal:	$in\_proximity(A, B, T)$
Clauses:	$\{(in\_proximity(A, B, T) \leftarrow$ $\quad located\_at(A, L1, T)\ \&$ $\quad adjoining(L1, L2)\ \&$ $\quad located\_at(B, L2, T)),$ $(in\_proximity(A, B, T1) \leftarrow$ $\quad located\_at(A, L1, T1)\ \&$ $\quad located\_at(B, L1, T1)\ \&$ $\quad \neg A = B)\} \cup Grid \cup Locations$
Preconditions:	$in\_problem(number\_of\_grid\_rows(NRows))\ \&$ $in\_problem(number\_of\_grid\_columns(NCols))\ \&$ $location\_sensitive\_objects(Objs))$
Actions:	$make\_grid\_square\_representation(NRows, NCols, Grid)\ \&$ $prompt\_for\_locations(Objs, NRows, NCols, Grid, Locations)$
Subgoals:	$\{located\_at/3\}$

Although modular, schemata are not self contained and normally require the program generator to construct other parts of the program for predicates needed by, but not defined in, the schema. The example above requires further generation to provide a definition of the $located\_at/3$ predicate. This gives an extra degree of flexibility to schemata but also requires careful structuring of the schema library to ensure that sub-definitions are compatible (in ecological terms) with the schemata which require them.

4 Problem Description Must be Incremental

One of our early setbacks in the ECO project was to find that a carefully crafted problem description system failed in tests with ecologists. The reason for failure was not some representational deficiency in the problem description language - it was because our ecologists did not have a convenient place to begin talking about the problem. It was easy for them to follow other people's descriptions but not to start their own. This "blank sheet of paper syndrome" led us to investigate methods of incremental description. Formal languages are versatile in supporting this, as can be demonstrated by tackling a simplified example from recent collaborative work with Agusti. An informal description of the example problem is as follows:

> A configuration problem requires us to produce designs based on: the components available; a specification of the expected properties of the design; models of standard types of configuration; a description of the current state of our (partial) configuration; a memory of those components which we have rejected; and a selection of the "top" component amongst those we might want to use in the next stage of configuration.

We show how the description of this problem might be done incrementally, starting with a pre-supplied template (specific to this type of task) and elaborating it in several steps.

Initial Problem Template

We begin the description by introducing the basic data types from the informal description, a set-based language. All the terms in our language denote sets and we introduce new terms as subsets of existing terms or of the universal set, *top*:

$$\begin{aligned} specification &\subseteq top \\ design &\subseteq top \\ models &\subseteq top \\ components &\subseteq top \\ partial\_config &\subseteq top \\ rejected &\subseteq components \\ top\_comp &\subseteq components \end{aligned}$$

In addition to the basic data types we supply functions ranging over these sets, according to the major functional relationships in the configuration task.

$$\begin{array}{rcl}
configuration(components, specification, models, & & \\
partial_config, rejected, top_comp) & \subseteq & design \\
identify(specification) & \subseteq & components \\
select(components, partial_config, rejected) & \subseteq & top_comp \\
combine(top_comp, partial_config, models) & \subseteq & partial_config
\end{array}$$

Step 1: Strengthening functional relationships

The functional expressions of the problem template are weak in the sense that they describe each function over its argument sets in general but say nothing about its particular behaviour over any given subset of the general sets. One way of strengthening the expressions is to define them over all subsets of the original sets by introducing (implicitly) universally quantified variables.

$$\begin{array}{l}
C \subseteq components \ \& \ S \subseteq specification \ \& \ M \subseteq models \\
\& \ P \subseteq partial_config \ \& \ R \subseteq rejected \ \& \ T \subseteq top_comp \ \rightarrow \\
\qquad configuration(C, S, M, P, R, T) \subseteq design
\end{array}$$

$$\begin{array}{l}
S \subseteq specification \ \rightarrow \\
\qquad identify(S) \subseteq components
\end{array}$$

$$\begin{array}{l}
C \subseteq components \ \& \ partial_config \ \& \ R \subseteq rejected \ \rightarrow \\
\qquad select(C, P, R) \subseteq top_comp
\end{array}$$

$$\begin{array}{l}
T \subseteq top_comp \ \& \ P \subseteq partial_config \ \& \ M \subseteq models \ \rightarrow \\
\qquad combine(T, P, M) \subseteq partial_config
\end{array}$$

Step 2: Refining the specification

We are now in a position to take each of the axioms from the previous step and add more detail to their description. To save space, we shall do this only for the *configuration* function, which is the main component of the specification. The question we ask at this stage is how we may decide on valid configurations based on the other information we have available. To do this, we have introduced a recursive definition of $configuration$ and split this across three axioms. The first of these describes the case where we have obtained, with the combination of *top_component* and *partial_spec* a design which satisfies the verification and completeness tests. The second axiom for $configuration$ covers the case where the addition of a *top_component* does not provide a verifiable design, in which case we may obtain a configuration if it is added to the set of *rejected* components and a new top component is selected. The third axiom covers the case where the addition of a *top_component* provides a design which is verifiable but not complete, in which case we continue to configure with the top component included in the partial configuration.

$$
\begin{aligned}
&C \subseteq components \;\&\; S \subseteq specification \;\&\; M \subseteq models \;\&\\
&P \subseteq partial\_config \;\&\; R \subseteq rejected \;\&\; T \subseteq top\_comp \;\&\\
&D \subseteq combine(T, P, M) \;\&\; verify(D, S) \;\&\; complete(D, S) \;\rightarrow\\
&\qquad D \subseteq configuration(C, S, M, P, R, T)
\end{aligned}
$$

$$
\begin{aligned}
&C \subseteq components \;\&\; S \subseteq specification \;\&\; M \subseteq models \;\&\\
&P \subseteq partial\_config \;\&\; R \subseteq rejected \;\&\; T \subseteq top\_comp \;\&\\
&X \subseteq combine(T, P, M) \;\&\; not(verify(X, S)) \;\&\\
&U \subseteq union(T, R) \;\&\; N \subseteq select(C, P, U) \;\&\\
&D \subseteq configuration(C, S, M, P, U, N) \;\rightarrow\\
&\qquad D \subseteq configuration(C, S, M, P, R, T)
\end{aligned}
$$

$$
\begin{aligned}
&C \subseteq components \;\&\; S \subseteq specification \;\&\; M \subseteq models \;\&\\
&P \subseteq partial\_config \;\&\; R \subseteq rejected \;\&\; T \subseteq top\_comp \;\&\\
&X \subseteq combine(T, P, M) \;\&\; verify(X, S) \;\&\; not(complete(X, S)) \;\&\\
&N \subseteq select(C, X, R) \;\&\; D \subseteq configuration(C, S, M, X, R, N) \;\rightarrow\\
&\qquad D \subseteq configuration(C, S, M, P, R, T)
\end{aligned}
$$

Step 3: Translation to a Horn–clause specification

The set expressions of the previous step are now sufficiently detailed that they are a step away from a conventional Horn–clause specification. One way of taking this step is to replace all inclusions of the form $X \subseteq P(A_1, \cdots, A_n)$, where X is a variable, with predicates of the form $P(A_1, \cdots, A_n, X)$. The intuition here is that the predicates can generate instances of the individuals inhabiting the subsets referred to by X. This transformation applied to the first axiom for $configuration$ gives:

$$
\begin{aligned}
&components(C) \;\&\; specification(S) \;\&\; models(M) \;\&\\
&partial\_config(P) \;\&\; rejected(R) \;\&\; top\_comp(T) \;\&\\
&combine(T, P, M, D) \;\&\; verify(D, S) \;\&\; complete(D, S) \;\rightarrow\\
&\qquad configuration(C, S, M, P, R, T, D)
\end{aligned}
$$

The same principle could be applied to the others.

Provided that we supplied additional definitions for the auxiliary predicates required by the preconditions of *configuration*, we could use the derived Horn–clauses as an executable specification for this particular interpretation of configuration. Our specification is not computationally efficient but there exist various methods for transforming inefficient logic programs into more efficient ones (see for example [5]). A more pressing problem is that the decisions we take during this sort of description (although precise in formal terms) are not clearly identified with tasks of relevance to some clearly identifiable group of engineers. They are motivated by what feels elegant in a logical language rather than what is appropriate to the culture of an application domain. In the later stages of formal specification this may be unavoidable (unless the domain is narrow enough to allow automatic programming) but in the early stages, where domain engineers must call the shots, it would be fatal. Cleland & Mackenzie [3] claim that this failure to relate to engineering as it is practised is one of the main impediments ot the development of industrial formal methods. We do not have a general theory of domain engineering so we need to separate the different types of expertise required at the many stages of problem description and define clearly the interfaces between each stage.

5 Problem Description Involves Diverse Engineering Styles

A large number of different engineering styles may have to interact in producing a specification. For a stock control system, for example, it may be necessary to combine the engineering talents of database engineers, process control engineers and business analysts in order to obtain a comprehensive picture of what needs to be done. Many of these groups are mature enough to have developed distinct styles of specification and are, rightly, resistant to change in the ways they do their jobs. Although it may in theory be desirable for them all to share the same general design methods and ontology, there is no prospect of this happening in practice. A more productive approach might be to deepen our formal understanding of each style of specification, with the aim of understanding each in the same formal language but without unduly disturbing current practice. If this can be done, even for a useful subset of each specification style, then we may be able to share the results of specification between disciplines without requiring a revolutionary change in work practices within each one. This is the motivation for recent work on a Lightweight Specification System [11] which provides a set of simple tools directed at particular styles of specification, interconnected only by the use of Horn-clause logic as a lingua franca between tools. The specification styles currently covered by the tool set are given on the left side of the table below, with the subset of Horn-clause logic produced by each tool on the right.

Specification Style	Subset of Horn-clauses
"Box and arrow" process sequencing, inspired by BSDM modelling	Definite Clause Grammars
Diagrammatic description of recursive problem solving	Recursive predicates with a single nominated "output" argument
Editing of logic programs by applying standard programming techniques	Horn-clauses in the Prolog style but excluding predicates sensitive to clause or subgoal ordering
Transformation of logic programs for efficiency	Prolog programs
Using a network diagram to connect groups of definitions	Prolog programs
Using an argumentation network to comment on key decisions in specification and link to development sequences	Prolog programs augmented with an IBIS-like argumentation network
Sketching the structure of windows and their major interactions in the user interface to a specification	Prolog programs containing system specific interface procedures

Each of the LSS tools is intended to be easy for those in particular engineering cultures to pick up. There are also obvious opportunities for communication between tools, via the underlying formal language. For example, in [12] we describe how a combination of the process modelling, recursion, techniques editing, transformation and user interface tools an be applied to the design of a simple diagnostic system. In [13] we describe a more extensive experiment in applying LSS to the specification of one of its own tools.

It is interesting to compare the approach adopted in LSS with our earlier ECO work. ECO was based on an intensive collaboration with domain specialists over several years. As a result, it contained a substantial amount of domain-specific information and was highly prescriptive about how this could be used in its own (narrow) domain of application. Many of the LSS tools are based on a much more superficial understanding of the engineering principles within a domain - allowing communication in the styles appropriate to particular engineering groups but not attempting to automate the engineering process itself.

6 Summary and Conclusions

This paper is inspired by recent developments in the software engineering, formal methods and artificial intelligence communities. Jackson [7] presents a compelling argument for the development of specialised software engineering disciplines, aligned to problem domains rather than specification languages. Luqi & Goguen [10] note the effectiveness of formal methods applied to carefully restricted domains for which the formal notation and inference methods may be tailored to existing standard practice. Breuker [2] notes the need within the knowledge based systems community to understand problem solving methods in terms of application domains. These representatives of diverse research communities are all looking for roughly the same thing - a more rigorous understanding, in engineering terms, of how problems are solved in carefully circumscribed domains.

Research in problem description is based on the assumption that it is worthwhile to describe problems in a formal language tuned to a domain of application, not driven by whatever paradigm is currently fashionable in software engineering. If Jackson is right, then a better understanding of problem description might help to develop more precise engineering specialisation in software design - but only if we can chart standard routes from problem description to executable specification. Most of the systems currently available from the requirements engineering or software engineering community give only half of this picture. Most requirements engineering systems concentrate on methods and ontologies for problem description but do not describe how an engineering specialist may use these to engineer software specifications. Most CASE tools present a high level interface to a software engineering language, leaving it to domain specialists to adapt this to their variety of engineering. Comparatively few researchers have tried to fit both halves of the picture together. From the experiments so far, it seems that success depends on the ability to balance the demands of formality and communication. The languages of problem and solution must be formal (in a broad sense) or there can be no precision in our engineering. However, they must match to the styles of engineering employed in target domains - a useful general purpose problem description method is about as likely as a general purpose mechanical engineering method.

This paper could not be an exhaustive analysis of this area of research because there does not yet exist a coherent research field - it draws on ideas from software engineering, requirements engineering, knowledge engineering and computational logic but is not subsumed by any single one of these. We have, instead, described some of the defining characteristics of problem description systems: the requirement for executability in order to confirm ontological understanding; the desirability of incremental description as a means of starting simply and gradually incorporating complexity of description; and the need to accommodate numerous styles of communication in recognition of the diversity of engineering styles even within narrow domains.

References

[1] M. Aben. *Formal Methods in Knowledge Engineering.* PhD thesis, University of Amsterdam, 1994. ISBN 90-5470-02809.

[2] J. Breuker. Reusable components for artificial problem solvers: The common kads library experience. In *Proceedings of IJCAI'93 Workshop on Knowledge Sharing and Information Interchange*, 1994. Short position paper.

[3] G. Cleland and D. MacKenzie. Inhibiting factors, market structure and the industrial uptake of formal methods. In *Workshop on Industrial-Strength Formal Specification Techniques*, pages 46–60, Boca Raton, Florida, 1995.

[4] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, September, 1992.

[5] N.E. Fuchs and M.P. Fromherz. Schema-based transformations of logic programs. In T.P. Clement and K. Lau, editors, *Logic Program Synthesis and Transformation, Workshops in Computing (Proceedings LOPSTR-91.* Springer Verlag, 1992.

[6] IBM. Business system development method technical descriptions. Technical Report GE19-5387, SC19-5309, SC19-5310, SC19-5312, SC19-5313, International Business Machines Corporation, 1992.

[7] M. Jackson. Problems, methods and specialisation. *Software Engineering Journal*, November:249–255, 1994.

[8] J. Kingston, J. Doheny, and I. Filby. Evaluation of workbenches which support the commonkads methodology. *Knowledge Engineering Review*, 10(3), 1995.

[9] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, 1989.

[10] Luqi and J.A. Goguen. Formal methods: Promises and problems. *IEEE Software*, in press, 1996.

[11] D. Robertson. Lightweight formal specification. In *Proceedings of ONR/ARPA/AFOSR/ARO/NSF workshop on Increasing the Practical Impact of Formal Methods for Software Architectures*, Monterey, California, 1995.

[12] D. Robertson. Distributed specification. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1996. Submitted to European Conference on Artificial Intelligence 1996, available as DAI research paper.

[13] D. Robertson. An empirical study of the lss specification toolkit in use. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1996. Submitted to 8th International Conference on Software Engineering and Knowledge Engineering, available as DAI research paper.

[14] D. Robertson, A. Bundy, R. Muetzelfeldt, M. Haggith, and M Uschold. *Eco-Logic: Logic-Based Approaches to Ecological Modelling.* MIT Press (Logic Programming Series), 1991. ISBN 0-262-18143-6.

[15] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1985.

[16] B. J. Wielinga, A. Th. Schreiber, and J. A. Breuker. KADS: A modelling approach to knowledge engineering. *Knowledge Acquistion Journal*, 4(1):5–53, 1992. Special issue 'The KADS approach to knowledge engineering'. Reprinted in: Buchanan, B. and Wilkins, D. editors (1992), *Readings in Knowledge Acquisition and Learning*, San Mateo, California, Morgan Kaufmann, pp. 92-116.