Pragmatics in the Synthesis of Logic Programs[1]

Dave Robertson †, Jaume Agustí ‡

†Dept. of Artificial Intelligence,
University of Edinburgh,
80 South Bridge, Edinburgh
Email: dr@aisb.ed.ac.uk

‡IIIA (CSIC),
Campus UAB, 08193 Bellaterra,
Barcelona,
Email: agusti@iiia.csic.es

Abstract

Many of the systems which we, and those who have worked with us, have built were intended to make it easier for people with particular backgrounds to construct and understand logic programs. A major issue when designing this sort of system is pragmatics: from the many logically equivalent ways of describing a program we must identify styles of description which make particular tasks easier to support. This paper describes three ways in which we have attempted to understand the pragmatics of particular domains using well known methods from computational logic. These are: design using parameterisable components; synthesis by incremental addition of program slices; and meta-interpretation.

1 Why Domain Specific Systems are Needed

This paper concerns the use of generic forms of design of logic programs in domain-specific applications. Much of the logic programming literature has a different concern: to construct systems which are as general as possible, and preferably independent of domain. The reason why we are interested in focussed systems is that the problems we tackle normally involve engineers who have neither the time nor inclination to learn about logic. They tend to be interested in preserving the styles of description which they already know and trust, so our uses of logic programming must mesh with these. However, we do not want to make our systems so domain-specific that we must invent a new style of design each time we tackle a new problem. This creates a tension between the desire for generic design methods and the need for lifecycles which are tailored to particular domains. In what follows we show how we have used generic methods from logic programming to underpin domain-specific design systems. In each of the main sections we begin by summarising the method; then exemplify its use in applied projects; finally explaining those factors which limited the size of our applications. To save space, we have anchored the discussion in simplified formal descriptions of the methods - used as an aid to explanation rather than a description of the full method.

Some general themes recur in the different methods. We often use a high level language targeted at a particular engineering group. This appears as the problem description (Φ) in Section 2 and as the parameterisation conditions (Ψ) of Section 3. We frequently supply a library of standard design patterns, such as the parameterisable components of Section 2 or the program slices of Section 3. Sometimes we can split the design lifecycle into different stages which are highly modular but which connect closely for later analysis, as in the meta-interpretation of Section 4.

[1]Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

2 Parameterisable Components

We have used parameterisable components in situations where it was possible to predict standard patterns of program description on a fairly large scale (*i.e.* where many of the patterns comprise more than one predicate) and where variation in use of these patterns can be controlled via parameters which can be explained to the engineers in our target domain. Although generic notions of parameterisability exist within logic programming (*e.g.* [10]) and functional programming (*e.g.* [4]) we use more specific forms of parameterisation adequate for our applications. Our method of designing such systems is as follows:

- We identify basic features of the domain which engineers use in order to select and instantiate standard patterns. Using an interface tailored to the task, statements in this restricted language are supplied by those engineers. We call this a problem description and represent it below using the symbol, Φ.

- We build a library of parameterisable components, each of the form:

  $$component(N, C, D, P)$$

  where N is the name of the top-level predicate defined by the component and C is a set of conditions which must all be satisfied from the problem description in order to instantiate the component. When instantiated, D is a set of clauses defining the program for N, and P is the set of subsidiary predicates which we require in order to supply us with an executable component. If the component is self-contained then P is empty.

- We construct a system which selects and instantiates components according to the problem description supplied by engineers. We describe this, in its simplest form, using the predicate $parameterise(P, \Phi, D)$ which denotes that a program consisting of the set of clauses D for satisfying the predicates named in P can be constructed through parameterisation using the problem description, Φ. The predicate $Select(P, N, P_r)$ selects from P the predicate named N to construct, leaving remaining names P_r. The predicate $Best(S, K)$ chooses the most appropriate component, K, from the set, S of candidates.

$$
\begin{array}{l}
parameterise(\{\}, \Phi, \{\}) \\
parameterise(P, \Phi, D \cup D_r) \ \leftarrow \\
\quad Select(P, N, P_r) \ \wedge \\
\quad Best(\{(C', D', P')|(component(N, C, D, P'') \ \wedge \ \forall G. G \in C \ \rightarrow \ \Phi \vdash G)\}, (C, D, P_n)) \ \wedge \\
\quad parameterise(P_r \cup P_n, \Phi, D_r)
\end{array}
$$

Our earliest use of this form of design was in constructing animal population models from problem descriptions couched in the terminology used by ecologists (see [9]). An example of a component (simplified from [9]) is determining whether two animals (A and B) are considered to be in proximity to each other at time point T. There are various standard ways of defining this but one way is by assuming that the physical area is divided into grid squares and each animal is assigned a square. By satisfying $grid(G)$ and $locations(L)$ we find out what is the appropriate grid and where the animals are initially located. This information, when added to the definition of proximity, gives us the program for the component. To complete the definition we need to know how the dynamics of change in location works so *located_at* is in the set of new predicates.

$$component(in\_proximity, \left\{ \begin{array}{l} grid(G), \\ locations(L) \end{array} \right\}, \left\{ \begin{array}{l} in\_proximity(A,B,T) \leftarrow \\ \quad located\_at(A,L_1,T) \wedge \\ \quad adjoining(L_1,L_2) \wedge \\ \quad located\_at(B,L_2,T), \\ in\_proximity(A',B',T') \leftarrow \\ \quad located\_at(A',L',T') \wedge \\ \quad located\_at(B',L',T') \wedge \\ \quad not(A' = B') \end{array} \right\} \cup G \cup L, \{located\_at\})$$

Engineers using this sort of system control design only through the problem description - in other words, they have little or no control over the decisions taken by the *Select* and *Best* predicates introduced earlier. Such decisions often involve domain-specific knowledge so the size of problem which we can tackle by this means is limited by the amount of such knowledge which we can supply and maintain. Nevertheless, there exist practical applications which are small enough to be tractable but large enough to be worthwhile. For example:

- The WWW site for our research group (URL `http://www.dai.ed.ac.uk/groups/ssp/index.html`), was built and is maintained by automatic generation from a problem description. It can be explained succinctly in terms of the abstract description above. The problem description, Φ, is given using a simple set of facts describing our research group. Each $component(N, C, D, P)$ specifies part of the site by instantiating D via conditions C derivable from Φ. The conditions in C can all be determined automatically in this very narrow application, and the range of possible components is sufficiently small that *Select* and *Best* can also be determined automatically. In other words, those altering the problem description don't need to know how the specification is synthesised. The specification is converted automatically to HTML via the Pillow translator (currently available from `http://www.clip.dia.fi.upm.es`).
- A prototype system for connecting safety shutdown logic to segments of codes of design practice is described in [5]. In this case, the problem description, Φ, is a set of formal descriptions of parts of the codes of practice. Each $component(N, C, D, P)$ defines a standard segment of the specification of the shutdown logic, where C relates D to Φ. The control over choice of component (via *Select* and *Best*) is under the control of designers. The design support system keeps track of how the codes of practice relate to the designers' choices of components.

3 Synthesis by Slices

The idea of constructing predicates via a sequence of "slices" manifests itself in different forms in the logic programming community. We have been most strongly influenced by the style of development called "techniques editing", first explained in [6] and later elaborated by others (for example [3]). Our method of designing these systems is as follows:

- We identify the basic flows of control (often called "skeletons") which our target group of engineers will want to use. Each of these is defined by a set of clauses which define a predicate P of arity, n. A set of parameterisation conditions, Ψ_s, (applied similarly to those described in Section 2) are used to instantiate appropriate variables in the clauses.

$$skeleton(\Psi_s, \left\{ \begin{array}{c} P(A_1, \ldots, A_n) \leftarrow C_1 \\ \vdots \\ P(A'_1, \ldots, A'_n) \leftarrow C_m \end{array} \right\})$$

- We define transformations (often called "techniques") as shown below. Each transformation adds an argument, N, to P and may add additional goals to the body of each clause (transforming each C to C'). Each transformation requires the instantiation of a set of parameterisation conditions, Ψ_t.

$$\left\{ \begin{array}{c} P(A_1, \ldots, A_n) \leftarrow C_1 \\ \vdots \\ P(A'_1, \ldots, A'_n) \leftarrow C_m \end{array} \right\} \stackrel{\Psi_t}{\Rightarrow} \left\{ \begin{array}{c} P(A_1, \ldots, A_n, N_1) \leftarrow C'_1 \\ \vdots \\ P(A'_1, \ldots, A'_n, N_m) \leftarrow C'_m \end{array} \right\}$$

  These transformations normally do not alter the original flow of control of the predicate, thus making it easier subsequently to combine predicates.

- We build an interface with which the target group of engineers can select a flow of control and progressively extend it. We summarise this process using the predicate $techniques(D)$ which succeeds if there is a definition, D, obtainable by techniques editing. The predicate $Choose(S_f, F)$ chooses an appropriate flow of control, F, from the set of skeletons, S_f. $Instantiate(\Psi)$ instantiates the parameters of a skeleton or of a transformation. $Next(S_t, T)$ chooses a transformation step appropriate to the current partial definition. $Completed(D)$ denotes that we have applied sufficiently many transformations to complete our definition.

$$\begin{array}{rcl} techniques(D) & \leftarrow & Choose(\{(\Psi'_s, D'_s) | skeleton(\Psi'_s, D'_s)\}, (\Psi_s, D_s)) \wedge \\ & & Instantiate(\Psi_s) \wedge \\ & & extend(D_s, D) \end{array}$$

$$\begin{array}{rcl} extend(D_s, D) & \leftarrow & Next(\{(\Psi'_t, D'_n) | D_s \stackrel{\Psi'_t}{\Rightarrow} D'_n\}, (\Psi_t, D_n)) \wedge \\ & & Instantiate(\Psi_t) \wedge \\ & & extend(D_n, D) \\ extend(D, D) & \leftarrow & Completed(D) \end{array}$$

This approach is most often applied to small-scale design of predicates, using a small library of highly adaptable skeletons and techniques. An example of a skeleton in such a library might be a definition of list traversal, where the parameterisation conditions $predicate\_name(P)$ and $number\_of\_cases(N)$ supply the predicate name and number of recursive cases. Each $\mathcal{T}$ is a test which is later supplied by the user of the program and is used to differentiate the recursive cases of the skeleton.

$$skeleton(\{predicate\_name(P), number\_of\_cases(N)\}, \left\{ \begin{array}{l} P([H_1|T_1]) \ \leftarrow \ \mathcal{T}_1 \wedge P(T_1) \\ \vdots \\ P([H_N|T_N]) \ \leftarrow \ \mathcal{T}_N \wedge P(T_N) \\ P([]) \end{array} \right\})$$

A difficulty with using skeletons and techniques at this scale is that the decisions which users must make (both in selecting and parameterising the components) are both numerous and detailed. In the example above we must know we want to base our program on list traversal in order to choose the skeleton and, once chosen, we must be able to say how many recursive cases we need. In some cases it can be desirable to make such decisions explicit. For example:

- In our use of techniques editing in supporting novice Prolog programmers [1] we worked at a fine level of detail but limited the diversity of the library used by novices so that the range of decisions they had to make was still small. The parameterisation conditions for skeletons, Ψ_s, are established via simple questions such as the number of cases in a traversal skeleton. The parameterisation conditions for techniques, Ψ_t, is similarly direct. At any point in design, the choice of *Next* step is from a small repertoire and the decision of when the design is *Completed* can be determined with respect to whatever example the novice programmer chose to tackle.

- A more diverse library is contained in the techniques editing tool of the LSS system. We have used this for more sizeable designs - the largest being a reconstruction of the tool itself [8]. However, this sort of system can only be used by engineers who are trained in the use of logic programming techniques because the questions which one must answer in satisfying Ψ_s and Ψ_t can be sophisticated (such as the precise form of term deconstruction). The choice of *Next* step, although from a fixed set of options each time, also demands Prolog expertise. Only designers can tell when the design is *Completed* because it is always possible to add more slices to a predicate.

- In more recent work, Castro has applied techniques editing to a narrow domain of application (the synthesis of animal population dynamics models). In his system the library of skeletons and techniques is domain-specific and Ψ_s and Ψ_t are complex. To avoid the ecological modellers using his system from having to learn about parameterisation Castro used a problem description (similar in purpose to the one in Section 2) to allow description of features in ecological terminology. These are then translated into the language needed for parameterisation.

4 Enhancement by Meta-Interpretation

Sometimes when we have built programs we want to construct additional mechanisms which don't change the definitions of the original programs but give us new ways of using them. For example, we might want to relate the execution of a program to some form of commentary on what it is doing. A natural way of doing this in a logic programming style is through meta-interpretation. Different variants of this method of design exist but we have been most influenced by the ideas of layered meta-interpretation given in [11]. This type of meta-interpreter tends to be problem-specific so the method is best explained by a simplified example.

Suppose that we have a collection of equations of the form $equation(V_v, I, P)$, where V_v is a variable-value pair of the form $value(X, V)$ (with X being the name of the output variable and V recording its value); I is the list of inputs needed by the equation; and P is the procedure relating inputs to output. For example:

$$equation(value(x, X), [value(y, Y)], X \ is \ Y + 10)$$
$$equation(value(y, Y), [], Y = 5)$$

We might write a simple interpreter for these equation definitions which solves for any variable-value pair by selecting an appropriate equation; solving its inputs recursively and calling the equation's procedure.

$$solve(V_v) \quad \leftarrow \quad equation(V_v, I, P) \ \wedge \ solve\_inputs(I) \ \wedge \ P$$

$$\begin{array}{rcl} solve\_inputs([V_v|T]) & \leftarrow & solve(V_v) \ \wedge \ solve\_inputs(T) \\ solve\_inputs([]) & & \end{array}$$

We can use this program to find the values for variables. for example, solving the goal $solve(value(x, X))$ binds X to 15.

We may later find that just solving the equations isn't enough. We also want to retain certain contextual information related to our interpreter's choice of variable-values. We use assertions of the form $context(V_v, E)$ to denote that the variable-value pair V_v is endorsed by the set of contextual information E. For example:

$$\begin{array}{l} context(value(x, V), \{cx\}) \\ context(value(y, V), \{cy\}) \end{array}$$

We could adapt our original interpreter to propagate this contextual information by adding an extra "slice" to predicates *solve* and *solve_inputs*. For example, the program:

$$e\_solve(V_v, E_i \cup E_v) \quad \leftarrow \quad equation(V_v, I, P) \ \wedge \ e\_solve\_inputs(I, E_i) \ \wedge \ P \ \wedge \ context(V_v, E_v)$$

$$\begin{array}{rcl} e\_solve\_inputs([V_v|T], E_v \cup E_r) & \leftarrow & e\_solve(V_v, E_v) \ \wedge \ e\_solve\_inputs(T, E_r) \\ e\_solve\_inputs([], \{\}) & & \end{array}$$

will find both variable-values and endorsements by satisfying goals of such as $e\_solve(value(x, X), E)$, binding X to 15 and E to $[cx, cy]$. The problem in doing this is that the means of solving the equations and propagating endorsements ar now closely bound together in the same predicate definitions. An alternative way of achieving the same result without transforming the equation interpreter is to define the endorsement strategy as a meta interpreter which sits "on top of" the original equation interpreter. For example:

$$\begin{array}{rcl} endorsed(solve(V_v), E_b \cup E_v) & \leftarrow & clause(solve(V_v), B) \ \wedge \\ & & context(V_v, E_v) \ \wedge \\ & & endorsed(B, Eb) \\ endorsed((A \wedge B), E_a \cup E_b) & \leftarrow & endorsed(A, E_a) \ \wedge \\ & & endorsed(B, E_b) \\ endorsed(G, \{\}) & \leftarrow & not(G = (A \wedge B)) \ \wedge \\ & & is\_built\_in(G) \ \wedge \\ & & G \\ endorsed(G, E) & \leftarrow & not(G = solve(V_v) \ \vee \ is\_built\_in(G)) \ \wedge \\ & & clause(G, B) \ \wedge \\ & & endorsed(B, E). \end{array}$$

Using this interpreter we can obtain the same endorsements as before by giving goals such as $endorsed(solve(value(x, X)), E)$. However, we have a much looser connection between the equation solving and endorsement strategies, so we could alter either (within limits) without having to change the other. We don't have this flexibility in the *e_solve* definition.

We have applied this sort of design strategy in reconstructing simulation models which were originally implemented in a System Dynamics notation (a standard process engineering paradigm in which state variables are thought of as "tanks" with "flows" of substance between them being controlled by systems of equations). The largest of these is the BIONTE model of key vegetation production processes in the Amazon. Its Carbon and Nitrogen sub-models contain a total of 210 variables, of which 60 are parameters. Each of these parameters (and the equations connecting them to the 14 state variables) were obtained from different combinations of field data, physiological measurement and literature survey. Hence the context of the model is highly heterogeneous. Brilhante [2] reconstructed the BIONTE Carbon and Nitrogen sub-models using meta-interpretation to reproduce the equation solving strategy for simulation and confirmed that it produced results which closely matched the original model[2]. We then extended the meta-interpreter to propagate contextual information, which could then be used to assess the heterogeneity of the data supporting predictions given by the model. The original extensions to the interpreter were by normal extension (like in *e_solve* above) but we later used a layered approach.

The main difficulty in this style of design is its flexibility. The point of meta-interpretation is normally to express some sort of explicit problem solving strategy and these vary depending on the problem in hand. Although it is possible to imagine sets of parameterisable components or techniques (in the style described earlier) for building meta-interpretation strategies, we are not aware of this being done in practice. Until it is, we lack the empirical evidence to determine whether useful notions of standard practice exist at this level of description.

5 Conclusions

Strong notions of pragmatics are necessary in the application of formal methods because engineers cannot afford to waste time trying numerous inappropriate (but mathematically correct) styles of description before learning ones which are appropriate to their style of work. Our descriptions of pragmatics are based on patterns of use of logic, tempered with experience in how and where to apply these patterns. We have shown three such patterns; described the problems we have tackled using them; and explained some of the factors which limit the sizes of our systems.

In describing how ideas from logic program underpin these applications we have left aside the discussion of the interfaces by which groups of engineers supply the necessary problem descriptions (Φ and Ψ) and satisfy the selection conditions ($Select$, $Best$, $Choose$, $Instantiated$ and $Completed$). Inventing these user interfaces has been a major issue in almost every system we have designed but space limitations prohibit a discussion of these issues here. Readers with an interest in the interfaces used should refer to the papers cited for each system. Unlike the underlying formal methods, the interfaces we build normally differ between domains of application because each new domain brings new assumptions about the way in which groups of engineers wish to communicate.

One of the advantages of systems with explicit notions of pragmatics should be that they

[2] This surprised us since our interpreter used simple differences between time points when applying equations, rather than the approximations to continuous change normally used in System Dynamics modelling.

can be used cooperatively. If we possess additional knowledge about how a specification has been built then perhaps we can use this knowledge in other systems which are used to extend or adapt the existing definition. This issue isn't well explored but it is suggested by the methods we have described. The notions of parameterisation from Section 2 can be used to supply components for later extension by the techniques of Section 3. The products of either of these two methods could be augmented by meta-interpretation of the sort described in Section 4. We are a long way from fully utilising this form of synergy but an example of what we can do with existing tools is given in [7].

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] V. Brilhante. A system for representing uncertainty of evidence in ecological models. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1995. MSc Thesis.

[3] T.S. Gegg-Harrison. Basic prolog schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University, September 1989.

[4] J. Goguen. *Principles of Parameterised Programming*, pages 159–225. Addison Wesley, 1989.

[5] J. Hesketh, D. Robertson, N. Fuchs, and A. Bundy. Lightweight formalisation in support of requirements engineering. *Journal of Automated Software Engineering*, 5(2):183–210, 1998.

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

[7] D. Robertson. Distributed specification. In *Proceedings of the 12th European Conference on Artificial Intelligence*, Budapest, Hungary, August 1996.

[8] D. Robertson. An empirical study of the lss specification toolkit in use. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering, Nevada, USA*. Knowledge Systems Institute, Illinois, 1996. ISBN 0-9641699-3-2.

[9] 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.

[10] D.T. Sannella and L.A. Wallen. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.

[11] L.U. Yalcinalp and L.S. Sterling. Uncertainty reasoning in prolog with layered meta-interpreters. Technical Report TR 90-110, Center for Automation and Intelligent Systems Research, Case Western Reserve University, Ohio, USA, 1990.