Expressing Program Requirements using Refinement Lattices

Dave Robertson †, Jaume Agustí ‡, Jane Hesketh †, Jordi Levy ‡
†Department of Artificial Intelligence, University of Edinburgh.
‡IIIA, Centre d'Estudis Avançats de Blanes, Blanes, Spain.

Abstract. Requirements capture is a term used in software engineering, referring to the process of obtaining a problem description – a high level account of the problem which a user wants to solve. This description is then used to control the generation of a program appropriate to the solution of this problem. Reliable requirements capture is seen as a key component of future automated program construction systems, since even small amounts of information about the type of problem being tackled can often vastly reduce the space of appropriate application programs. Many special purpose requirements capture systems exist but few of these are logic based and all of them operate in tightly constrained domains. In previous research, we have used a combination of order sorted logic (for problem description) and Prolog (for the generated program) in an attempt to provide a more general purpose requirements capture system. However, in our earlier systems the connection between the problem description and the resulting program was obtained using *ad hoc* methods requiring considerable amounts of domain–specific information, thus limiting the domain of application of the system. We are experimenting with languages which provide a formal connection between problem description and application program, thus eliminating the need for domain–specific information in the translation process. This paper introduces a formal language for requirements capture which bridges the gap between an order sorted logic of problem description and the Prolog programming language. The meaning of a Prolog predicate is often characterised according to the set of bindings which can be obtained for its arguments. It is therefore possible to develop a hierarchical arrangement of predicates by comparing the sets of results obtained for stipulated variables. Using this hierarchical structure, we provide proof rules which may be used to support part of the requirements capture process. We describe the notation used for the refinement lattice; define its relationship to Prolog and demonstrate how the language can be used to support requirements capture. An interactive system for extracting Prolog programs from our refinement hierarchies, using an algorithm similar to the one described in this paper, has been implemented.

1 Introduction

Previous work on requirements capture, described in [4], attempted to control the generation of Prolog programs by applying domain knowledge from a problem description supplied by the user. The point of having a problem description separate from the application program was to enable the formal language in which users described the domain to be fitted more closely to terminology with which they would be familiar (our intended users were ecologists with little programming expertise). A diagram representing the general architecture of the main system used in this research is shown in Figure 1. In it there are two key mechanisms: the program generator which constructs Prolog programs by assembling components from a library of program schemata; and the front–end package which assists the user in selecting and restricting template sentences to form a problem description. The problem description connects the front–end package to the program generator, since statements in the problem description are

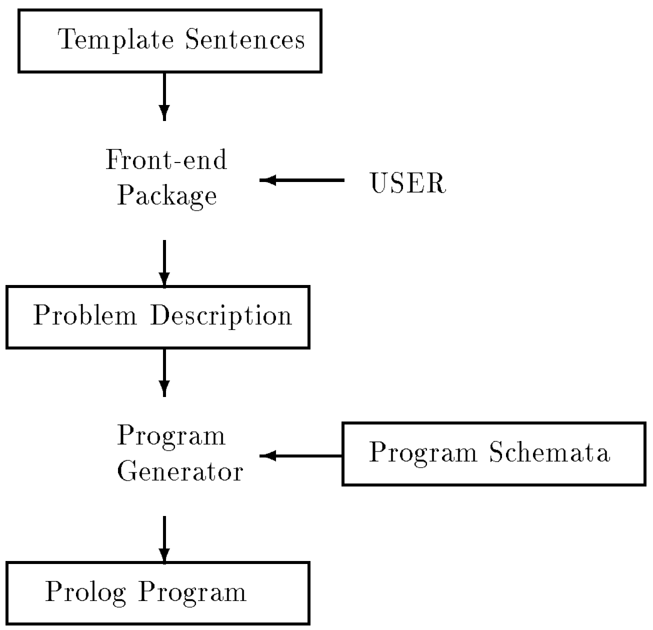

Figure 1: General architecture of the existing requirements capture system

used to control the selection and application of schemata during the construction of the program.

This approach is attractive because it buffers users from part of the programming task. However, there is a tension between the demands of users for a notation to which they can relate and the need for computational sophistication in their application programs. This tends to create a conceptual gap between the languages of problem description and application. The trade–offs which were made in attempting to bridge this gap are discussed in [5] but the end–result is normally that the language used for problem description is different from the language used to describe the application program. This can become a serious problem if the means by which the two languages interact during program generation is not well understood.

One way to tackle these problems is to devise a language which can be used for problem description but also has a straightforward translation to an application programming language. This language has to be expressive but it must also be easy to use. In addition, it should be capable of describing a programming problem in general terms or in greater detail, depending on users' preferences. Previous work by Bundy and Uschold ([1]) has attempted to provide this sort of uniform language based on typed lambda calculus but they have yet to implement these ideas in a working system and the complexity of the mathematics involved makes it difficult to see how users without specialist training could feel confident about using it. A solution to this problem would be to "dress up" the mathematics in a form which is more easily understood. Unfortunately, it is often difficult to make an inherently complex notation appear simple. An alternative, which we adopt in this paper, is to start with comparatively simple underlying principles and to manipulate these to obtain complex programs. A good source of ideas for this approach is in logic programming, which (in the form of pure Prolog programs) embodies a simple but powerful programming paradigm. A second source of inspiration is to be found in recent set–based specification languages. In particular we have drawn upon ideas from the COR system of refinements ([2]).

The core of the requirements capture language depends on representing a lattice of sets of results of predicates. This constitutes our problem description language. Section 2 introduces this notion in the context of Prolog[1]. This is followed, in Section 3 by a description of the way in which expressions in the language may be translated into Prolog. Since this is intended to be a high level language, not all of the axioms translate directly into Prolog and some are used, with the aid of proof rules, to control problem description. In Section 4 we describe some of the proof rules which we use later, in Section 5, to provide guidance in defining set lattices. Finally, in Section 6, we describe how programs (at differing levels of detail) may be extracted from our lattices.

2 Denoting Argument Sets

It is conventional to define the meaning of a logic program to be the set of ground unit goals deducible from that program. Thus, if we have the program shown below:

$$\begin{array}{rcl} grandparent(A,B) & \leftarrow & parent(A,C) \,\&\, parent(C,B) \\ parent(fred,joe) & & \\ parent(joe,mary) & & \\ parent(ann,joe) & & \end{array} \tag{1}$$

then the meaning of the predicate $grandparent/2$ in this program would be described by the set of unit goals:

$$\{grandparent(fred,mary), grandparent(ann,mary)\} \tag{2}$$

This gives a form of "global" meaning for a predicate in terms of all its arguments but it is possible to define more local interpretations in terms of stipulated arguments. We shall use the notation $V : P$ to denote the set of instances for the variable V which can be obtained from the goal P.

Using program (1) as an example, we can use the ':' operator to denote the sets of instances obtained for either or for both of the arguments to $grandparent/2$, giving the following three sets:

$$\begin{array}{rcl} A : grandparent(A,X) & = & \{fred, ann\} \\ B : grandparent(X,B) & = & \{mary\} \\ (A,B) : grandparent(A,B) & = & \{(fred,mary),(ann,mary)\} \end{array}$$

To simplify the descriptions in this paper we shall assume that only a single variable appears on the left hand side of the ':' operator. However, it should be possible to extend all the definitions of this paper to the more general case of a vector of variables.

In order sorted logics, it is normal to restrict the range of objects over which variables in formulae are permitted to range. We can achieve this effect using our notation by permitting the variables inside the goal expression to be restricted using the ':'

---

[1]Throughout this paper we shall be using "pure" Prolog, without complicating features such as cut or side–effecting predicates

operator. This permits any predicate to be applied over sets of objects, rather than over individuals as would be the case in standard first–order predicate calculus. The interpretation of a predicate argument applied in this way is defined as the set of results for the variables on the left of the ':' operator, given the application of the predicate to every combination of elements in the sets denoted in its arguments. For example, if we take the *parent*/2 definitions from program 1 then we could define the following set:

$$A : parent(A, B : parent(B, X)) \tag{3}$$

To obtain the set denoted by the above expression we first find the interpretation for its sub–expression:

$$B : parent(B, X) \quad = \quad \{fred, joe, ann\}$$

We then obtain the set of all solutions for the goals:

$$\begin{aligned} A : parent(A, fred) \quad &= \quad \{\} \\ A : parent(A, joe) \quad &= \quad \{fred, ann\} \\ A : parent(A, ann) \quad &= \quad \{\} \end{aligned}$$

The union of these sets gives us the interpretation for the original expression[2]:

$$A : parent(A, B : parent(B, X)) \quad = \quad \{fred, ann\}$$

Since all of our terms represent sets of objects we can introduce some standard set operators as follows:

Definition 1 *If A and B are set expressions then we allow the set operators:*

- *$A \cap B$ for the intersection of A and B.*
- *$A \cup B$ for the union of A and B.*
- *$A \supseteq B$ if B is a subset of A.*

This allows us, for example, to say that the set of parents is larger than the set of grandparents :

$$A : parent(A, X) \quad \supseteq \quad B : grandparent(B, Y)$$

The use of the $\supseteq$ operator allows us to arrange our set expressions into a lattice. To provide a "top" and "bottom" to this lattice we shall use the symbol $\top$ to denote the entire universe of discourse and $\bot$ to denote the empty set of objects. The full syntax of refinement expressions appears below:

[2]This is the same interpretation as we obtained earlier for $A : grandparent(A, X)$

Definition 2 *A refinement formula is of the form $H \supseteq B$, where:*

- *H is the head of the refinement and is a primitive set expression.*
- *B is the body of the refinement and can be any set expression.*
- *A primitive set expression is of the form $V : E$, where V is a variable appearing in E and E is one of the following:*
  - *A Prolog goal.*
  - *A term of the form $Q(A_1, \cdots, A_N)$, where Q is a predicate name and each A_I is either a variable, constant or set expression.*
- *A set expression is one of the following:*
  - *A primitive set expression, $V : E$*
  - *A union of set expressions, $V_1 : E_1 \cup V_2 : E_2$*
  - *An intersection of set expressions, $V_1 : E_1 \cap V_2 : E_2$*
  - *The difference between two set expressions, $V_1 : E_1 - V_2 : E_2$*

  V is said to be restricted by the expression E. Any variable which is not restricted in this way is said to be unrestricted.

The next section will make more clear why the restrictions on syntax supplied in definition 2 are needed. It is worth noting in passing that set expressions for first order predicate calculus have also been introduced in [3] but in a different form and for different purposes.

3 Mapping Prolog to the Refinement Language

Section 2 introduced the basic notation for the refinement language. The purpose of this section is to show how the language can be understood in terms of Prolog. To simplify our explanation, we shall demonstrate the correspondence for unary predicates but the same principles apply to predicates of any arity.

The $\supseteq$ operator can be interpreted in terms of the $\leftarrow$ operator by recognising that if we have a formula such as:

$$V_1 : P(V_1) \quad \supseteq \quad V_2 : Q(V_2) \tag{4}$$

then it must be true that any successful result for $Q(V_2)$ would imply the same result for $P(V_1)$. Therefore we can rewrite the formula as:

$$P(V) \quad \leftarrow \quad Q(V) \tag{5}$$

The $\cap$ operator can be interpreted in terms of the & operator because the intersection of the results from two goals must be the same as the set of results from the conjunction of those goals. Thus the expression:

$$V_1 : P(V_1) \quad \cap \quad V_2 : Q(V_2) \tag{6}$$

corresponds to the formula:

$$P(V) \quad \& \quad Q(V) \tag{7}$$

Similarly, the $\cup$ operator can be interpreted in terms of the $\vee$ operator by rewriting expressions of the form:

$$V_1 : P(V_1) \quad \cup \quad V_2 : Q(V_2) \tag{8}$$

to produce the new expression:

$$P(V) \quad \vee \quad Q(V) \tag{9}$$

Notice that the set of results obtainable from 7 and 9 are the sets denoted by set expressions 6 and 8, respectively.

Finally, any nested variable restrictions (using the ':' operator) within terms must be converted into preconditions for logical rules. Thus, if we have an expression of the form:

$$V_1 : P(V_1) \quad \supseteq \quad V_2 : Q(V_3 : A(V_3), V_2) \tag{10}$$

we would rewrite it to the expression:

$$P(V) \quad \leftarrow \quad A(V_3) \ \& \ Q(V_3, V) \tag{11}$$

It is important to remember that not all the refinement formulae are intended to translate directly into Prolog. In general, the refinement relation is more "permissive" than standard implication and with it we can represent a wide variety of information, only part of which is sufficiently precise to constitute a Prolog program. In particular, it is not always possible to translate from refinements which have restricted variables in the head but these variables do not appear in the body, since these introduce existential variables into the head of the clause. Thus an expression such as 12, below, can not be guaranteed to translate into Prolog.

$$V_1 : P(V_2 : A(V_2), V_1) \quad \supseteq \quad V_3 : Q(V_3) \tag{12}$$

For example, we could define the following refinement for the predicate $add(A, B, C)$, which adds together the natural numbers A and B to obtain C:

$$N : natural(N) \quad \supseteq \quad C : add(A : natural(A), B : natural(B), C) \tag{13}$$

This could be translated into the following Prolog clause which, although not a particularly useful program, is always true:

$$\begin{array}{rcl} natural(N) & \leftarrow & natural(A)\ \& \\ & & natural(B)\ \& \\ & & add(A, B, N) \end{array}$$

However, we could also write the following refinement:

$$C : add(A : natural(A), B : natural(B), C) \ \supseteq \ N : natural(N) \tag{14}$$

Like refinement 13, this makes sense as a refinement axiom (since all the naturals are included in addition over naturals) but, if translated into Prolog it becomes:

$$\begin{array}{rcl} add(A, B, C) & \leftarrow & natural(A)\ \& \\ & & natural(B)\ \& \\ & & natural(C) \end{array}$$

Clearly, this rule does not always hold. The reason is that refinement 14 is defining a general set property of the $add/3$ predicate – that it can (potentially) generate any natural number – while refinement 13 defines a direct relationship between the naturals and addition. Since our refinement language is, in this sense, very flexible we must be careful which axioms are allowed to be translated into Prolog. However, provided such checks are in place, we can benefit from the extra flexibility during problem description. For this, we need to use some standard proof rules, which are the topic of the next section.

4 Refinement Proof Rules

Since all the expressions in the language refer to sets, we can use proof rules from set theory to perform many of the operations necessary during requirements capture. This section describes some of the proof rules which we currently use and we anticipate that further, derived rules will be added to the collection as the system matures – for instance, rules describing the preservation of unions and intersections of predicates and a full set of rules for the set difference operator. In subsequent sections we shall show some of these rules in operation. In the proof rules which follow, the symbols A, B and C denote set expressions.

Proof rule 1 *The universal set ($\top$) includes any set:*

$$\vdash \ \top \supseteq A$$

Proof rule 2 *Any set includes the empty set ($\bot$):*

$$\vdash \ A \supseteq \bot$$

Proof rule 3 *Any set includes itself:*

$$\vdash \quad A \supseteq A$$

Proof rule 4 *The refinement relation is transitive:*

$$A \supseteq B, B \supseteq C \quad \vdash \quad A \supseteq C$$

Proof rule 5 *A set, C, includes the union of any two sets which it includes separately:*

$$C \supseteq A, C \supseteq B \quad \vdash \quad C \supseteq A \cup B$$

Proof rule 6 *The union of a set with any other set includes the original set:*

$$\vdash \quad A \cup B \supseteq A$$
$$\vdash \quad A \cup B \supseteq B$$

Proof rule 7 *The intersection of a set with any other set is included in the original set:*

$$\vdash \quad A \supseteq A \cap B$$
$$\vdash \quad B \supseteq A \cap B$$

Proof rule 8 *A set which is included independently in two others is included in their intersection:*

$$B \supseteq A, C \supseteq A \quad \vdash \quad B \cap C \supseteq A$$

Proof rule 9 *The union of two intersections is the same as the intersection of the unions:*

$$\vdash \quad (A \cap B) \cup (A \cap C) \supseteq A \cap (B \cup C)$$

Proof rule 10 *A set expression is included within another set expression if they have the same predicate name and arity and the terms at corresponding argument positions are refinements. For ease of explanation, we have shown the simpler case of this rule for a predicate with a single argument:*

$$A \supseteq A' \quad \vdash \quad V : P(A) \supseteq V' : P(A')$$

5 Defining a Refinement Lattice

In Section 6 we describe how a program may be extracted from a refinement lattice. As a precursor to this, we explain how such lattices may be constructed and show how the refinement language may be used to help control their development.

It would be possible to define complete programs entirely within the refinement language. For example the standard *append*/3 program, which concatenates the lists in its first and second arguments to form the list in its third argument, could be defined as:

$$
\begin{aligned}
L2 : append([H|T], L1, [H|L2]) &\supseteq L3 : append(T, L1, L3) \\
L : append([], L, L) &\supseteq L1 : list(L1)
\end{aligned} \tag{15}
$$

Using the translations described in Section 3, this could be rewritten into the Prolog definition:

$$
\begin{aligned}
append([H|T], L1, [H|L2]) &\leftarrow append(T, L1, L2) \\
append([], L, L) &\leftarrow list(L)
\end{aligned} \tag{16}
$$

However, this doesn't seem to us to be the most advantageous use of the language, since it merely replicates a standard logic program. In defining refinement lattices a key idea is that people should be allowed to "rise above" the level of the application program in the initial stages of refinement. The language supports this by allowing two ways of adding to the lattice: by creating new refinements and by extending existing refinements. We shall consider each in turn below.

5.1 Creating New Refinements

When creating a new refinement it is possible to assist the user in two ways. The first is by flagging any "gaps" in the specification which are created by the addition of the refinement. This happens when an axiom is introduced which refers to terms which are not defined in the existing refinement lattice. For example, if we introduce a refinement denoting that a possible refinement of diagnoses would be the set of confirmed diseases:

$$
D : diagnosis(D) \supseteq C : confirmed(X : disease(X), C) \tag{17}
$$

then we have introduced two new set expressions: $confirmed/2$ and $disease/1$, which may be defined in the refinement lattice. If we wish to attach $disease/1$ at the top of the lattice we could add the refinement:

$$
\top \supseteq X : disease(X) \tag{18}
$$

Note that this attachment says nothing about the meaning of $disease/1$. It merely introduces it as a predicate of arity 1.

5.2 Extending Refinements

In addition to adding new information, it is common to want to combine existing refinements in order to be more specific about the way in which they apply. To support this process we permit users to restrict the size of a refinement expression on either (or both) the left or right sides. Since this could result in an overdefined expression – for example by over–restricting the left–hand side of the refinement – we must also apply a test for overdefinition to the resulting expression (See Section 5.3).

Definition 3 *A refinement of the form $A \supseteq B$ is an extension of the refinement lattice $\mathcal{H}$ if:*

- $A' \supseteq B' \in \mathcal{H}$ *and*
- $A' \supseteq A$ *and*
- $B' \supseteq B$

For example, we might have added the information that locations of fish are included in aquatic habitats; that aquatic habitats include rivers and that Carp are fish:

$$H : aquatic\_habitat(H) \supseteq L : location(F : fish(F), L) \tag{19}$$
$$H : aquatic\_habitat(H) \supseteq R : river(R) \tag{20}$$
$$F : fish(F) \supseteq C : carp(C) \tag{21}$$

Now if we add the information that the locations of Carp are included in rivers:

$$R : river(R) \supseteq L : location(C : carp(C), L) \tag{22}$$

we can show that this is a valid extension as follows:

- By definition 3 using axiom 19, we have an extension if:
$L : location(F : fish(F), L) \supseteq L : location(C : carp(C), L)$ and
$H : aquatic\_habitat(H) \supseteq R : river(R)$
- By proof rule 8 we can establish that:
$L : location(F : fish(F), L) \supseteq L : location(C : carp(C), L)$ if
$F : fish(F) \supseteq C : carp(C)$.
- $F : fish(F) \supseteq C : carp(C)$ from axiom 21.
- $H : aquatic\_habitat(H) \supseteq R : river(R)$ from axiom 20.

5.3 Preventing Overdefinition of Refinements

We would like, as far as possible, to protect users against including refinements which are overdefined within the existing lattice. We use the symbol, $\perp$, as the empty set expression and assume that all new sets added will be (potentially) larger than $\perp$. Therefore:

Definition 4 *A refinement, $A \supseteq B$ is overdefined if, in conjunction with the other axioms of the existing refinement lattice, $A \supseteq B \vdash \perp \supseteq B$.*

One of the main purposes of this definition is in limiting the ways in which set expressions can be refined, thus reducing the range of choices available to users when constructing the lattice. We shall describe two techniques for providing this type of control: mutual exclusion and argument restriction.

5.3.1 Mutually Exclusive Set Expressions

We can provide a means of trapping mutually exclusive sets by adding axioms for stating which intersections between sets are not permitted, then using these to test for overdefinedness of the lattice.

Definition 5 *$X_1 : A_1$ and $X_2 : A_2$ are mutually exclusive if* $\perp \supseteq ((X_1 : A_1) \cap (X_2 : A_2))$.

For example, we might want to have sets corresponding to even and odd numbers:

$$N : number(N) \supseteq E : even(E) \tag{23}$$
$$N : number(N) \supseteq D : odd(D) \tag{24}$$

We could then add the information that there is nothing which is both even and odd by adding the axiom:

$$\perp \supseteq E : even(E) \cap D : odd(D) \tag{25}$$

If we then attempt to define a set (call it $bad/1$) which is a refinement of both even and odd, using the axioms:

$$E : even(E) \supseteq B : bad(B) \tag{26}$$
$$D : odd(D) \supseteq B : bad(B) \tag{27}$$

To prove that this is overdefined we need to show that:

$$\perp \supseteq B : bad(B)$$

By rule 4, using axiom 25:

$$\begin{array}{c} \perp \supseteq E : even(E) \cap D : odd(D) \\ E : even(E) \cap D : odd(D) \supseteq B : bad(B) \end{array} \vdash \perp \supseteq B : bad(B)$$

By rule 8:

$$\begin{array}{c} E : even(E) \supseteq B : bad(B) \\ D : odd(D) \supseteq B : bad(B) \end{array} \vdash E : even(E) \cap D : odd(D) \supseteq B : bad(B)$$

The preconditions of this rule are satisfied by axioms 26 and 27. Therefore we have proved that our hierarchy is overdefined.

5.3.2 Argument Restriction

It is sometimes useful to be able to define a predicate which can range over only particular sets of arguments but not others. For example, we might want to say that spiders only eat living things. We can express this using the axiom:

$$X : living(X) \;\supseteq\; X1 : eats(S : spider(S), X1) \tag{28}$$

If we also add the constraint that nothing is both living and dead:

$$\perp \;\supseteq\; X1 : living(X1) \cap X2 : dead(X2) \tag{29}$$

then we can protect against generalisation of the $eats/2$ predicate. For example, if we try to add the axiom:

$$X : eats(S : spider(S), X) \;\supseteq\; X1 : dead(X1) \tag{30}$$

we can prove that this is overdefined as follows:

- By proof rule 4, $\perp \;\supseteq\; X : dead(X)$ if:
  $\perp \;\supseteq\; X1 : living(X1) \cap X2 : dead(X2)$ (Axiom 29) and
  $X1 : living(X1) \cap X2 : dead(X2) \;\supseteq\; X : dead(X)$
- By proof rule 8, $X1 : living(X1) \cap X2 : dead(X2) \;\supseteq\; X : dead(X)$ if:
  $X1 : living(X1) \;\supseteq\; X : dead(X)$ and
  $X2 : dead(X2) \;\supseteq\; X : dead(X)$
- By proof rule 4, $X1 : living(X1) \;\supseteq\; X : dead(X)$ if:
  $X1 : living(X1) \;\supseteq\; X2 : eats(S : spider(S), X2)$ (Axiom 28) and
  $X2 : eats(S : spider(S), X2) \;\supseteq\; X : dead(X)$ (Axiom 30)
- By proof rule 3, $X2 : dead(X2) \;\supseteq\; X : dead(X)$

5.4 A Simple Example

Having defined mechanisms for creating and extending refinements we introduce, in this section, a short example to demonstrate the way in which the language may be used to develop incrementally a requirements specification. We shall use a (somewhat contrived) biological example, in which we wish to represent populations of wolves and deer which have different probabilities of survival depending on their location. To begin, we can introduce the concept of probabilities using the refinement:

$$\top \;\supseteq\; P : probability(P) \tag{31}$$

We could then go on to provide more specific information pertaining to probabilities. In particular, we could say that a more restricted type of *probability* is the survival factor of animals:

$$P : probability(P) \;\supseteq\; S : survival(A : animal(A), S) \tag{32}$$

At his point, we have introduced, as part of expression 32 a requirement for $animal/1$ to be placed in the lattice. This is flagged as one of the gaps in the requirement specification and we plug this gap by adding $animal/1$ below $\top$. At the same time, it is convenient to add $wolf/1$ and $deer/1$, as refinements of $animal/1$, and $red\_deer/1$ as a refinement of $deer/1$:

$$\begin{aligned} \top &\supseteq A : animal(A) && (33)\\ A : animal(A) &\supseteq W : wolf(W) && (34)\\ A : animal(A) &\supseteq D : deer(D) && (35)\\ D : deer(D) &\supseteq R : red\_deer(R) && (36) \end{aligned}$$

We might then decide to introduce a refinement of *survival* which is dependent on the the location of the animals:

$$\begin{aligned} S : survival(A : animal(A), S) \;\supseteq \qquad & (37)\\ F : fl(L : location(A : animal(A), L), B : animal(B), F) \end{aligned}$$

This again introduces a gap in the specification, for $location/2$, which we first introduce below $\top$ and then define using two axioms:

$$\begin{aligned} \top &\supseteq L : location(A : animal(A), L) && (38)\\ L : location(A : animal(A), L) &\supseteq H : hill(H) && (39)\\ L : location(A : animal(A), L) &\supseteq P : pasture(P) && (40) \end{aligned}$$

We might now decide to be more specific about the types of results which we would expect to obtain from $fl/3$. For example, we could stipulate that the results in the third argument for deer on hills might be the integers between 50 and 100, while the same argument for wolves on hills might be the integers between 40 and 60.

$$\begin{aligned} F : fl(L : hill(L), A : deer(A), F) &\supseteq N : between(50, 100, N) && (41)\\ F : fl(L : hill(L), A : wolf(A), F) &\supseteq N : between(40, 60, N) && (42) \end{aligned}$$

Finally, we could be more specific about the locations of particular groups of animals. For example, we could give possible locations for *red_deer* to be hills.

$$L : location(A : red\_deer(A), L) \;\supseteq\; H : hill(H) \tag{43}$$

6 Extracting a Program

In Section 5 we demonstrated how a lattice of refinements could be constructed. This lattice is capable of describing a large number of different programs, which vary on two dimensions:

- The level of detail at which a program in the lattice is described will vary depending on the depth to which we descend through the chains of refinement. The further we travel towards the bottom of the lattice the more detailed our programs become.
- There may be more than one possible refinement of a set expression at any given point in the lattice. These produce choice points in the extraction of program details.

Bearing the above considerations in mind, the method used to extract a program from the refinement lattice is based on a simple principle. Recall the mapping between refinements and implication which has been shown using formulae 4 and 5. Using this mapping, if we take any sequence of refinements down through the lattice from some top level set expression then by translating the refinements of that sequence into axioms of Prolog we shall have produced a partial program the results of which are included in the top–level set expression. For example if we have the sequence of refinements:

$$\begin{aligned} X : a(X) &\supseteq Y : b(Y) \\ X : b(X) &\supseteq Y : c(Y) \end{aligned}$$

Then we could translate these into the Prolog clauses:

$$\begin{aligned} a(X) &\leftarrow b(X) \\ b(X) &\leftarrow c(X) \end{aligned}$$

which, given further definitions for $c(X)$ would allow us to obtain results for $a(X)$ in terms of $c(X)$. Thus we can think of extracting a program from a refinement lattice as traversing the lattice from some top–level set expression, supplying an upper bound on the generality of the program, down to more precise set expressions which supply a lower bound on the program. An interactive system, based on this technique has been implemented and is described in [6] but (to save space) only the basic algorithm is described in this paper.

Some additional complexity is introduced into the algorithm because we permit nesting of set expressions. This means that when we are finding sequences of refinements we need to do more than simply match the left and right sides of the appropriate refinements – we also need to ensure that set expressions contained in the matching expressions can be coerced toward a non–empty intersection. For example, if we have the refinement lattice:

$$\begin{aligned} X : a(C : carnivore(C), X) &\supseteq Y : b(C : carnivore(C), Y) \\ X : b(H : herbivore(H), X) &\supseteq Y : c(H : herbivore(H), Y) \\ C : carnivore(C) &\supseteq B : bear(B) \\ C : herbivore(C) &\supseteq B : bear(B) \end{aligned}$$

then a valid refinement sequence would be:

$$
\begin{aligned}
X : a(B : bear(B), X) &\supseteq Y : b(B : bear(B), Y) \\
X : b(B : bear(B), X) &\supseteq Y : c(B : bear(B), Y)
\end{aligned}
$$

and this would translate to the Prolog clauses given below. Note that this translation involves a further refinement, since we are requiring that each bear in the set of solutions for argument Y of $b/2$ will appear in the set of solutions for argument X of $a/2$.

$$
\begin{aligned}
a(B, X) &\leftarrow bear(B) \ \& \ b(B, Y) \\
b(B, X) &\leftarrow bear(B) \ \& \ c(B, Y)
\end{aligned}
$$

Since traversal of the refinement lattice was required in order to constrain both $carnivore/1$ and $herbivore/1$ to $bear/1$ it may be useful to retain this information in the completed program as well. Therefore the algorithm for unifying set expressions (described later) must accumulate the refinements it uses so that these can be added to the main sequence.

The final component of the algorithm takes care of the cases where either an intersection of set expressions is required or the union of set expressions is required in order to include more than one sequence of refinements into the program. For instance, if we have the refinement lattice:

$$
\begin{aligned}
X : a(X) &\supseteq Y : b(Y) \\
X : a(X) &\supseteq Y : c(Y) \\
X : c(X) &\supseteq Y : d(Y) \\
X : c(X) &\supseteq Y : e(Y)
\end{aligned}
$$

Then, instead of a linear refinement sequence, we could extract from this the refinements:

$$
\begin{aligned}
X : a(X) &\supseteq Y : b(Y) \cap Y : c(Y) \\
X : c(X) &\supseteq Y : d(Y) \cup Y : e(Y)
\end{aligned}
$$

which translate to the Prolog axioms:

$$
\begin{aligned}
a(X) &\leftarrow b(X) \ \& \ c(X) \\
c(X) &\leftarrow d(Y) \vee e(Y)
\end{aligned}
$$

The full refinement algorithm is given below. Note the recursive use of the algorithm when unifying set expressions and also the need to propagate the set intersections from unification through the right hand side of the smaller of the refinement expressions.

Algorithm 1 *We write $refinement(T_1, T_2, P)$ to denote that T_2 is a valid refinement of T_1 producing axiom set P, given refinement lattice $\mathcal{H}$. The algorithm for this is as follows:*

- $refinement(T_1, T_2, P)$ *if* $refinement(T_1, T_2, \{\}, P)$
- $refinement(T, T, P, P)$
- $refinement(V : A, V : B, P, P'')$ *if*
  - $(V : A' \supseteq V : B') \in \mathcal{H}$ *and*
  - $unify(A, A', A_u, P, P')$ *and*
  - $propagate\_bindings(A_u, B', B_u)$ *and*
  - $refinement(V : B_u, V : B, P', P'')$
- $refinement(P(A_1, \cdots, A_n), P(A'_1, \cdots, A'_n), P, P')$ *if*
  - *For each* A_I *and* A'_I: $refinement(A_I, A'_I, P_I)$ *and*
  - $P' = \bigcup P_I \cup P$
- $refinement(V : A, V : B_1 \cap B_2, P, P'')$ *if*
  - $refinement(V : A, V : B_1, P, P')$ *and*
  - $refinement(V : A, V : B_2, P', P'')$

Algorithm 2 *We write* $unify(A, A', A_u, P)$ *to denote that set expressions* A *and* A' *have a shared subset defined by set expression* A_u*, yielding axiom set,* P*. The algorithm for this is as follows:*

- $unify(A, A', A_u, P)$ *if* $unify(A, A', A_u, \{\}, P)$
- $unify(A, A', A_u, P, P'')$ *if*
  - $refinement(A, A_u, P, P')$ *and*
  - $refinement(A', A_u, P', P'')$

Algorithm 3 *The procedure,* $propagate\_bindings(A, B, B')$ *takes each term of the form* $V : X$ *contained in* A *and replaces every occurrence of* $V : \_$ *in* B *with* $V : X$*, yielding the new term,* B'*.*

6.1 A Simple Example

The example in this section uses axioms 31 to 43 from Section 5.4 to provide a refinement lattice. Given these axioms, we demonstrate how programs containing differing levels of detail may be extracted. Our first step must be to specify a set expression which (potentially) contains all the results we require. Suppose that we are interested in a program for determining the survival of *red_deer*. Our top–level goal for the refinement algorithm is therefore:

$$refinement(S : survival(D : red\_deer(D), S), R, P) \tag{44}$$

The algorithm will first descend to the levels of refinement which are closest to the top–level set. Using axiom 37 it can reach the set:

$$S : fl(L : location(D : red\_deer(D), L), D : red\_deer(D), S) \tag{45}$$

In doing this it has had to unify *animal* and *red_deer*, using axioms 35 and 36. The refinements extracted are therefore:

$$\begin{array}{rcl} D : deer(D) & \supseteq & D : red\_deer(D) \\ D : animal(D) & \supseteq & D : deer(D) \\ S : survival(D : red\_deer(D), S) & \supseteq & S : fl(L : location(D : red\_deer(D), L), \\ & & \quad D : red\_deer(D), S) \end{array}$$

Applying the translation algorithm to these refinements gives the partial program:

$$\begin{array}{rcl} deer(D) & \leftarrow & red\_deer(D) \\ animal(D) & \leftarrow & deer(D) \\ survival(D, S) & \leftarrow & red\_deer(D) \ \& \\ & & location(D, L) \ \& \\ & & red\_deer(D) \ \& \\ & & fl(L, D, S) \end{array}$$

We may not be content with this level of detail so we can force the algorithm to search further down through the lattice. Using axiom 41 we can extend downwards to:

$$S : between(50, 100, S) \tag{46}$$

This uses axiom 39 to unify *location* and *hill* so this – along with the axiom corresponding to the refinement step – is added, giving the refinements:

$$\begin{array}{rcl} D : deer(D) & \supseteq & D : red\_deer(D) \\ D : animal(D) & \supseteq & D : deer(D) \\ L : location(D : red\_deer(D), L) & \supseteq & L : hill(L) \\ S : fl(L : hill(L), D : red\_deer(D), S) & \supseteq & S : between(50, 100, S) \\ S : survival(D : red\_deer(D), S) & \supseteq & S : fl(L : location(D : red\_deer(D), L), \\ & & \quad D : red\_deer(D), S) \end{array}$$

Applying the translation algorithm to these refinements gives the partial program:

$$\begin{array}{rcl} deer(D) & \leftarrow & red\_deer(D) \\ animal(D) & \leftarrow & deer(D) \\ location(D, L) & \leftarrow & red\_deer(D) \ \& \\ & & hill(L) \\ fl(L, D, S) & \leftarrow & hill(L) \ \& \\ & & red\_deer(D) \ \& \\ & & between(50, 100, S) \\ survival(D, S) & \leftarrow & red\_deer(D) \ \& \\ & & location(D, L) \ \& \\ & & fl(L, D, S) \end{array}$$

7 A Larger Example

This section describes the refinements necessary to represent a larger example and shows how these can be translated into a working logic program. The problem we have chosen is one of medical diagnosis. Let us assume that we think of diagnosis as being some procedure which suggests diseases based on the symptoms we already know (K) and those we could ask about (A). Our top–level definition might therefore be:

$$D : disease(D) \quad \supseteq \quad D1 : diagnosis(K, A, D1) \tag{47}$$

We might then decide to give some options for what we consider to be valid types of diagnoses. First, we can say that the set of diagnoses would include any diseases we had confirmed:

$$D : diagnosis(K, A, D) \quad \supseteq \quad C : confirmed(K, A, C) \tag{48}$$

A second option for diagnosis might involve asking questions about disease candidates and thereby adding to the list of symptoms we know. Let us further assume that each of the elements we know about is recorded as a data structure of the form $k(S, V)$, where S is a symptom and V is its known value (*e.g.* S might be *sweating* and V might be *profuse*). Each question will have to be contained in a program capable of generating all questions and each value will have to be obtained from a program which can ask the user for appropriate values. Our definition is therefore:

$$\begin{aligned} &D : diagnosis(K, A, D) \supseteq \\ &\quad D1 : diagnosis([k(S : question(C : candidates(K, A, D1, C), S), \\ &\qquad\qquad V : ask(A, S, V)|K], A, D1) \end{aligned} \tag{49}$$

The above two definitions might be sufficient to allow diagnoses in conditions where we always wanted to ask questions until some diseases were confirmed. However, we might also want to allow for diagnoses which were not fully confirmed but still possible (having run out of questions). To allow this possibility we add:

$$D : diagnosis(K, A, D) \quad \supseteq \quad P : possible(K, A, P) \tag{50}$$

We now need to define what it means to be a confirmed disease. For this we shall use a predicate, $for\_each(A, B, C)$, which succeeds if each result A generated by program B satisfies test C; and $member(E, L)$ which succeeds if element E is in the list L. We shall also introduce the predicate, $symptom(D, S, V)$, which succeeds if the disease D has symptom S, with value V. Our definition of confirmed diseases is then:

$$\begin{aligned} &C : confirmed(K, A, C) \supseteq \\ &\quad D : for\_each((S, V), symptom(D, S, V), member(k(S, V), K) \end{aligned} \tag{51}$$

A definition of possible diseases can be obtained using a similar expression to that for confirmed diseases, except that we will be satisfied if only a single symptom is confirmed. For this we need to employ a further predicate, $for\_some(A, B, C)$ which succeeds if any result A generated by program B satisfies test C.

$$
\begin{aligned}
& P : possible(K, A, P) \supseteq \\
& \quad D : for\_some((S, V), symptom(D, S, V), member(k(S, V), K)
\end{aligned}
\tag{52}
$$

To determine the symptom candidates (used in asking questions during diagnosis) we need to find the set of symptoms which are currently not known. For this we shall use the standard Prolog predicates, $setof(A, B, C)$ (which gives the set C of elements of form A such that the goal B succeeds) and the closed-world negation operator, $\neg$.

$$
\begin{aligned}
& C : candidates(K, A, D, C) \supseteq \\
& \quad C1 : setof(S, (symptom(D, S, V), \neg member(k(S, \_), K)), C1)
\end{aligned}
\tag{53}
$$

The set of questions is simply defined as any member of the list of candidate questions:

$$S : question(C, S) \quad \supseteq \quad S1 : member(S1, C) \tag{54}$$

Finally, the set of values which have been successfully asked of the user is defined as those for which the user has been successfully prompted, given the list A of askable symptoms.

$$S : ask(A, S, V) \quad \supseteq \quad V1 : prompt\_user(A, S, V1) \tag{55}$$

The above refinements (47 to 55) can be translated, via the program extraction mechanism, into the following logic program:

$$
\begin{array}{rcl}
diagnosis(K, A, D) & \leftarrow & confirmed(K, A, D) \\
diagnosis(K, A, D) & \leftarrow & candidates(K, A, D, C) \ \& \\
& & question(C, S) \ \& \\
& & ask(A, S, V) \ \& \\
& & diagnosis([k(S, V)|K], A, D) \\
diagnosis(K, A, D) & \leftarrow & possible(K, A, D) \\
confirmed(K, A, D) & \leftarrow & for\_each((S1, V), symptom(D, S1, V), member(k(S1, V), K)) \\
possible(K, A, D) & \leftarrow & for\_some((S1, V), symptom(D, S1, V), member(k(S1, V), K)) \\
candidates(K, A, D, C) & \leftarrow & setof(S, (symptom(D, S, \_), \neg member(k(S, \_), K)), C) \\
question(C, S) & \leftarrow & member(S, C) \\
ask(A, S, V) & \leftarrow & prompt\_user(A, S, V)
\end{array}
$$

8 Conclusions

The language introduced in this paper embodies what we claim to be a novel approach to requirements capture. It has the following features:

- The space of requirements is described using a lattice of refinements between sets of potential results from Prolog programs.
- Construction of a Prolog program can be achieved by searching this requirement space, having delimited the upper and lower bounds within which the completed (partial) program must lie.
- Guidance during the construction of the refinement lattice is obtained by the application of logically consistent set–theoretic proof rules.

Although the algorithms presented in this paper are comparatively simple, we have yet to test whether they can readily be applied by real users. We are currently producing a first prototype (implemented in Prolog) to test our ideas. Of major importance in this activity is to develop new proof rules to help guide users in supplying and extending refinements.

References

[1] A. Bundy and M.. Uschold. The use of typed lambda calculus for requirements capture in the domain of ecological modelling. Research Paper 446, Dept. of Artificial Intelligence, Edinburgh, 1989.

[2] J. Levy, J. Agusti, F. Esteva, and P. Garcia. An ideal model of an extended lambda-calculus with refinement. Ecs-lfcs-91-188, Laboratory for the Foundations of Computer Science, 1991.

[3] D. McAllester, B. Givan, and T. Fatima. Taxonomic syntax for first order inference. In *Proceedings of KR-89*, 1989.

[4] 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.

[5] D. Robertson, M. Uschold, A. Bundy, and R. Muetzelfeldt. The ECO program construction system: Ways of increasing its representational power and their effects on the user interface. *International Journal of Man Machine Studies*, 31:1–26, 1988.

[6] M. Salib. Using refinement logic in requirements capture and program generation. Technical report, Department of Artificial Intelligence, University of Edinburgh, 1992. Unpublished MSc thesis.