ecai1

ECAI 96. *12th European Conference on Artificial Intelligence*
Edited by W. Wahlster
Published in 1996 by John Wiley & Sons, Ltd.

Distributed Specification

David Robertson[1]

Abstract. Most existing work on formal specification is focussed on a particular method or specification language, considered in isolation. In practice, few non-trivial specifications are produced by a single person, or by a group of persons with a common view of the world. It is far more common for a variety of views of a problem to coexist, each with different forms of communication. It has proved difficult to assemble such heterogeneous specifications - leading to breakdowns in communication and consequent failures of systems. As more of this communication is conducted remotely by electronic means the need to support distributed specification in a controlled way is increased. This paper presents one way of tackling this problem, based on a set of tools for describing specifications at a variety of stages in their development. By constraining the interfaces between tools, we aim to provide a more structured system for collaborative specification.

1 Introduction

Few people would disagree that software specification is a poorly constrained and ill-disciplined process. It was hoped that computational logic would ease this difficulty by elevating specifications from instruction-style programming to declarative problem description. This change of perspective is often helpful but it does not lay to rest a number of key problems endemic to the process of specification - it merely exhibits these in another form. In particular:

- We have been slow to develop a clear idea of standard practice in problem description. Most tools for formal specification leave this to experience and imagination rather than reinforcing well understood patterns of design.
- The "flat", undifferentiated structure of a specification often makes it difficult to spot the key intuitions which make its purpose obvious. The normal textual descriptions of logical formulae are sometimes not the best vehicles for expressing such intuitions.
- With some exceptions (*e.g.* [8]), there have been few attempts to reinterpret existing informal software specification methods using computational logic. There are even fewer examples of such attempts resulting in systems simple enough to use in practice. Thus, there is a gap between what domain specialists want to say and the means by which computational logic allows them to say it.
- Although sophisticated systems for verification of logical specification have been developed, there is no evidence that validating such specifications has been made any easier - and it is here that many of the problems are thought to occur.

Hitherto, these problems have received little attention - partly because logical specifications are normally produced by dedicated specialists working alone on experimental prototypes. This is not a sustainable position, since most real specifications are produced collaboratively. Nor is it realistic to expect all parties to specification to learn the details of a computational logic - they have better things to do with their time. What we need is a system of collaborative specification which divides the task in a structured way, allowing the various participants to contribute via tools which are reasonably familiar to them but which have clear interfaces to the tools of others. This paper describes the first version of such a toolset, which we call a Lightweight Specification System because each tool is intended to be easy to pick up (see [6] for further discussion of lightweight specification). Although a prototype, LSS has been applied to significantly sized specification tasks - the largest of which is the definition of one of its own tools (see [7] for details). This was a particularly difficult test because the idiosyncrasies of application code (even in a logic programming environment) are often resistant to declarative specification. Nevertheless, LSS could specify just over half of the 85 predicates (constituting 850 lines of code) of that application.

Space limitations prohibit a full description LSS. Each of the sections 2 to 8 below summarises a separate tool according to the main feature of specification which it is designed to support. Connecting these sections is a running example, in which a component of a basic diagnostic specification is designed (Section 2); fitted within a more general diagnostic description (Section 3); made more compact (Section 4); merged into a specification for patient processing (Section 5); supplied with an interface (Section 6); the macro-structure of the combined specification reviewed (Section 7); and parts of the specification annotated with justifications of design decisions (Section 8). All of the tools described in this paper are operated using graphical interfaces which make the formal manupulations quick to perform. In this paper we concentrate on the underlying mechanisms rather than the interface. Those interested in the latter are referred to [7]. A selection of the tools are publicly available as Macintosh binaries from `ftp://ftp.dai.ed.ac.uk/pub/dr`.

2 Reinforcing Standard Practice

Although in theory computational logic admits a wide variety of means of expression, in the more mature languages this flexibility is restrained by established practice. Ways of describing many of the standard components of specifications are sufficiently well understood that we no longer need to work them out from first principles each time we build a specification. This can accelerate development; help to avoid "unconventional" formulations of standard specifications; and make it easier to explain specifications to others from within the same development culture. Techniques editing (first suggested by [4] and later developed by, for example, [1], [3], [2]) is one way of making standard practice more explicit for logic programming. This is the basis for our techniques tool, which we explain by example below.

Suppose that in a diagnostic system we want to build a specification for a predicate $covered(Symptoms, Hypotheses, Diagnoses)$, which holds when $Diagnoses$ is a subset of an $Hypotheses$ set for which each diagnosed disease is a plausible conclusion based on the currently known $Symptoms$. Given that this will be recursive, it is standard practice to first consider how the recursion is controlled. We make this choice by browsing through a library of standard partial definitions until an appropriate one is found. This library does not

[1] Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, Scotland

ECAI 96. *12th European Conference on Artificial Intelligence*
Edited by W. Wahlster
Published in 1996 by John Wiley & Sons, Ltd.

need to be huge because there is flexibility to adapt the partial definitions later - the current system uses about 30 basic definitions. We choose the definition for a binary split over a list. This succeeds on an empty list or on either of two recursive clauses, depending on appropriate tests ($\mathcal{T}_1$ and $\mathcal{T}_2$) on the first element of the list. The definition is partial because we have yet to define these tests and we have also to add the arguments for symptoms and diagnoses.

$$
\begin{array}{l}
covered([]) \\
covered([H|T]) \leftarrow \\
\quad \mathcal{T}_1(\{H\}) \wedge \\
\quad covered(T) \\
covered([H|T]) \leftarrow \\
\quad \mathcal{T}_2(\{H\}) \wedge \\
\quad covered(T)
\end{array}
$$

As an aside, notice that we can suggest a number of properties which we would expect (though not guarantee) this partial program to possess. For example it should terminate, have a well founded ordering and its clauses are normally mutually exclusive. The system has a record of these types of assumption, expressed in the simple style shown below, and we shall return to this in Section 7.

$$
\begin{array}{l}
terminates(covered) \\
wfo(covered) \\
mutually\_exclusive\_clauses(covered)
\end{array}
$$

Returning to our running example, we now decide on the role which $Symptoms$ must play in the definition. In this case it is simply carried through the recursion and this extension is performed by a single editing operation.

$$
\begin{array}{l}
covered(Symptoms, []) \\
covered(Symptoms, [H|T]) \leftarrow \\
\quad \mathcal{T}_1(\{H\}) \wedge \\
\quad covered(Symptoms, T) \\
covered(Symptoms, [H|T]) \leftarrow \\
\quad \mathcal{T}_2(\{H\}) \wedge \\
\quad covered(Symptoms, T)
\end{array}
$$

We can now instantiate $\mathcal{T}_1$ and $\mathcal{T}_2$ in order to distinguish between the cases when the first element of the list is or is not a plausible conclusion, based on the symptoms.

$$
\begin{array}{l}
covered(Symptoms, []) \\
covered(Symptoms, [H|T]) \leftarrow \\
\quad plausible\_conclusion(Symptoms, H) \wedge \\
\quad covered(Symptoms, T) \\
covered(Symptoms, [H|T]) \leftarrow \\
\quad not(plausible\_conclusion(Symptoms, H)) \wedge \\
\quad covered(Symptoms, T)
\end{array}
$$

Our remaining problem is how to obtain the appropriate diagnoses. One way to do this is by introducing an argument which relates the final diagnoses set to a partial diagnoses set obtained from the recursive subgoals. This is added in a single operation which introduces three new "gaps" in the specification: the means of instantiating the final diagnoses in the base case ($\mathcal{U}_1$) and the means of relating the partial to final diagnoses in the recursive cases ($\mathcal{U}_2$ and $\mathcal{U}_3$).

$$
\begin{array}{l}
covered(Symptoms, [], Final) \leftarrow \\
\quad \mathcal{U}_1(\{\}, Final) \\
covered(Symptoms, [H|T], Final) \leftarrow \\
\quad plausible\_conclusion(Symptoms, H) \wedge \\
\quad covered(Symptoms, T, P) \wedge \\
\quad \mathcal{U}_2(\{P\}, Final) \\
covered(Symptoms, [H|T], Final) \leftarrow \\
\quad not(plausible\_conclusion(Symptoms, H)) \wedge \\
\quad covered(Symptoms, T, P) \wedge \\
\quad \mathcal{U}_3(\{P\}, Final)
\end{array}
$$

Our final step is to fill the gaps in the partial specification by instantiating $\mathcal{U}_1$ to $Final = []$; $\mathcal{U}_2$ to $Final = [H|P]$; and $\mathcal{U}_3$ to $P = Final$. These (with the explicit unifications automatically made implicit by the system) yield the definition below.

$$
\begin{array}{l}
covered(Symptoms, [], []) \\
covered(Symptoms, [H|T], [H|P]) \leftarrow \\
\quad plausible\_conclusion(Symptoms, H) \wedge \\
\quad covered(Symptoms, T, P) \\
covered(Symptoms, [H|T], Final) \leftarrow \\
\quad not(plausible\_conclusion(Symptoms, H)) \wedge \\
\quad covered(Symptoms, T, Final)
\end{array}
$$

3 Highlighting Structural Features

The techniques tool imposes structure on the process of specification by appealing to standard practices familiar to logic programmers. However, if the specification is to be distributed amongst several specialists it may not always be a logic programmer who is most competent to supply the initial structure of a predicate. The recursion tool provides a visual language for sketching the initial structure of recursive predicates which "return" a result in one of their arguments. Most people are accustomed, from school, to seeing set inclusions depicted as enclosures of boxes or circles. The recursion tool makes use of this visual metaphor to represent the key base and recursive subgoals of a predicate as box inclusions and the other subgoals (necessary for type checking or obtaining intermediate results) as separate boxes linked to those predicates which are dependent on them. For example, suppose that we want to build a larger diagnostic system which incorporates the definition of $covered$ which we constructed in the previous section. This uses $covered$ as the base case but has another recursive case where we revise our current hypotheses. The diagram for this is shown in Figure 1.

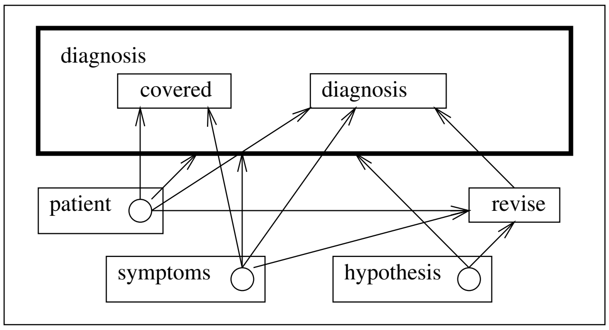

Figure 1. recursion diagram for our running example

The system automatically translates this diagram into the specification below.

$$
\begin{array}{l}
diagnosis(P,S,H,D) \leftarrow \\
\quad patient(P') \wedge \mathcal{U}_1(\{P'\},P) \wedge \\
\quad symptoms(S') \wedge \mathcal{U}_2(\{S'\},S) \wedge \\
\quad hypotheses(H') \wedge \mathcal{U}_3(\{H'\},H) \wedge \\
\quad covered(S,H,D') \wedge \mathcal{U}_4(\{D'\},D) \\
diagnosis(P,S,H,D) \leftarrow \\
\quad patient(P') \wedge \mathcal{U}_5(\{P'\},P) \wedge \\
\quad symptoms(S') \wedge \mathcal{U}_6(\{S'\},S) \wedge \\
\quad hypotheses(H') \wedge \mathcal{U}_7(\{H'\},H) \wedge \\
\quad revise(P,S,H,Hr) \wedge \\
\quad diagnosis(P,S,Hr,D') \wedge \mathcal{U}_8(\{D'\},D)
\end{array}
$$

Notice the large number of gaps in this definition ($\mathcal{U}_1$ to $\mathcal{U}_8$). This is because the graphical system doesn't have the representational sophistication necessary to define the fine details of the relationships between predicates. However, these gaps can be dealt with by the techniques tool, thus refining the specification. In the simplest case, we could instantiate each $\mathcal{U}_i(\{X'\},X)$ to $X' = X$, providing a basic recursive definition which we pick up in the next section.

4 Transforming for Efficiency

The definition of $diagnosis$ which we produced at the end of the previous section is:

$$
\begin{array}{l}
diagnosis(P,S,H,D) \leftarrow \\
\quad patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge \\
\quad covered(S,H,D) \\
diagnosis(P,S,H,D) \leftarrow \\
\quad patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge \\
\quad revise(P,S,H,Hr) \wedge \\
\quad diagnosis(P,S,Hr,D)
\end{array}
$$

The duplication of identical $patient$, $symptoms$ and $hypotheses$ conditions in both clauses, although perfectly acceptable as a specification, is redundant. We could make the definition more compact by combining the two clauses; sharing the identical conditions and splitting those which are non-identical as a disjunction. This yields:

$$
\begin{array}{l}
diagnosis(P,S,H,D) \leftarrow \\
\quad patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge \\
\quad (\, covered(S,H,D) \vee \\
\quad\;\; (\, revise(P,S,H,Hr) \wedge \\
\quad\;\;\;\; diagnosis(P,S,Hr,D))\,)
\end{array}
$$

The transformer tool provides a suite of transformations (including the one just described) which allow experienced logic programmers quickly to apply some of the more common manipulations. Knowing when it is best to apply these transformations, and in what order, requires a high level of expertise so it is prudent to confine them to a separate tool. At present, the repertoire of transformations is limited to common methods such as fold-unfold combination but this could be extended to include more sophisticated varieties which make use of information from the techniques editor (see [1] and [2] for more extensive discussions of this issue).

5 Utilising Standard Notations

The recursion tool attempts to make part of the specification process more accessible by emphasising part of the structure of a definition, independently of domain of application. However, there exist common forms of description, already used informally within domains, which are well suited to being interpreted as specifications in logic. In business process modelling it is often convenient to describe a process as a collection of sequential interacting sub-processes, each of which may depend on particular sources of information in order to be controlled. Separately from the information necessary to control a process, we may want to define the events which we would expect to observe as each subprocess carries through. Suppose, for example, that we want to define a simple patient processing system which, given a queue of patients, chooses a patient from the queue then either diagnoses that patient and recommends a treatment or refers him/her for further tests, continuing the processing on the remaining patients until the queue is empty. The recursive structure of this process can be drawn as the "box and arrow" diagram shown in Figure 2, in which sub-processes appear as boxes in sequence determined by the large arrows. The information sources appear as ellipses, connected to the process which either requires or supplies that information.

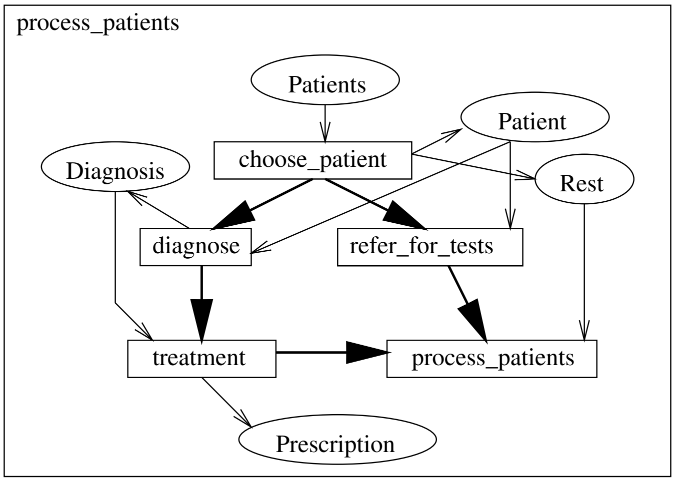

Figure 2. Process diagram for our running example

The process tool allows sequences of observable event definitions to be attached to sub-processes or sequence links. For example, we might want to attach to the end of the treatment sub-process the event that a prescription has been recommended to a patient, or record after the referral sub-process the event that a patient has been referred. Given these event definitions and the diagram above, the system can automatically generate the definitions shown below, expressed as definite clause grammars (with the $\Rightarrow$ symbol as the grammar operator and terminal symbols enclosed within square brackets, following the normal Prolog convention).

$$
\begin{array}{l}
process\_patients(Patients) \Rightarrow \\
\quad choose\_patient(Patients, Patient, Rest) \wedge \\
\quad diagnose(Patient, Diagnosis) \wedge \\
\quad treatment(Diagnosis, Prescription) \wedge \\
\quad [recommendation(Patient, Prescription)] \wedge \\
\quad process\_patients(Rest) \\
process\_patients(Patients) \Rightarrow \\
\quad choose\_patient(Patients, Patient, Rest) \wedge \\
\quad refer\_for\_tests(Patient) \wedge \\
\quad [referred(Patient)] \wedge \\
\quad process\_patients(Rest) \\
process\_patients([\,]) \Rightarrow [\,]
\end{array}
$$

We can now connect the diagnostic component (via the DCG $diagnose$ subgoal) to the predicate for $diagnosis$ which we developed with the recursion tool in Section 3, noting the observable event that a $Patient$ has been given a $Diagnosis$.

$$\begin{array}{l} diagnose(Patient, Diagnosis) \Rightarrow \\ \quad \{diagnosis(Patient, [], [], Diagnosis)\} \wedge \\ \quad [diagnosed(Patient, Diagnosis)] \end{array}$$

6 Sketching Window Interfaces

Since the interface to a software package is normally a key part of its operation, it is important to include some means of prototyping interfaces within the LSS toolset. Rapid prototyping of interfaces via graphical "sketches" of the windows and dialogue objects is a well known commercial method (some logic programming vendors already sell basic systems for this purpose) but the LSS version benefits from interacting with the other tools. Interface design begins by drawing, on a graphical pallette, a rough outline of the objects which should be contained in the window. The intention at this point is simply to state, rapidly, what should be there. Thus, each object is intentionally drawn quite crudely to preserve the feeling of a sketch. Figure 3 shows a sketch of a simple interface for our running example. We have drawn a window labelled $patient\_dialogue$ containing a text field with the label $patients$; an edit field to its right; a second text field with the label $treatments$; a scrolling text field to its right; and two buttons labelled ok and $cancel$. The link, labelled with a 'd', from the $cancel$ button to the top left corner of the window denotes that a click on this button should delete the window. The link labelled 'u' from the ok button to the scrolling text field indicate that clicking on this button should update the field in some way.

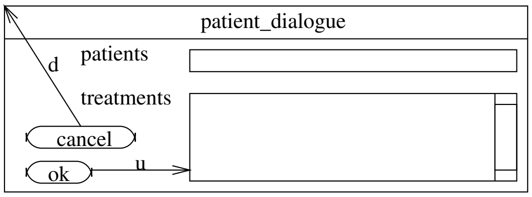

Figure 3. Window diagram for our running example

Once the graphical sketch is completed the diagram is translated automatically to a window definition identical to that which would appear in the final application, except that the system intercepts clicks on the window and uses these to prompt the user for changes to its format or that of any of its components. It is also possible to switch the system into simulation mode, at which point clicks on window components approximate to the behaviours sketched on the storyboard in Figure 3. This is staightforward for creation or deletion (for example, if the $cancel$ button is clicked the window disappears) but is less simple for updates between objects, since these do not completely specify how an object should be updated. To allow update behaviours to be simulated, the system prompts the designer to update the item by hand. This does not contribute to the formal specification directly but it gives designers a feel for the type of interaction which they might achieve. The completed window can be translated automatically into the system calls required to generate it (these are, of course, specific to the windowing system being used). The procedures for creation or deletion operations (like the deletion of the window by the $cancel$ button) can also be generated automatically.

For update operations we can only generate a partial specification but this can be produced in a form suitable for further refinement by the techniques tool of Section 2. For example, the update operation from the ok button of our sketch would be translated[2] as:

$$\begin{array}{l} click\_program(patient\_dialogue, p_1) \leftarrow \\ \quad \mathcal{U}(\{\}, Text) \wedge \\ \quad add\_window\_item(patient\_dialogue, p_2, Text) \end{array}$$

where $click\_program$ is the name of the procedure initiated by clicking the ok button; p_1 is the system identifier for the ok button; p_2 is the system identifier for the scrolling text field updated by the ok button; and add_window_item adds/adapts the scrolling text field. The condition $\mathcal{U}(\{\}, Text)$ is a "gap" in the specification (denoting that we need a value for the new $Text$ contents) which can be recognised by the techniques tool.

7 Viewing Macro-structure

When concentrating on the details of predicate definition it is easy to lose track of the general shape of the larger specification. One way of getting a better general picture of how Horn-clause definitions fit together is to link each predicate to each other (non-system) predicate which appears as one of its subgoals. This simple principle is used in the overview tool to allow the structure of specifications to be viewed graphically and extended (by introducing and linking new predicate definitions). Recall that Section 2 mentions that the techniques tool allows typical properties to be associated with partial specifications from its library. The overview tool provides a convenient interface for viewing these properties, since it is easy to propagate the properties associated with any given predicate to those which refer to it. This enables us to see not only what predicate definitions have been brought together for a given part of a specification but also the accumulation of properties which we might assume for these predicates. This is important because each property is assumed, not guaranteed. It is possible that in altering parts of the specification we might, legitimately, invalidate some of these properties and the overview mechanism provides a simple way of reviewing them. Whether or not a given property holds of a predicate is left to human judgement - although we are aware that automatic proofs of certain properties can be performed, our interest is in design and validation rather than verification.

8 Justifying Design Decisions

One of the main reasons for failure of communication in distributed specification is that justifications for design decisions are not recorded along with specifications. It is too easy then to assume that the specification is the only right one for the problem, rather than only one of many possible "right" specifications, and to forget what were the major factors in choosing it. For example, in the patient processing specification of Section 5 there is an implicit assumption that patients who are referred for treatment never come back into the queue for diagnosis. This isn't the only possible decision (perhaps they could enter another queue for testing) and it might be illuminating to have a thumbnail sketch of why other options were ignored. Mapping out the entire argument in justifying any design decision is impossible, since it may involve high-level goals which are difficult to represent in any useful formal way. However, coarse-grain argumentation

[2] We have removed some mundane system-specific information to simplify our description - the actual clause is a little more complicated

structures are commonly used in the requirements engineering community to record part of the justification for design decisions. This is the approach taken in the argument tool.

The model of argumentation used is based on that of [5], which was heavily influenced by earlier work on issue based argumentation. A semantic network is built in which nodes from a small selection of types (*e.g.* issues, positions and arguments) are related by typed links (*e.g.* a position may respond to an issue). Nodes in the argumentation network can refer to groups of predicates in the specification, thus allowing parts of the specification to be linked to contextual argumentation networks. Furthermore, each group of predicates is normally produced as the output of one of the specification tools, which my have used the output of yet another tool as the basis of its work. This allows the references from the argumentation structure to be used as "jumping off points" to following chains of development of the specification. For example, the definition of $diagnosis$ starts in Section 3 with the recursion tool (producing a basic recursive structure) and this is then elaborated using the techniques tool before being transformed (in Section 4) using the transformer tool. The argument tool allows us to follow through this chain of development, viewing the different specification styles on the way, and to link any of the stages to argumentation networks justifying key design decisions. Although the argumentation network was originally intended as a precursor to specification, we have found in practice that it is often most useful for post hoc rationalisation of design decisions, once "the dust has settled" from the flurry of initial design.

9 Conclusions

The artificial intelligence and formal methods communities have been quick to develop sophisticated specification languages but much slower to come to terms with the human aspects of how these may be used. The LSS toolkit tries to contain this problem by offering a variety of mechanisms for constructing specifications, each targeted on a small part of the task. This contrasts markedly with larger scale ontological work which attempts to describe common forms of problem solving method, inspired by observation of activity in particular domains (KADS for example [9]). Although useful in mapping out such domains, these exercises tend to overstretch human capacity to maintain a common ontology. It is easy for two people to have widely differing interpretations of the same ontological symbol and difficult to unpick this difference in a formal language because the ontologies of the higher layers of description are only loosely related to the ontologies of lower layers. LSS always links the external views of specification provided by each tool directly to the shared specification language (Horn clauses). This does not remove the need for a shared ontology, since the specification provided by any tool must be understood by whoever develops it further with another tool. However the need for a shared ontology is restricted to the interfaces between tools, rather than dispersed across the entire specification. This, we claim, should make it easier in a distributed environment, to avoid requiring everyone participating in the specification to have a uniform understanding of all the components. The LSS approach works best when styles of description which are established in domains of application can be mapped to manipulations of the underlying formal language. On the small scale of each individual tool this turns out to be surprisingly easy to do. The price we pay for this flexibility is that special tools are needed to manage the specification in the large. However, even here the complexity of the problem can be contained by specialising tools to well defined areas of organisation, such as grouping predicates or associating contextual arguments with specifications. This move from all-encompassing methodologies to highly partitioned specification may turn out not only to be expedient but inevitable as our style of work becomes more distributed.

ACKNOWLEDGEMENTS

This work is supported by the author's EPSRC advanced IT fellowship and by the DTI/EPSRC Safety Critical Systems initiative.

REFERENCES

[1] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental, 'Applying Prolog Programming Techniques', *International Journal of Human-Computer Studies*, **41**(3), 329–350, (September 1994). Also as Research Paper 641, Dept of Artificial Intelligence, University of Edinburgh.

[2] N.E. Fuchs and M.P. Fromherz, 'Schema-based transformations of logic programs', in *Logic Program Synthesis and Transformation, Workshops in Computing (Proceedings LOPSTR-91*, eds., T.P. Clement and K. Lau, Springer Verlag, (1992).

[3] T.S. Gegg-Harrison, 'Basic prolog schemata', Technical Report CS-1989-20, Department of Computer Science, Duke University, (September 1989).

[4] M. Kirschenbaum, A. Lakhotia, and L.S. Sterling, 'Skeletons and techniques for prolog programming', Tr 89-170, Case Western Reserve University, (1989).

[5] B. Ramesh and Luqi, 'Process knowledge based rapid prototyping for requirements engineering', in *Proceedings of the IEEE Symposium on Requirements Engineering*, pp. 248–255, San Diego, California, (1993). IEEE Computer Society Press.

[6] 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).

[7] 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.

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

[9] 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.