An Empirical Study of the LSS Specification Toolkit in Use

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

Abstract

The LSS (Lightweight Specification System) toolkit assists in the development of logic programs, using a variety of high level specification methods. Many other high level specification systems impose a single, uniform view of how specification should proceed. In practice, there is normally no single understanding of how to describe specifications - there are instead a variety of different forms of description which have evolved from the work practices of various domains. Any attempt to disturb these work practices in a radical way will, naturally, meet with resistance unless those who must be educated in new methods can see clearly that they will benefit (soon) from their efforts. LSS addresses this problem by providing a collection of comparatively simple independent tools, each of which is directed at a particular community of users who might reasonably be expected to adjust to the tool without excessive effort. In this sense, LSS is lightweight - it is intended to be easy to pick up. Communication between LSS tools is achieved by using Horn Clause logic as a standard language, although users of some of the tools are buffered from the logical details by interfaces targeted at the appropriate group of users. This allows the products of specification from some of the tools to be used as the basis for more detailed specification (perhaps by other people) using other tools. This paper summarises the current LSS system and describes the results of an experiment in applying it to a substantial software engineering task : the specification of one of its own tools.

1 Choice of Empirical Test

Ideally, every specification system should be evaluated against at least two fundamental criteria: its general effectiveness in tackling the classes of problems which it will encounter and its specific effectiveness in the hands of those who are intended to use it. Work on LSS began only recently so our experiment addresses the former criterion - we are have yet to conduct empirical tests on target groups of users but shall argue that the system is likely to fare well when subjected to this. Even confining ourselves to general effectiveness, there are few published examples of high-level formal specification systems being applied to

problems which would be recognised as realistic by practising programmers. The application described in this paper is realistic for the following reasons:

- The target program is a medium sized logic program, using approximately 850 lines of code in the definition of 85 predicates. It is not a "toy" application - indeed it defines one of the key tools in LSS itself.
- The size of predicate definitions varies from 2 lines to 68 lines; the number of arguments per predicate varies from 0 to 8; and a variety of standard definition styles are employed (including various styles of recursive definition as well as the occasional use of "dirtier" styles such as failure driven iteration). The application is not artificially skewed towards a particular type of algorithm or style of development.
- Approximately a quarter of the lines of code exist solely to control the window-based interface to the application. We do not have the luxury of ignoring the dynamics of the interface to the application.
- Neither the application nor the LSS toolkit were adapted during the experiment, even when comparatively simple extensions to the knowledge bases of some of the tools could have made the application easier to handle. No "fine tuning" was done to make the system look good.

Our empirical test of the current LSS system is straightforward - we chose one of LSS's own tools and attempted to regenerate the code for it using LSS. However, the analysis of our results is not so easy, since we are assessing the effectiveness of creative design rather than measuring a carefully controlled process. In Section 4 we shall summarise the results of our analysis, most of which will be qualitative but allowing rough estimates of the productivity of the system. To understand the task performed in the experiment it is necessary to have a flavour of LSS in general and the tool which was reconstructed (the techniques editor) in particular. This is provided in Section 2. An illustrative example of a specification sequence using several tools on a part of the reconstruction appears in Section 3. Readers interested in a broader discussion of the motivation for LSS are referred to [1].

2 Overview of LSS

LSS currently consists of eight tools, each of which is directed at a different form of specification. These tools are shown as rectangular boxes in the central segment of Figure 1. To assist users in performing the task specific to a tool, each harnesses knowledge about the task, as summarised on the left of Figure 1. All of the tools possess tool-specific data structures which are used to store and retrieve its specifications. These are summarised on the right of Figure 1. Some of the tools can also read or write information in the format recognised by the other tools. This is useful for refinement and for assembly. Refinement (as we mean it in this paper) takes place when the partial specification produced by one tool is picked up and extended by another tool (for example, a techniques history produced by the recursion tool may later be loaded into the techniques editor and extended). Assembly occurs when a number of different data files are brought together (for instance, the overview tool connects the predicate definitions from other tools to form a larger specification).

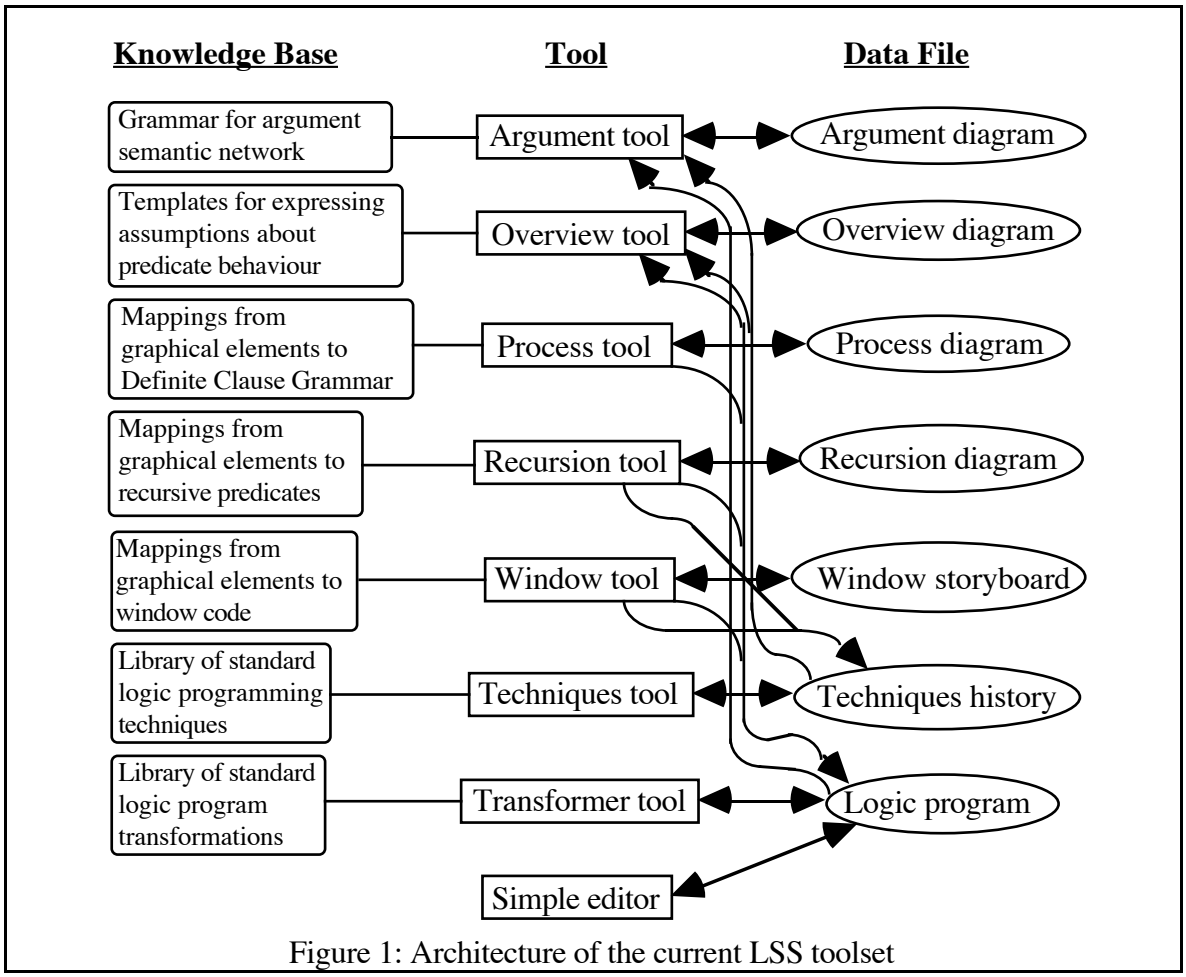


Figure 1: Architecture of the current LSS toolset

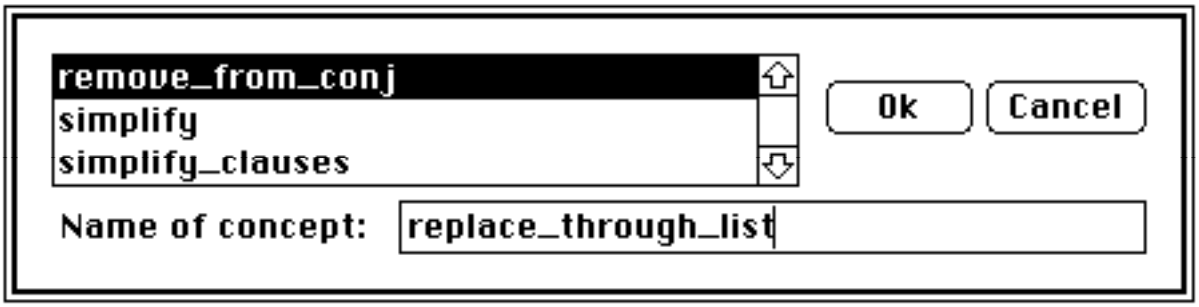
There are many potential connections between tools - hence the dense network of directed links in Figure 1 - but there is no obligation to use them all. On the contrary, it is to be expected that on any given project only a subset of the available tools would be useful, with the choice of which to use being determined by the backgrounds of the people involved and the overall objectives of the project. The table below summarises the intended community of users for each tool. A more detailed discussion of the potential user communities for some of the tools appears in [1].

Tool	Users
Argument tool :	People familiar with semantic networks for expressing the general structure of arguments - a method used by requirements analysts as a means of expressing pre-specification decisions.
Overview tool :	People who are familiar with logic programming and who want to highlight the macro-structure of a medium sized specification, using the subgoal structure as a map of the specification.
Process tool :	Process modellers who want to describe non-deterministic, sequential, interacting processes but who are more comfortable with a "box and arrow" diagram than with a logic program.
Recursion tool :	Those who want to view problem decomposition as recursion; are not logic programmers; but are able to specify the inputs to their specification using boxes and links.
Window tool :	For the early stages in interface design where users often want to sketch out the main components of their interface and the interactions between them, as a primer for detailed design.
Techniques tool :	Logic programmers (or would-be logic programmers) who want to describe their programs according to standard methods of construction for speed and/or for clarity of description to others.
Transformer tool :	Experienced logic programmers who want easy access to some of the standard methods for transforming logic programs, as a means of refining an existing specification.
Simple editor :	A last resort for people who want a basic, window based ASCII editor.

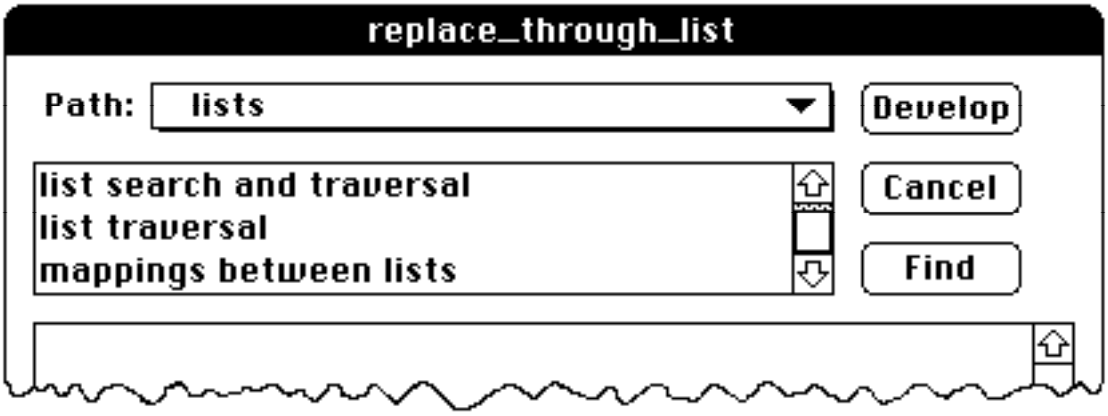
The techniques tool plays a special role in this paper because, as well as being one of the tools used, it is also the subject of specification. In the rest of this section we work through an example of the techniques editor in operation, using a small specification problem to

demonstrate its main functionality. In Section 3 we shall demonstrate how one of the components of the techniques editor - the filemanager - was reconstructed using LSS.

The techniques editor assists in constructing predicates in a structured style, beginning with a skeletal outline of the main flow of control and then extending this in a variety of ways. The editor dialogue commences by specifying the name of the predicate to be constructed, which in the example below is `replace_through_list` a predicate used in the system to substitute all occurrences of a given term within a list of ground clauses with a stipulated new term, giving a new list of clauses. The scrolling menu shows all the predicate names which have already been defined so that these may be retrieved if a user wishes to modify an existing definition. In the example, we wish to add a new definition.

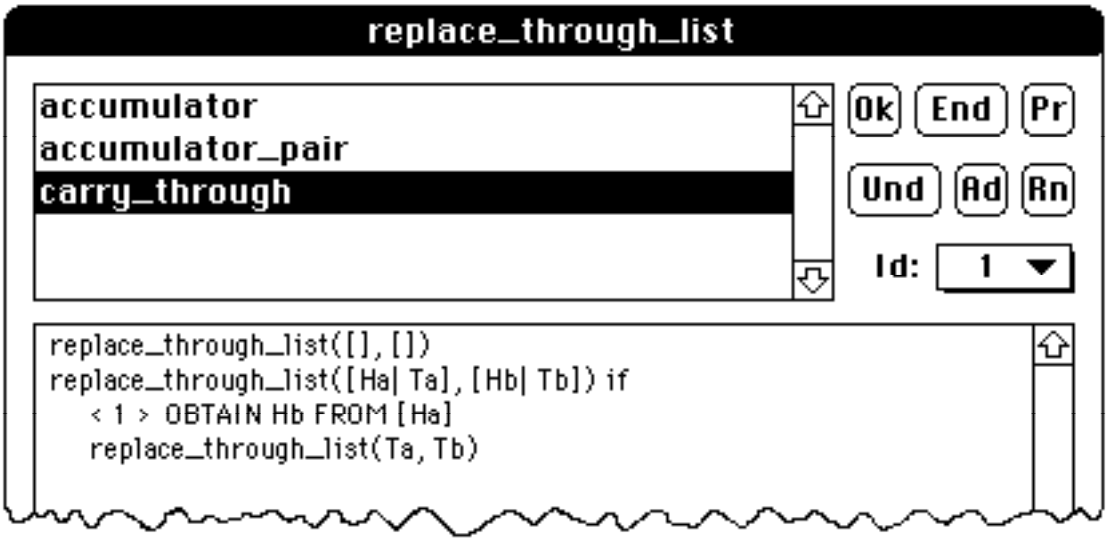


We are then shown a window, labelled with the predicate name, which allows us to navigate through a hierarchy of definitions of common skeletal flows of control. The popup menu at the top of the window shows our path down through the hierarchy while the lower scrolling menu shows the choice of skeletons immediately below the current position. The ragged edge on the bottom of the window signifies that the uninteresting portion below the edge has been "torn off" to save space.

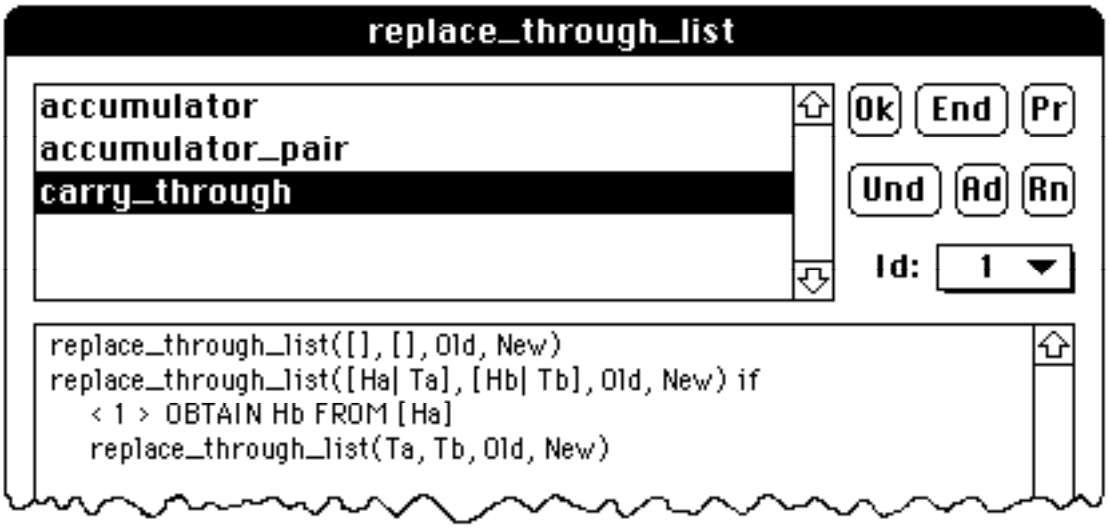


After searching through the hierarchy we find a skeletal definition which fits our purpose for this predicate. This is a skeleton for applying a mapping from all the elements of one list to

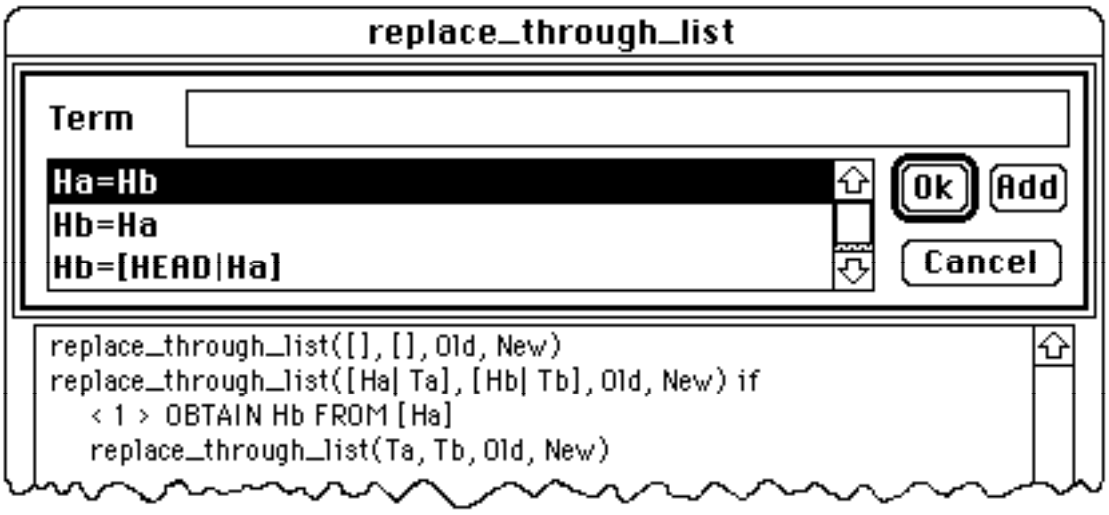
produce the elements of a second list. It is possible to use existing skeletal definitions as templates to retrieve completed predicate definitions from source files, using a simple matching algorithm to locate the predicates in a file which structurally resemble the chosen skeletal definition. This case based style of definition can be an effective way of reusing existing definitions (it is accessed via the 'Find' button). However, we are content in this case to use the skeleton given and choose it by clicking the 'Develop' button . This gives us the following window, displaying the skeletal technique; a menu of basic operations for extending it; buttons for performing variety of standard tasks; and a popup menu which is used to select the numbered gaps in the specification for instantiation. The line labelled <1> denotes a gap which needs to be filled to complete the skeleton - which in this case is the mapping relation between each element of the first list to the corresponding element of the second list.



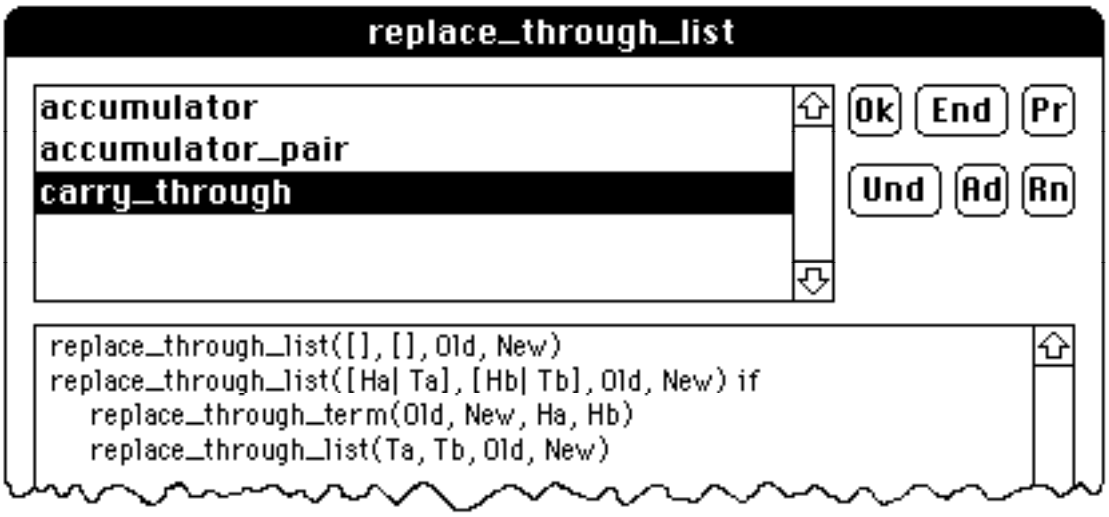
The predicate we want to define must replace all the existing occurrences of some term in the first list with a given new term. To do this, we must be able to carry both the old and new terms through the recursive definition. The extension method called 'carry_through' (from the menu above) allows us to achieve this by adding two new arguments, which are automatically threaded through the recursive definitions as shown below.



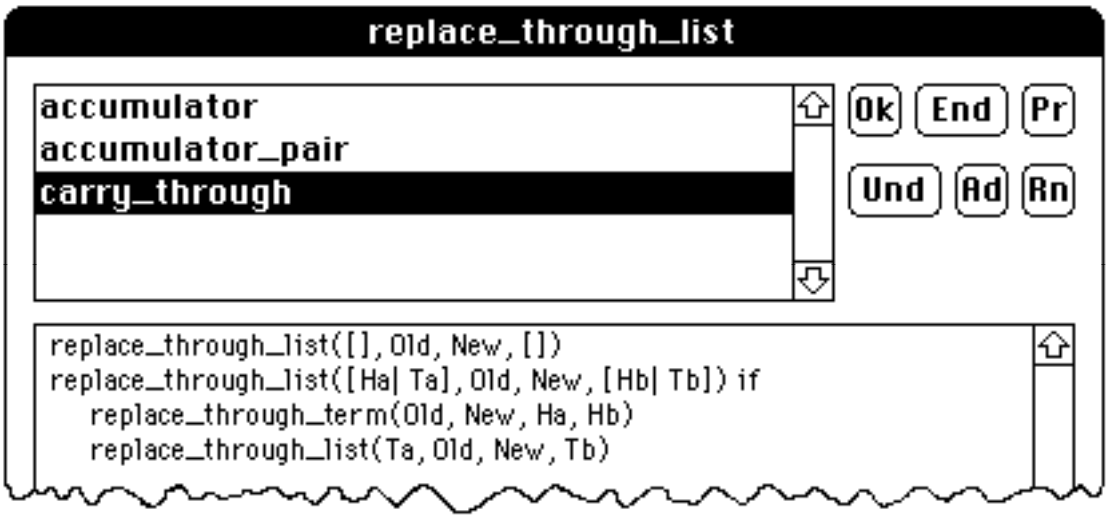
We can now instantiate the gap in the definition by clicking on its identifier in the popup menu. We are given the following prompt for instantiation. The editor generates a menu of likely methods of instantiation but, in this case, we want to perform the mapping via a new subgoal: `replace_through_term(Old,New, Ha, Hb)`, which will replace instances of `Old` with `New` in the element `Ha` to obtain `Hb`. This new subgoal is entered in the field provided.



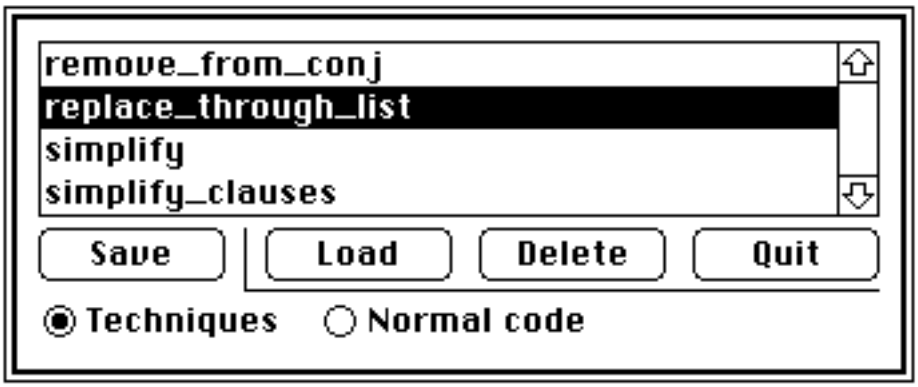
The definition is now complete but, for relations like this one which are being thought of as functions, it is standard practice to put the "output" arguments last in the argument sequence. The editor allows users to change argument positions, automatically maintaining consistency of position within the clause, so it is easy to make this alteration.



The final definition, with revised argument positions, appears below. We can retain it by clicking the 'Ok' button or forget about it by clicking the 'End' button. Had we made a bad decision at some earlier point in definition, we could step back to just before the problem occurred by clicking the 'Undo' button.



Having constructed a collection of predicates in this way, we can save them to files via the filemanager shown below. The definitions can be saved either as normal code or with their full editing histories, so that they can be reused later within the techniques editor or "replayed' via the argument tool (as we shall explain later). The filemanager also allows definitions to be loaded and deleted.



This completes our description of the techniques editor - the target for our experimental reconstruction. In the next section we select a small part of the editor and give a detailed example of its reconstruction. We then return (in Section 4) to the wider results of this empirical test.

3 An Example of Specification Using Several LSS Tools

It would be wasteful of space to plough through the entire process of reconstruction of the techniques editor, but it is worth working through a typical example of what was done. For this we have chosen the file manager, since it is one of the smaller parts of the system but is representative of the larger task. Since the filemanager uses a window interface it is convenient to begin the definition by sketching this using the window tool. Interface design begins by drawing a rough graphical outline of the objects which should be contained in the window, as shown in Figure 2. The intention at this point is not to describe the details of size and positioning of the window and its components but simply to state, rapidly, what should be there. Thus, each object is intentionally drawn quite crudely to preserve the feeling that we are not committing to detail. Figure 2 shows the main graphical pane (which we call a "storyboard" after the similar paper-based method of interface specification) with the palette of graphical objects on its left side. We have sketched the outline of a window containing a menu (top left); four buttons (labelled 'quit','save','load' and 'delete'), and two radio buttons (bottom left). The link, labelled with a 'd', from the 'quit' button to the top left corner of the window denotes that a click on this button should delete the window. The two links labelled 'u' from the 'load' and 'delete' buttons to the menu indicate that clicking on either of these buttons should update the menu in some way.

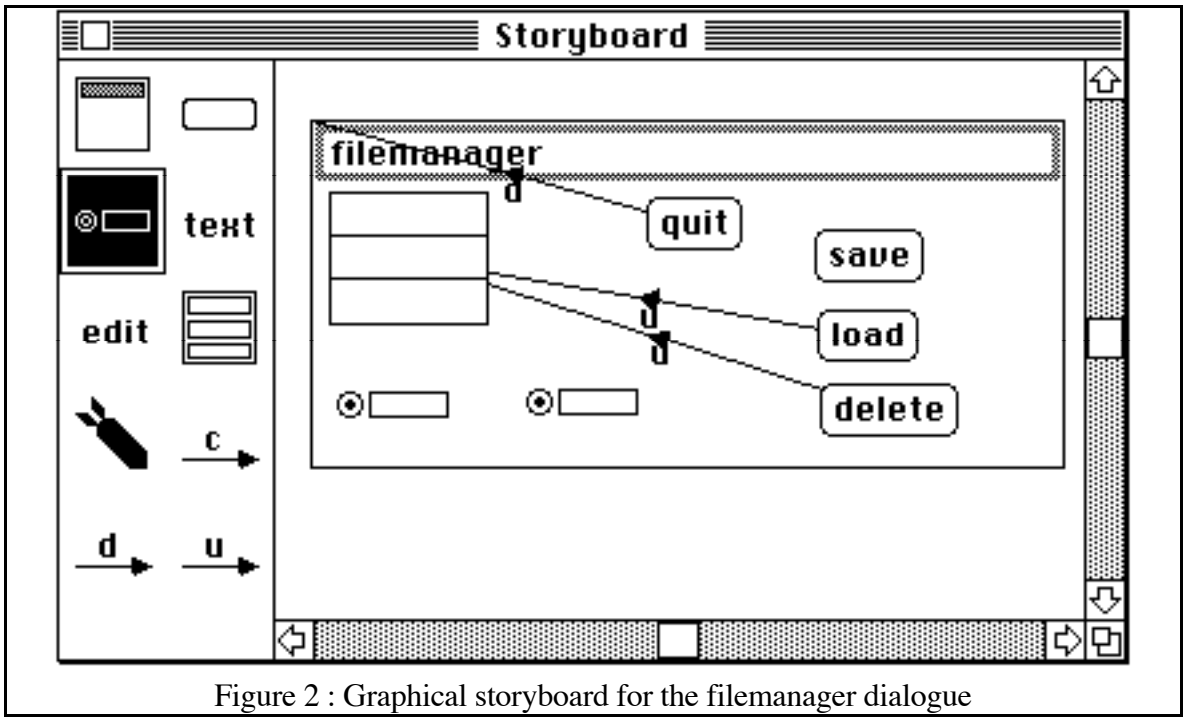


Figure 2 : Graphical storyboard for the filemanager dialogue

Once the graphical sketch is completed the diagram from Figure 2 is translated automatically to a window definition identical to that which would appear in the final application, except that the storyboarding 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. The top part of Figure 3 shows the closing stages of adaptation of the filemanager window. The user has clicked on the scrolling menu and has been offered a modification window (partially obscured by the filemanager window in the figure) which allows the user to alter the parameters for menu items - for example it would be possible to change the menu from a scrolling menu to a popup menu or to change its location or size. 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 2. This is staightforward to achieve for creation or deletion (for example, if the 'quit' 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 calls up the appropriate modification window and the user can update the item by hand. This does not contribute to the formal specification directly (which would be very hard indeed to arrange) but it gives users a feel for the type of interaction which they might achieve. The completed

window can be translated automatically into the graphical system calls required to generate it, as shown in the lower part of Figure 3.

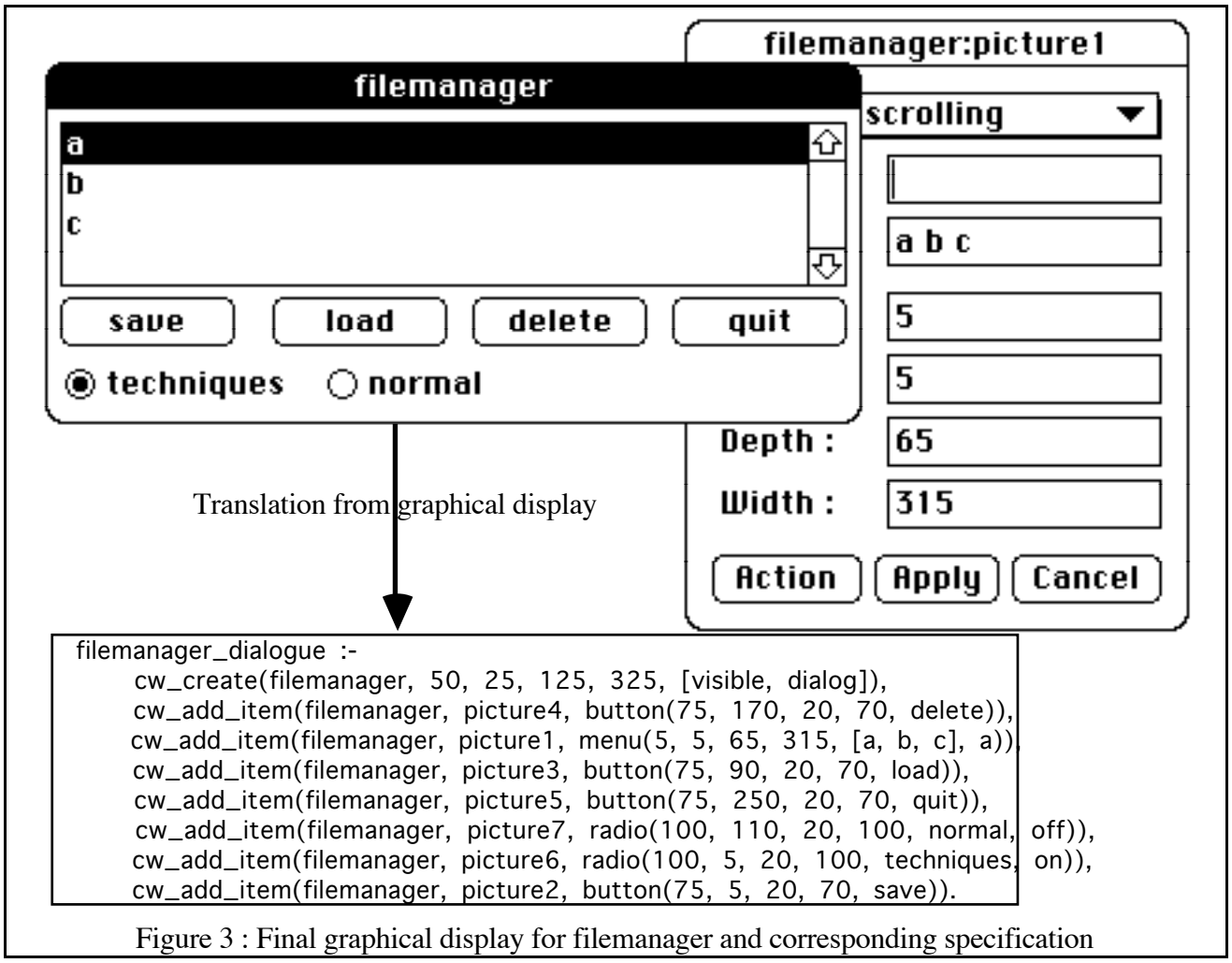


Figure 3 : Final graphical display for filemanager and corresponding specification

The system can also generate automatically partial specifications for the predicates corresponding to the delete, and update links on the storyboard. These are shown in Figure 4, for the 'load' (`picture3`),'delete' (`picture4`)and 'quit' (`picture5`)buttons respectively. For the 'load' and 'delete' procedures we have introduced `'$Update'` subgoals which represent gaps in the specification - we know that there is a need to update the menu items (`Items1`) and preselected menuitem (`Preselected1`)in some way but we have yet to define how this is to be done.

```
filemanager(filemanager, picture3) :-
    '$Update'([], Items1),
    '$Update'([], Preselected1),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, Items1, Preselected1)).
filemanager(filemanager, picture4) :-
    '$Update'([], Items1),
    '$Update'([], Preselected1),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, Items1, Preselected1)).
filemanager(filemanager, picture5) :-
    wkill(filemanager).
```

Figure 4 : Partial specification of button behaviours generated from the storyboard

The techniques editor can be used to fill these gaps, using the mechanism demonstrated in Section 2. The result of this is shown in Figure 5, with the instantiations performed by the techniques editor shown in boldface. Notice that for the 'picture3'and 'picture4'buttons we have duplicated the original clauses and set different conditions for each, to provide a different behaviour when there exists a non-empty set of existing clause definitions (as defined by 'pclauses') than when there is none.

```
filemanager(filemanager, picture3) :-
    setof(P, C^pclauses(P, C), [H| T]),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, [H| T], [H])).
filemanager(filemanager, picture3) :-
    \+ pclauses(_, _),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, [], [])).
filemanager(filemanager, picture4) :-
    setof(P, C^pclauses(P, C), [H| T]),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, [H| T], [H])).
filemanager(filemanager, picture4) :-
    \+ pclauses(_, _),
    cw_add_item(filemanager, picture1, menu(5, 5, 65, 315, [], [])).
filemanager(filemanager, picture5) :-
    wkill(filemanager).
```

Figure 5 : Specification of button behaviours instantiated by the techniques editor

The code produced by the techniques editor in Figure 5 is approximately half that needed for the complete filemanager interface definitions. We still have to provide a definition for the behaviour of 'picture2'- the 'save' button which could not have its dynamic behaviour sketched on the storyboard because it effects no other window components. We also need extra subgoals to allow the selection of files for saving and loading predicates. While it would have been possible to add these within the techniques editor (by applying a more complex instantiation of the gaps in Figure 4) it felt unnatural to do so because the gaps are

intended to record the direct sources of instantiation for key variables rather than to allow arbitrarily complex sequences of subgoals to be stitched into the code. This final augmentation is delegated to the basic ASCII editor.

Constructing the filemanager interface in this way is quick but it leaves us with a trail of partially constructed programs from the storyboard tool, the techniques editor and the ASII editor. The argument tool allows us to review the connections between these and to set them in the context of more general explanations of our design decisions. The central element of the argument tool is a semantic network composed from a small set of predefined node and link types. In the current tool we have adopted the link and node types used by Ramesh & Luqi [2] but many other semantic network grammars have been used by others and the grammar of our system could be adapted to these if required. Rather than examining the argument tool in general let us look at how it would place in context a particular design decision for the filemanager example.

Although it is not apparent simply by looking at the predicate definitiions, one of the main influences on the structure of the filemanager was the decision to have a separate file management window in the first place. We could have made a different choice to obtain the same functionality (for example we could have attached similar operations directly to the techniques editor windows) but, on balance, it felt like a better design decision to parcel the operations together. What swayed the argument was that the same filemanager window seemed likely to be useful for other tools, although at the time it wasn't absolutely clear that their would be enough uniformity of operation to support this. This argument can be outlined in the argument tool using the semantic network shown in Figure 6. This network is similar in display to that shown by the argumentation tool itself but the use of colours, box shapes and larger fonts in the actual tool makes its diagrams difficult to patch directly into a paper, so we have replicated it by hand here. The nodes in the diagram correspond to stages in the argument: requirements (R), issues (I), positions (P), arguments (A), assumptions (As), decisions (D), constraints (C) and specification definitions (D). These are related by

directed links - for example the requirement that we may want to delete selectively generates the issue of how to make this easy to do.

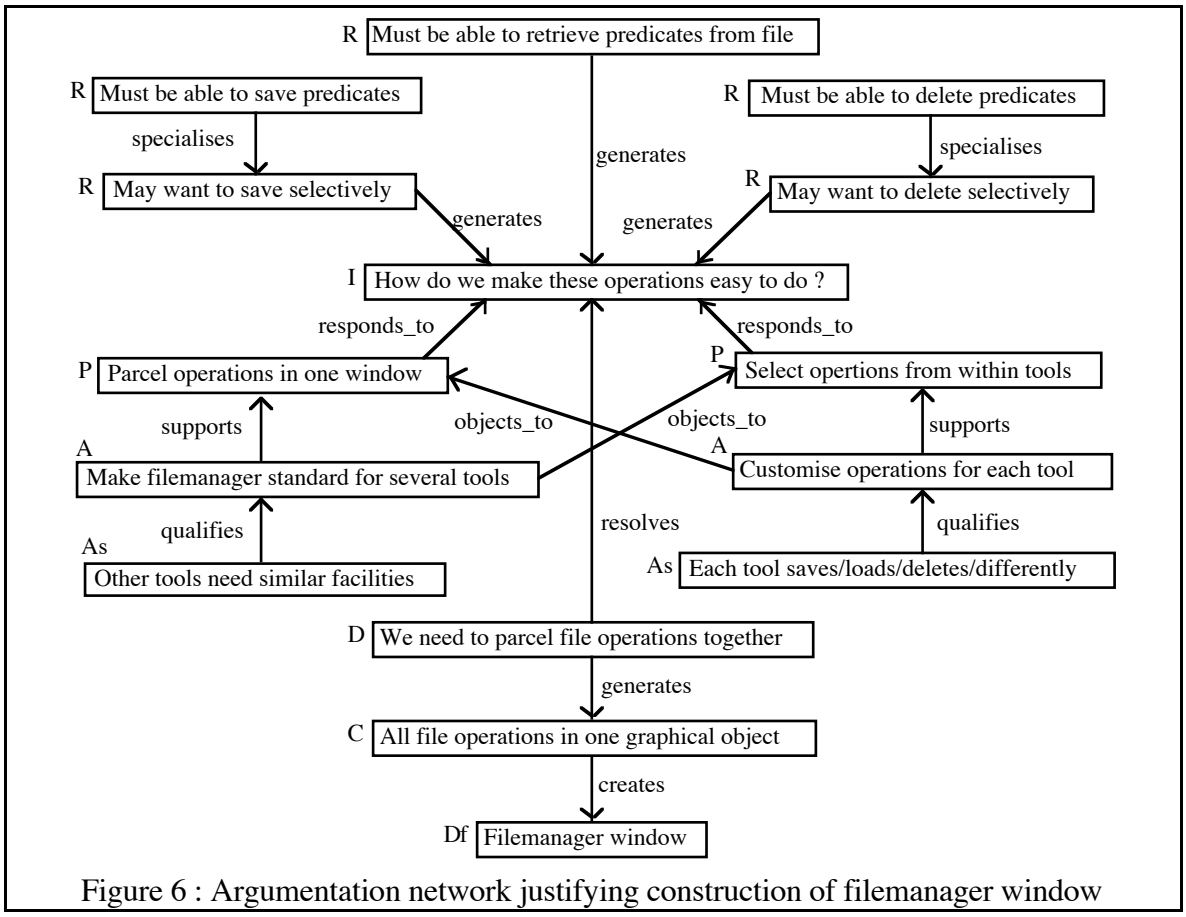


Figure 6 : Argumentation network justifying construction of filemanager window

The nodes of the argumentation network can be "opened" to view more detailed textual descriptions of the concept involved or to follow a pointer to the specification definitions related to it. Since each tool maintains a record of the sources of any of the predicates which it imports, and notes this in a header to any files which it creates for those predicates, this provides an entry point to the sequence of development. Figure 7 illustrates this process for our running example. Recall that we constructed the filemanager using (in sequence) the window tool, techniques editor and ASCII editor. The first two of these produce both logic programs and techniques histories (records of the sequence of editing steps employed). Thus, if we connect the final filemanager specification (logic program 3 in Figure 7) to the definition node of the argumentation diagram then we can unravel the stages of development back through the earlier technique histories and logic programs, using a techniques viewer when appropriate to provide a replay of techniques application. The idea of linking

specifications to argumentation networks has already been promoted by Ramesh & Luqi [2] but combining this with LSS's heterogeneous toolset and techniques editing operations permits a carefully controlled and (if required) a fine grained recapitulation of specification development.

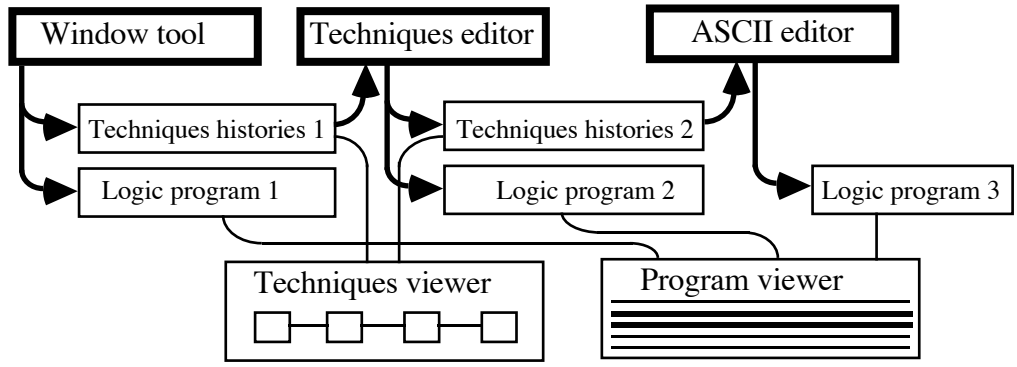


Figure 7 : Generating and viewing chains of specification development

4 Analysis of LSS Performance

Of the 85 predicates used in the techniques editor, 49 could be specified using the LSS tools. We count a predicate as having been successfully specified if it can be defined using any combination of the tools, excluding the ASCII editor, in a manner which corresponds to the intended pattern of use of the tool. This last condition is important because it is possible (with great effort) to construct any logic program with tools like the techniques editor, by "clever" manipulation of some of the editing operations. This temptation to push the tools beyond normal usage has been avoided as far as possible, although (as noted at the outset) we have as yet no experimental analysis of how users other than the author of LSS will want to operate the system. Of the predicates which we considered inappropriate for LSS, about a third were excluded because they were too simple to warrant the effort of applying a special purpose tool (predicates of two or three lines are easier to write by hand). The majority of those which we couldn't handle were difficult because they had no obvious, generic structure on which to base techniques editing construction; nor did they define the dynamics of window components (although many referenced information stored in the windowing system); nor did they permit the sort of stereotypic conceptualisation which would have suited one of the more specialised tools. In general, covering 49 out of 85 definitions seems an encouraging result since LSS was intended to be used to construct limited types of

particularly elegant specifications in a directed way. It is an unexpected bonus that it seems to be able to reconstruct substantial portions of "dirty" specifications which have been written in the heat of prototyping and with embarrassingly little attention to elegance of expression. Nevertheless, there were some unexpected lessons, which we summarise below.

4.1 Instantiating Window Behaviours

The gaps left in the specification by behavioural sketches which require updates to window components were difficult to fill using the current techniques editor. This is because much of the substance of these updates requires references to configurations of subgoals which are specific to the window management system being used (e.g. file accessing sequences; patterns of address for edit fields). The current techniques editor contains no skeletal or extension definitions for this type of task - nor is it clear that these would best be employed within the more generic techniques tool. There is scope for a specialised window procedure editor which operates in concert with the storyboard tool. A good candidate for the architecture of this tool might be the schema-based generation systems which we devised in our earlier ECO system [3].

4.2 Need for an Improved Transformation Tool

The transformation tool does not figure largely in this paper because it was used only occasionally in the reconstruction experiment. However it would have found much greater use had its library of transformations been more sophisticated. It turned out that one of the most common transformations was to "pack together" two or more clauses with a common sequence of subgoals into a single clause with the single common sequence and the other non-shared subgoals as disjuncts. This enables the earlier tools to construct predicate definitions without being concerned about replication of subgoals, in the knowledge that the definitions can be merged in an efficient manner later. An example of this (taken from the reconstruction) is in defining the predicate given below, which is used in an algorithm for matching clauses read from a file against a set of target clauses (this forms part of the case based mechanism mentioned briefly in Section 2). The techniques editor produces the

following specification for it which, although logically correct, replicates the part of the definition shown in boldface.

```
match_each_predicate(end_of_file, TargetClauses, [], null, null).
match_each_predicate(X, TargetClauses, FClauses, F, S) :-
   \+   (X=end_of_file),
   fun_and_arity_for_clause(X,   F,   Arity),
   read_rest_predicate(F,  Arity,  Rest,  Next),
   Arity>0,
   abstract_arg(TargetClauses, Arity, [X| Rest], F, N, FClauses, S).
match_each_predicate(X, TargetClauses, FClauses, PNm, S) :-
   \+   (X=end_of_file),
   fun_and_arity_for_clause(X,   F,   Arity),
   read_rest_predicate(F,  Arity,  Rest,  Y),
   match_each_predicate(Y, TargetClauses, FClauses, PNm, S).
```

The transformation tool can merge the two clauses containing replicated subgoals to yield the following more efficient (though probably less readable) definition

```
match_each_predicate(end_of_file, TargetClauses, [], null, null).
match_each_predicate(X, TargetClauses, FClauses, PNm, S) :-
   \+   (X=end_of_file),
   fun_and_arity_for_clause(X,   F,   Arity),
      read_rest_predicate(F,  Arity,  Rest,  Next),
   ( (Arity>0,
     abstract_arg(TargetClauses, Arity, [X| Rest], F, N, FClauses, S),PNm=F);
   match_each_predicate(Next, TargetClauses, FClauses, PNm, S) ).
```

Unfortunately this useful facility was not applicable to all of the cases where transformation could have improved the efficiency of definitions. This was partly because that particular tool is still under development and so lacks a comprehensive armoury of tranformation methods; partly because more thought needs to go into the control mechanisms for such tools; but also reflects the lethargy which has afflicted the logic programming community (with a few notable exceptions such as [4],[5]) in translating some of its elegant and powerful theroretical results into methods of value to software engineers.

4.3 Extending the Knowledge Base of the Techniques Editor

Although it was a pleasant surprise to find that the small collection of skeletal and extension definitions in the current editor could cope with a reasonable proportion of the real examples, there is still plenty of scope for extension of the repertoire. However, the urge to add complex methods which are highly effective in limited cases must be tempered with caution, since we do not want to create a massive library of methods which require hours of careful scrutiny to find the most appropriate. The current system uses les than 30 methods and, at a rough guess, the upper limit for browsing might be in the region of 100. The methods used

in the techniques editor merely encode generic patterns of structural development of the specification, rather than appealing to more ambitious notions of abstract plans of algorithm development. This, we feel, gives a greater chance that the range of available methods can be kept under control whilst still ensuring acceptable coberage.

4.4 Harnessing the Higher Level Tools

The process and recursion tools, which use high level graphical languages to describe partial specifications, were not used at all in the reconstruction experiment. It would have been pleasing to have involved the full range of tools in the experiment but it is not unduly worrying that these higher level tools did not find a role to play. Their main purpose is to make specific types of partial specification easy to describe. This makes them better suited for initial design than for reconstruction, when the full specification is already known and it feels more natural to concentrate on the mechanics of how it can be put together rather than on the ways in which it might be conceptualised.

5 Closing Comments

The LSS system occupies an unusual position in the research world because it attempts to draw together a number of different methods for dealing with formal specification without becoming enmeshed in the more esoteric aspects of the disciplines from which it draws inspiration. The general intention is to glean from logic programming (and the wider formal methods community) methods of specification which have a foundation in formal theory but which can be married to forms of use which appeal to software and requirements engineers. In practice, we have targeted a number of specific areas thus far:

- The use of standard methods of logic program construction as a means of standardising design of specifications. This is the basis of the techniques tool and draws on a variety of earlier theoretical work, including [6] and [7].
- Adapting transformation techniques to fit into appropriate niches in the lifecycle of specifications. The work here has been influenced by [4] and [8].

- Understanding "informal" argumentation structures in the context of formal specification. Here we have drawn on the existing work of [2] and others, placing it in the context of logic programming.
- Constructing graphical visualisation methods for specification methods which retain a direct mapping to the underlying specification language but which, by their form of communication, emphasise the key elements of the design. The inspiration for our early steps in this direction was from [9].
- Understanding established "informal" specification methods as manipulations of specifications in a computational logic. For this we have sought sources in application domains - most recently business process modelling.
- Testing the effectiveness of computational logics when applied to specification tasks of realistic complexity. This is the motivation for [3] and for the current paper.

The reconstruction experiment described in this paper encourages the belief that by careful selection and controlled application of hitherto theoretical methods we can achieve practical benifits without requiring unrealistically high overheads of training in computational logic. There is still long way to go, and the gulf between the theoretical and application communities remains wide. Experiments such as the one described here help bridge this gulf.

Acknowledgements

The research described in this paper is supported by an EPSRC IT Advanced Fellowship for the author and an EPSRC/DTI research project under the UK Safety-Critical Systems Initiative.

References

[1] D.Robertson, Lightweight Formal Methods, *Proceedings of the Monterey Workshop on Formal Methods: Software Architectures*, Monterey California, 1995.

[2] B.Ramesh & Luqi, Process Knowledge Based Rapid Prototyping for Requirements Engineering, *Proceedings of IEEE Symposium on Requirements Engineering*, San Diego, California, pp 248-255, IEEE computer Society Press 1993, ISBN 0-8186-3120-1

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

[4] N.Fuchs & M.Fromherz, Schema Based Transformations of Logic Programs, *Proceedings of the International Workshop on Logic Program Synthesis and Transformation*, Springer Verlag, 1992

[5] W.Vasconcelos & N.Fuchs, An Opportunistic Approach for Logic Program Analysis and Optimisation Using Enhanced Schema-Based Transformations, *Proceedings of the Fifth workshop on Logic Progrm Synthesis and Transformation*, Springer Verlag, 1995.

[6] L.Sterling & M.Kirschenbaum, Applying Techniques to Skeletons, Technical Report TR 91-179, Case Western Reserve University, 1991.

[7] A.Bowles, D.Robertson,W.Vasconcelos, M.Vargas-Vera & D.Bental, Applying Prolog Prolog Techniques, *International Journal of Man Machine Studies* 41(3), pp 329-350, 1994

[8] W.Vasconcelos, *Extracting, Organising, Designing and Reusing Prolog Programming Techniques*, PhD Thesis, Dept. of Artificial Intelligence, University of Edinburgh, 1995.

[9] G.Allwien & J.Barwise (eds), Working Papers on Diagrams and Logic, Indiana University Logic Group, report IULG-93-24, Visual Inference Laboratory, Indiana University, 1993.