MONOGRAFIES DE L'INSTITUT D'INVESTIGACIÓ
EN INTEL·LIGÈNCIA ARTIFICIAL

Institut d'Investigació
en Intel·ligència Artificial

Monografies de l'Institut d'Investigació en Intel·ligència Artificial

Pragmatics in the Synthesis of Logic Programs

David Robertson

Foreword by Jaume Agustí
Institut d'Investigació en Intel·ligència Artificial
Bellaterra, Catalonia, Spain.

Series Editor
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Foreword by
Jaume Agustí
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Volume Author
David Robertson
Institut d'Investigació en Intel·ligència Artificial
Consell Superior d'Investigacions Científiques

Institut d'Investigació
en Intel·ligència Artificial

ISBN: *********
Dip. Legal: **********

Ordering Information: Text orders should be addressed to the Library of the IIIA, Institut d'Investigació en Intel·ligència Artificial, Campus de la Universitat Autònoma de Barcelona, 08193 Bellaterra, Barcelona, Spain.

Printed by CPDA-ETSEIB.
Avinguda Diagonal, 647.
08028 Barcelona, Spain.

Contents

List of Figures

Foreword

TEXT OF FOREWORD GOES HERE

Bellaterra, June 2000

Jaume Agustí
IIIA, CSIC

Abstract

This thesis presents novel uses of structural synthesis methods for Horn clause specifications which are executable as logic programs. These methods apply to the early design and refinement of these specifications - a stage which has been identified as crucial to the adoption of formal specification methods in practical applications.

Two methods of automated design are described. The first employs a distributed form of design in which a number of specialised tools independently construct parts of a specification and share design information via a shared language. Empirical evaluation of this system reveals a number of difficulties stemming from the loose integration between tools and the lack of a framework for incremental refinement of the specification. The second system addresses some of these problems by providing a system of refinement applicable to problems which can be viewed as transformations on sets of axioms. In this system, all Horn clauses express relations over sets of axioms (themselves Horn clauses). This allows early specifications to be defined by constraining these sets. Initially, these specifications are refined by a system of rewrites on inequalities between the sets. Subsequently, specifications are given detail by introducing task-specific skeletal definitions to describe the means of relating elements of sets, and by adding argument slices to carry additional information through the specification.

The methods described in the thesis and supported by the LSS and HANSEL systems have been tested empirically on non-trivial specification tasks, including a reconstruction of one of the experimental tools and the design of models similar to examples taken from the KADS knowledge engineering method.

Part I

Background

Chapter 1

Problem Definition

This thesis is primarily concerned with methods which assist in the early design and refinement of logic programs. The motivation for the research is, however, broader because the problems of early design for logic programs are shared, and are arguably more severe, in the rest of the formal specification community. A recent survey into factors inhibiting the uptake of formal methods in industry [Cleland and MacKenzie, 1995] identifies the following problems:

- The process of applying formal methods to problems does not correspond to existing design processes. Designers want to produce code (or at least executable specifications) quickly and use these to help guide further design. Many existing formal methods make it too slow to devise and revise.
- Most engineers lack basic training in the mathematics needed to apply formal specification methods from scratch, and, if training is supplied, it is difficult to predict which engineers will adapt well to this form of thinking. A substantial part of the difficulty stems not from the mathematics itself but in using it appropriately for particular kinds of problem.
- For those not intimately involved with formal methods there appears to be a plethora of competing methods. Simply selecting the specialised method appropriate to a task is ineffective because there are too many competitors and the selection criteria are seldom defined.
- Project managers view formal methods as "all or nothing" solutions. The initial investment in resources is so high that if they do not deliver high benefits there may not be time or money to shift to conventional design methods, thus jeopardising entire projects.

The root of these problems appears to be in the way formal specification is supported in early design - an issue which has received surprisingly little attention. One reason for this omission is that conventional formal specification methods concentrate on training practitioners in the use of the formal language, under the assumption that deep understanding of the language gives the freedom

to use any design method which the language allows. This may be effective for very small specifications but is unsatisfactory for larger specifications where software process managers may require explicit styles of description and methods of refinement to be followed in order to control, assess and audit design choices at a level higher than the formal language itself. Free-style design can also be difficult if more than one designer is involved because it can be difficult to relate and amalgamate the designs produced in different ways by different people, even if the formal specification language is shared. Examples of these sorts of problems, taken from projects in which the author was involved, are given in [Robertson, 1998b].

Most conventional software engineering methods, and some requirements engineering methods, require different views of the design to be produced at appropriate stages in the design process. These are connected together, either tightly or loosely depending on the method and design stage, and together provide a framework for communicating different forms of information between stages of the design lifecycle. We know that it is possible to introduce formal methods into existing lifecycles, normally with the aim of clarifying the links between design stages but these efforts tend to be shaped by the established lifecycle into which formality is being absorbed [Robertson and Agusti, 1999]. This thesis looks at the problem from the opposite perspective: taking a commonly used style of computational logic and exploring how the pragmatics of design in that logic inspire certain kinds of lifecycle model.

This type of study can never be confined purely to formal theory because judgement about what constitutes a plausible lifecycle model is subjective. Although we may use abstract theory to generate forms of design lifecycle we must then present evidence that relates these to lifecycles accepted as standard in particular domains. We also need to show that tool support is possible for the design stages we invent, since a key reason for formality in design is automation and this requires that tools can be built.

It is impossible to explore the entire space of possible lifecycles, even for a single computational logic, so this thesis considers two contrasting approaches. To make the comparison between approaches more direct the specifications constructed in each are in the same formal notation - Horn clauses - and are executed as Prolog programs.

The first approach uses a logic to provide a standard description language for partially completed specifications and uses this language to communicate between a number of self-contained design tools in a system called LSS. This provides a loosely constrained design process where the decisions about when and where each tool is used are not prescribed by the formal framework. As an example of this, imagine that we are building a specification for a patient processing system. Some parts of this, such as the overall sequencing of stages in processing, might conveniently be represented as a block and arrow diagram. Other stages, such as the general diagnostic procedure, might best be visualised as a way of allocating patients to particular diagnostic categories. Other stages might be viewed in a more algorithmic style. The problem of coordinating these

different views is tackled by translating each into a shared formal language. We work through this sort of example in detail with LSS in Chapter 6.

The second approach takes the opposite view, where specifications in a logic are developed using a framework (supported by a tool named HANSEL) which determines when and where particular design decisions are made. This sort of framework is best applied when we have a form of high level description for early design which covers the entire problem and which we can use to structure refinement - essentially top-down design. An example from the knowledge engineering world is in task modelling where we begin with a high level, overall description of a standard task (such as a style of diagnosis) and progressively flesh this out. We look at examples from one such method using the HANSEL system in Chapter 12.

A note on terminology. In this thesis we use the word "specification" to refer to the sets of Horn clauses constructed by our design systems. Often these specifications are executable but we do not refer to them as "programs" because at some design stages (particularly in HANSEL) they may be far from what is normally considered an executable program. This is different from the use of "specification" in deductive synthesis (see Section 2.1), where the "specification" is a description of the property desired of a program and a program is constructed deductively from that specification. In the HANSEL system we make use of properties of our sets of Horn clauses but we refer to these as "properties of a specification" rather than "a specification of a program" because our use of properties is not for deductive synthesis and the clauses to which they apply are not always executable programs. The purpose of our specifications is usually to describe idealised models of systems or processes. When doing this, we sometimes talk about the "model", meaning our specification of the model. In Section 2.4.3 we encounter yet another use of "specification": in providing domain-specific descriptions of problems which are used to guide the synthesis of specifications appropriate to these problems. To emphasise this purpose we sometimes refer to this form of specification as a "problem description".

The structure of the main thesis follows the chronology of the research. Part II describes the first, diverse style of design, embodied in the LSS toolset. In it, we first introduce the approach (Chapter 3); then define the formal concepts needed by the design tools (Chapter 4); then describe the tools which were built to perform this sort of design using the given formal concepts (Chapter 5); then give a worked example to explain how the tools work together (Chapter 6); finally evaluating the toolset by bringing in earlier evaluations on related tools and performing a case study using LSS to reconstruct a substantial logic program. The evaluation reveals a number of problems with the highly distributed style of design in LSS - stimulating investigation into the second, more rigidly structured approach of the HANSEL system in Part III. The structure of this part follows that of Part II: an introduction (Chapter 8) followed by formal concepts (Chapter 9) then an overview of the HANSEL tools (Chapter 10) then a worked example which revisits an specification developed with LSS (Chapter 11) and finally an evaluation(Chapter 12) in which early design models from an es-

tablished knowledge-based systems design method are compared to analogous specifications developed in HANSEL.

The principal questions addressed in this thesis, from the perspective of logic programming, are:

- Is a Horn clause language supplemented by techniques descriptions (obtained through direct manipulation and structure editing) an effective means of sharing design knowledge in a loosely coordinated design framework ?
- We know from earlier research [Robertson et al., 1994] that close links can be made between set-based refinement and logic programming. Can we harness these links to give a methodical system of refinement which is inspired by logic programming.
- The point of coordination in specification design is to obtain design processes which parallel those of conventional software engineering. One of the key elements of this process is that commitments made in design phases are checked in subsequent verification phases. Can the use of formal refinement in design give routes for easily obtaining some kinds of verification information?
- Can we invent tools which assist in design in the styles described above? In particular, can we reconcile the tension between the diversity of description styles and forms of automation found in a system like LSS with the uniformity of method and automation found in systems like HANSEL.

In Chapter 13 we return to these questions when summarising the contributions of this thesis.

Chapter 2

Related Work

One of the contributions of this thesis is to pull together a number of strands of research. This means that the amount of related work is potentially very large. To bound the discussion, we explore each strand starting close to the work of the thesis and charting the area around this, with sparse references to more distantly related research. In [Robertson and Agusti, 1999] we give a broad-ranging exploration of the variety of methods and tools related to this research.

2.1 Synthesis of Logic Programs from Specifications

The majority of research on synthesis of logic programs assumes that an initial specification is provided and the task is then to build an appropriate program for it. For this to be of practical value it has to be more effective in the chosen domain for people to conceive initial specifications (and then derive programs) than it is to design their programs by conventional means. Experience from within the algebraic specification community suggests that there are serious problems in arranging for this to happen, even for systems embodying powerful notions of refinement. To quote [Sannella and Tarlecki, ming] talking about their experiences with Extended ML:

> "Actually writing specifications of programs that use higher-order functions, exceptions, partial functions, *etc.* is difficult and the resulting specifications are hard to read. Worse, the specifications of simple total first-order functions need to include boring conditions that rule out exceptions and partiality. Finally, actually developing programs from specifications is hard work.

This thesis is part of the minority of work currently exploring answers to these sorts of problems relevant to logic programming. It is therefore intended to be complementary to the majority of work in synthesis of logic programs from

specifications which we summarise below, taking [Deville and Lau, 1994] as our guide:

- Constructive synthesis: builds a program for a specification by performing a constructive proof of the specification and extracting from that proof a program for computing the relations contained in the specification.
- Deductive synthesis: derives a program as a logical consequence of a specification by applying logically sound deduction rules to it, the correctness of the resultant program with respect to the specification being established through the correctness of the deduction rules.
- Inductive synthesis: derives a program from incomplete information, typically in the form of examples. The aim is to produce a compact program which covers the examples in a way which is "natural" with respect to some acceptance criteria.
- Structural synthesis[1]: builds a program from a collection of schematic components, giving particular design strategies which are expressed as structural changes to the program. The emphasis here differs from that of the approaches above in that the choice of schematic component is often a pragmatic one, tempered by what is acceptable in a domain of application (see [Robertson and Agusti, 1998]).

The method of design used in this thesis is structural synthesis.

2.2 Views of Lifecycles

One of the reasons why it is difficult to express specifications in logic is that it is both abstract and flexible, so there are many different ways of describing similar problems. This is also true for traditional programming languages and for the sorts of knowledge representation languages used in the knowledge-based systems community. These communities, however, have developed numerous methods for assisting in the design of software in important domains and/or for standard classes of task. Usually these promote a particular view of the design lifecycle, starting from early requirements or architectural features and perhaps extending through to verification, validation and maintenance. This broader view of design, essential in traditional software engineering but largely ignored in logic programming, is important to this thesis, so we introduce it here.

A vast literature exists on software lifecycles but, for simplicity, we give only examples directly relevant to the remainder of the thesis. First, an example of a standard lifecycle model from mainstream software engineering is given (Section 2.2.1); then a design method from the knowledge engineering community is compared to it (Section 2.2.2); then we introduce the notion of distributed specification central to the LSS system (Section 2.2.3); and finally discuss the use of logic programs in testing (Section 2.3).

[1] Classified under "Informal Methods" in [Deville and Lau, 1994]

2.2.1 A Lifecycle from Software Engineering

There is no single, standard lifecycle model for software design but one which is common is the "V" model, shown in Figure 2.1. This is used in software design standards such as IEC 61508 *Functional Safety: Safety Related Systems* which is a new European standard for design of programmable electronic systems used in safety-critical systems. The basic idea is that design commitments are made (and documented) on the way down the left arm of the "V" and these commitments are scrutinised on the way up the right arm of the "V" (the dotted lines on the figure illustrate this transfer of information from design to validation and verification).

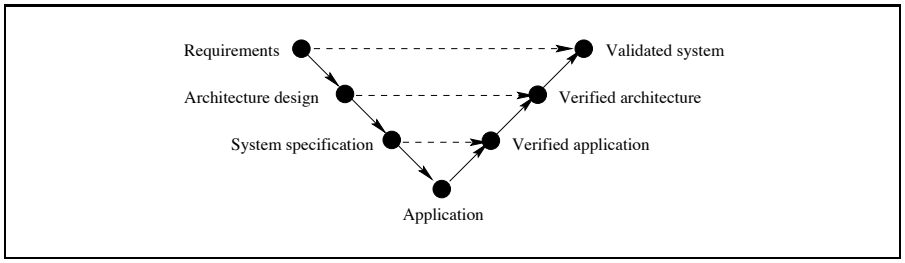


Figure 2.1: A "V" lifecycle model

The design stages in Figure 2.1 commence by establishing requirements for the system to be built - documenting these in a form which can be validated once the system is completed. In the next design phase the architecture of the system is described, which commits designers to the general style of problem solving that the system will exhibit and which will later be verified to establish that this style has been followed appropriately. The system specification is then written in a way which can be verified with respect to the application (through some mixture of testing, proof or informal argument). The human process of following this sort of lifecycle can be complex, with many of the stages being revisited as requirements shift and lessons are learned from early implementations of an application. The purpose of a lifecycle model is to provide a framework for keeping this form of complex design process under control.

2.2.2 A Design Method from Knowledge Engineering

Lifecycle models are traditionally associated with mainstream software engineering. By contrast, knowledge engineering has been slow to develop analogous form of control on design processes. The reasons for this are a matter of debate. One reason might be that knowledge engineering is a newer form of design than software engineering, so strong lifecycle models have not had time to mature. Another reason might be that the domains to which knowledge engineering is applied are often loosely constrained and uncertain, so the style of design is often driven by prototyping and experimental programming which can be carefully or-

ganised but often are not. A third reason might be that many knowledge based systems are small by software engineering standards and it is therefore possible to concentrate on the artifact being built rather than on the process through which it is engineered.

Whatever the causes, the current situation is that the *de facto* only commonly used lifecycle models for design of knowledge based systems are those of the KADS method [Wielinga et al., 1992a, Schreiber et al., 1993, Tansley and Hayball, 1993, Wielinga et al., 1992b]. Many subtly different interpretations of this method exist but the main method involves four layers of abstraction. At the lowest level, the domain layer contains concepts and relations specific to the domain of application. Above this, the inference layer gives a declarative description of the problem solving processes used. On top of that, the task layer provides a procedural interpretation of the inference layer. Finally, the strategy layer connects together the problem solving tasks. Central to this arrangement is a library of interpretation models, which provide configurations of inference functions and knowledge roles appropriate to particular types of task. Once an interpretation model is identified, it is then adapted and instantiated in order create the inference layer of the KADS model. To assist users in classifying interpretation models by task, a hierarchy of tasks is provided. The leaf nodes represent particular interpretation models, while other nodes divide the interpretation models into subsets of related tasks.

One of the interpretation models is an assessment task. This is shown in Figure 2.2 in the diagrammatic style normally used in KADS for these models. The boxes in this diagram represent knowledge roles: types of knowledge which are the inputs and/or outputs to inference functions (shown as ovals). This diagram is intended to be matched against the task being performed in some particular domain of application, with appropriate instantiations and (if necessary) adjustments being made to suit the problem in hand. Suppose, for example, that our task was to write an expert system for assessing the suitability of applicants for a job. We could match this task to the diagram in Figure 2.2 as follows:

- The case description matches to the applicants CV.
- The abstract case description is then a description of the key abilities and weaknesses of the applicant (abstracted from the case description).
- The ideal case is a stereotypical "ideal" CV.
- In the context of which we specify, as the norm, the key abilities required for the job.
- The abstract case description is then matched against the norm to obtain a comparison of the candidates abilities against those ideally required for the job, yielding a decision class.

The instantiated interpretation model for assessment is shown in Figure 2.3.

We shall return to this example in Section 12.1 as part of the evaluation of the HANSEL system.

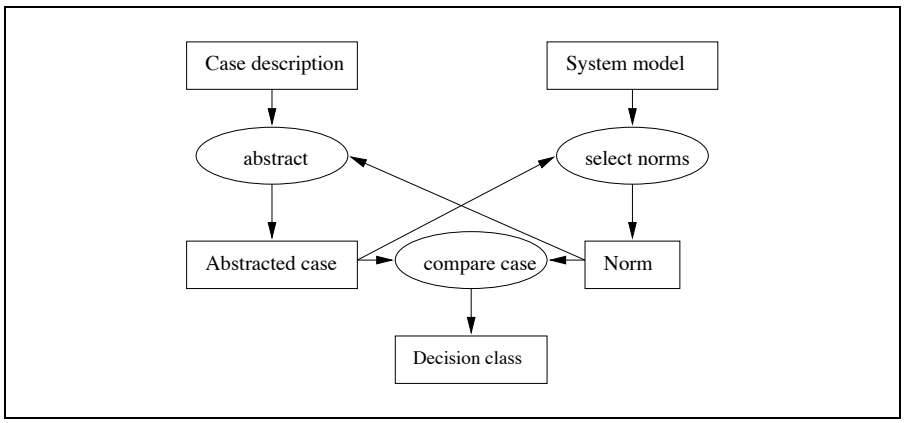


Figure 2.2: KADS inference structure diagram for an assessment task(adapted from [Tansley and Hayball, 1993] page 294).

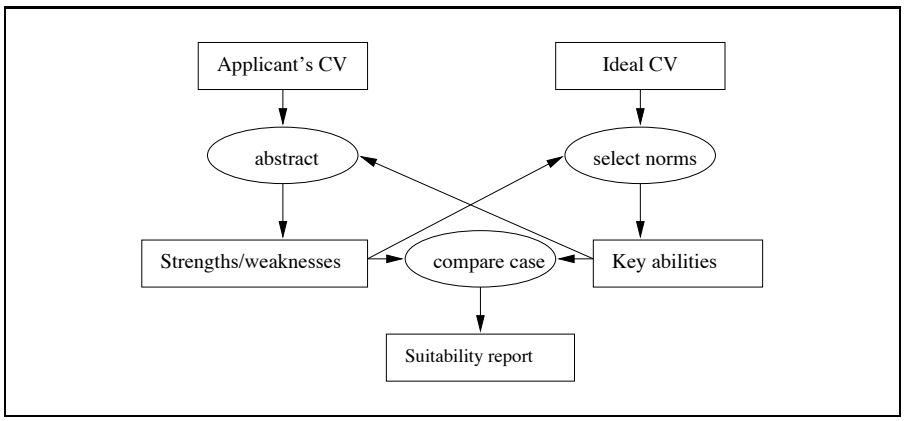


Figure 2.3: KADS assessment model instantiated to a job application.

There have been attempts to build automated design systems for part of the KADS lifecycle or for lifecycles similar to that of KADS. The $(ML)^2$ system [van Harmelen and Balder, 1992] used different order-sorted logic to describe the domain specific component of models; related this to standard forms of inference by a meta-logic (in predicate calculus) describing how the domain knowledge might be used; and described how tasks should be performed using dynamic logic. A subset of the full system was mechanised, essentially by restriction to Horn clauses, so it is possible sometimes to obtain executable specifications from this system. The MIKE system [Angle et al., 1998] takes a different view by taking inspiration from the layered style of design in KADS but using its own (related) style of specification which makes it easier to produce executable specifications. Essentially, MIKE design begins by drawing an initial model containing nodes and links similar to a KADS interpretation model; the next level allows inference methods, described in a logic, to be introduced for the components of the structural model; then to these inference methods are added definitions of the algorithms and data structures required to execute them. Both $(ML)^2$ and MIKE move away from logic programming in an attempt to tame the difficulties of early design through more sophisticated representation languages. However, it is not clear that this route makes for easier design because, even if it is in theory possible to describe knowledge engineering problems elegantly in such languages, designers must understand the more sophisticated languages in order to choose appropriate representations. In this thesis we attempt to remain close to Horn clauses for specification, although in the HANSEL system we extend this through set-based refinement. In Chapter 12 we shall return to KADS viewed from the perspective of HANSEL and shall draw similarities between some KADS models and HANSEL specifications.

2.2.3 Distributed Tasks United by a Shared Language

The LSS system of Chapters 3 to 7 was stimulated by the Explore system described in [Fuchs and Fromherz, 1994]. Both Explore and LSS allow different parts of a specification to be described using different tools, some graphical and some textual. The LSS system makes more use of techniques editing as a way of interchanging partially described specifications, where in Explore the interchange language is the Explore/L language which is an object-oriented variant of Prolog. The Explore system makes more use of schema based transformations on completed specifications to turn them into efficiently executable programs - LSS pays little attention to this design stage in order to avoid repeating the work of Explore.

2.3 Testing

Although this thesis is not about software testing one of the advantages of the use of formality in design lifecycles is that information relevant to testing can be carried automatically from early design stages to later ones (see Sections 9.3

and 9.7). The way in which we currently use this information is comparatively straightforward: test instances are obtained from executing the specification and it is left to the person doing the testing to determine whether these tests succeed, either by calling them as normal Prolog goals or by other means. Much more sophisticated forms of analysis exist for logic programs. Two of relevance to this thesis are:

- Verification systems which can be applied directly to logic programs and derive information relevant to testing. A recent example, given in [Le Charlier et al., 1999], performs a variety of analyses for features such as modes, types and termination.
- Systems which give declarative descriptions of testing methods and thus help to automate the process of testing. An example appears in [Gorlick et al., 1990], where a method is described for using logic programs to specify tests. These test specifications can be run in two "directions"; either as a generator of tests which conform to the specification or as a validator that the specification meets a given test.

This thesis is complementary to forms of analysis like those above because the method of accumulating properties during design, described in Sections 9.3 and 9.7, gives a way of obtaining information during design which would not always be derivable by analysis of the logic program and which might be useful in testing regimes. If one agrees with [Perry, 1995] that "the biggest problem with testing is boredom" then it seems useful to build mechanisms for acquiring testing information into automated design tools because this helps reduce the tedium of defining test goals by hand for every predicate. On the other hand, we cannot expect to acquire all appropriate tests in this way so our claim to assist in this area is modest.

2.4 Systems of Formal Design for Specifications

Many of the tools described in this thesis design specifications directly, either by editing a specification directly in its basic formal language or by editing graphical symbols with direct correspondence to the underlying formal language or by using a pseudo-English translation of the formal language. This section describes other research in this area, concentrating on logic programming editors.

2.4.1 Design Based on Structure Editing

A structure editor is a design tool which allows specifications (or programs) to be built directly in the formal specification language using syntactic components appropriate to that language. The advantages of a structure editor are that these components may be large, saving time in adding them, and if they are well chosen then they may easily be selected and combined. The LSS techniques editor (Section 5.1) and the HANSEL specification window (Section 10.1) are structure editors.

Within logic programming there has been a variety of attempts to produce structure editors, each of which makes different choices of structural component and editing style. These systems can be split into those which are based on maintaining certain properties throughout editing (a minority) and those which don't (the majority). The LSS and HANSEL systems do not guarantee properties of specifications during editing, although HANSEL generates properties for retrospective testing.

An example of a structure editor intended to guarantee termination of recursive specifications is the Recursion Editor [Bundy et al., 1991]. This starts from a basic recursion schema and extends this by applying edit commands which perform structural changes such as adding an additional argument to the predicate. Each of these changes adds appropriate subgoals and additional cases to ensure that the predicate being constructed will be terminating when it is completed. The price paid for this guarantee is that only a limited class of logic programs can be described. It is also claimed in [Whittle, 1998] that the way in which termination is guaranteed, through guaranteed correct transformation rules, makes it difficult to extend the system by adding new transformations because the potential interactions with earlier transformations must be considered each time a new editing operation is added. An antidote to this is to maintain a proof of termination during the editing and to have the editing operations manipulate this proof, with the corresponding program being extracted from it. This is the approach used in the $C^{Y}NTHIA$ editor [Whittle, 1998] which is a descendant of the Recursion Editor but constructs functional programs (in ML) rather than logic programs. To our knowledge, neither the Recursion Editor nor $C^{Y}NTHIA$ has been applied to significantly sized applications, although $C^{Y}NTHIA$ was tested on a sample of four novice ML programmers. Similarly to the experiments with techniques editing reported in Section 7.1.1, these novices made fewer errors when using the structure editor. The termination checking was reported to "have only a limited benefit in program writing" but this may have been because of the limited scope of the experiments.

Many of the more general purpose logic program structure editors, which do not guarantee properties during editing, were inspired by the notion of stepwise enhancement originating in Sterling's group [Kirschenbaum et al., 1989, Sterling and Kirschenbaum, 1993]. The essence of this method is described in Section 4.1. The early work (such as [Kirschenbaum et al., 1989]) concentrated more on the benefits of fixing the flow of control (via choice of skeleton) prior to transformations which were guaranteed not to disturb the flow of control, thus making certain forms of predicate combination easier. Later work gave greater emphasis to design method (see [Naish and Sterling, 1997] for example). As the basic principle became more widely applied it was adapted for other purposes and tested in application domains. Some examples of this are:

- In [Plummer, 1990] a method of designing Prolog programs from parameterisable structures called "cliches" is described. The basic idea is that a library of template definitions is provided and by instantiating these variables we get refinements of the templates. In this sense, the idea is

a version of the notion of parameterised programming [Goguen, 1989]. In practice the choice of cliches is quite close to those in many techniques editors.

- Gegg-Harrison provided a classification of "schemata" (similar to skeletons) for Prolog list processing predicates [Gegg-Harrison, 1989] and describes a method for using these to introduce recursion to novice programmers and for guided debugging [Gegg-Harrison, 1991].

- A simple techniques editor for novice programmers was built by the author [Robertson, 1991]. This is of historical interest only, since it was an ancestor of the TEd and LSS editor.

- The TEd editor [Bowles et al., 1994, Bowles and Brna, 1993]) was used in an extensive evaluation of techniques editing with novice programmers, summarised in Section 7.1.1. This has libraries of skeletons and extensions which are simpler than many other such editors (including those described in this thesis) but had a comparatively sophisticated interface targeted at supporting novice programmers. In its final version, this included an analogical mechanism for relating problem features to solved examples from a case library and using these to offer appropriately instantiated skeletons.

- The TeMS system [Castro, 1999] is a highly domain-specific implementation of a techniques editor which assists ecological experts in the construction of a class of population dynamics models. It is not strictly a structure editor because the ecologists describe problem features without editing the specification of a population model directly. It is included here to make the point that when such systems become domain-specific they may move away from direct structured editing to a more layered form of specification in which the language of initial problem description differs from that of the final specification (see Section 2.4.3).

2.4.2 Design Based on Direct-Manipulation Graphical Interfaces

Structure editing, as described in the previous section, manipulates specifications directly in the textual formal language of the specification. It is, however, possible to manipulate specifications through diagrammatic representations which have a direct correspondence to the textual formal language of the specification. We refer to these as direct-manipulation graphical interfaces. In some cases this correspondence can be tight in the sense that whatever is drawn in the diagrammatic language has a corresponding, unique translation into the textual formal language. This is the case for the LSS Process Editor (Section 5.2) and Recursion Editor (Section 5.3). In other cases the correspondence may be looser in the sense that only diagrams which have reached some defined stage of completion are translatable to the textual formal language. We know that it is possible to build direct-manipulation graphical interfaces to generic forms of logic (this is

done for Horn clauses in [Agusti et al., 1998]) but it is more common in practice to build such interfaces for narrower subsets of a formal language, with the chosen subset often determined by the task or the domain of application.

The use of this style of formal description is now common (see [Stasko et al., 1998] for a compilation of some recent research) but the fact that most systems are task or domain-specific makes it difficult to give a definitive guide to the area. Within logic programming itself graphical interfaces have (arguably) been more deeply explored in debugging programs (most notably [Eisenstadt and Brayshaw, 1988]) than in designing them. A notable exception is the Explore system [Fuchs and Fromherz, 1994] (introduced in Section 2.2.3) which provided a number of direct-manipulation graphical editors (for example for state transition networks and for user interface specification) that translated to a specification in a specification language derived from Prolog. The graphical tools in Explore went beyond those in LSS in providing a bi-directional mapping between diagram and specification, rather than a uni-directional mapping from diagram to specification. In principle, bi-directional mappings should be possible in LSS but we have not explored this sufficiently to be sure whether the benefit gained from this sort of flexibility is worth the additional cost in developing the mappings.

2.4.3 Design Separating Problem Description from Specification

The design systems of Sections 2.4.1 and 2.4.2 rely on human designers being able to describe their problems directly in a language (graphical or textual) which gives an adequate specification of the solution to it. Sometimes this may not be practical because the designers do not have sufficient background in the specification language to be able to describe their problem in those terms. A solution to this problem is to give designers a simpler, often domain-specific, language in which to describe their problems and use this to control the generation of an appropriate specification. In a sense, the early stages in HANSEL definition are like this because they begin with definitions using set inequalities, which might be viewed as an early problem description language, and these are refined into Horn clause specifications. There are, however, other ways of obtaining a separation between problem description and specification.

- A domain-specific formal language can be used for problem description, with domain-specific synthesis being used to connect this to structural synthesis of specifications. One of the earliest examples of this was the MECHO system [Bundy et al., 1979], which constructed a domain-specific problem description by parsing example mechanics problems stated in natural language and used this to select and instantiate a set of appropriate equations which were then passed to an equation solver for solving the problem. This performed well on the very narrow range of mechanics problems upon which it was targeted but was not extended beyond these. At around the same time experiments were being conducted in domain-specific synthesis of programs, an early landmark being [Barstow et al., 1982]. A

descendant of MECO, the ECO system [Robertson et al., 1991], reformulated the basic idea of MECO to more general program synthesis. The ECO problem description language was a simple form of sorted logic which was manipulated by selecting domain-specific templates (the domain was a branch of ecological modelling) and using these to guide the construction of logic programs describing ecological simulation models.

- The problem description can be viewed as a specification for which an appropriate program is generated through deductive proof. A recent example of this is the Amphion system [Lowry et al., 1994, Lowry and Van Baalen, 1997] which has been used, in different incarnations, to generate programs for solving domain-specific problems such as solar geometry. The basic idea in this type of system is that problems, described via a domain-specific graphical interface, are represented as specifications of the form $\forall I.\exists O.C_1 \wedge \ldots \wedge C_n$, where each C is a predicate application or an equality defining a variable through a function application. A deductive theorem prover is used to generate an application program for the problem specification, using domain-specific proof tactics to control the search. Although Amphion is driven by deductive synthesis, so strictly it has no place in our survey of structured synthesis systems, we justify including it here as an example of similar attitudes to problem description being adapted to different methods of synthesis.

Both of the solutions described above rely for their effectiveness on domain-specificity. This makes it possible to acquire problem descriptions more readily from domain experts (via domain-specific interfaces) and allows the problem of automated generation of a solution from a distantly related problem description to be tamed by using domain-specific generation methods. A more detailed discussion of domain-specific specification methods appears in [Robertson, 1996b] and a more detailed summary of some domain-specific declarative specification systems appears in [Fuchs and Robertson, 1996].

2.5 Set-Based Refinement for Logic Programs

The early stages of design in the HANSEL system of Chapters 8 to 12 rely on the use of specifications expressed as logic programs relating sets of axioms via set inequalities between the theories represented by those axioms. This is introduced technically in Section 9.1, and developed further in Section 9.2. Central to this approach is the idea that many forms of problem solving can be viewed as manipulations of theories. One way of describing this is by expressing early specifications using Horn clauses where variables refer to axiom sets and set inequalities in the bodies of the clauses place constraints on the theories of those axiom sets. For example, the clause $p(S1, S2) \leftarrow \tau(S1) \supseteq \tau(S2)$ says that the relation p holds between axiom sets $S1$ and $S2$ if the theory of $S1$ includes that of $S2$. The mechanism of refinement used in the early stages of HANSEL design uses rewrites on these inequality subgoals to add detail to the specification

- so in the example above we would refine $\tau(S1) \supseteq \tau(S2)$ but would not alter $p(S1, S2)$, except by the addition of arguments.

An alternative, more radical, view is to represent the Horn clauses themselves as inequalities and to view synthesis as set-based refinement of whole clauses (not just their subgoals). This provides a highly flexible form of refinement (see [Levy, 1994] for a generic rewriting system and theoretical results) and extends other expressive styles of specification, such as Lambda-calculus (see [Levy et al., 1991] for an example of this sort of extension). It is this flexibility and expressive power, however, which has made such languages difficult to harness for realistically sized synthesis tasks because it appears to be difficult for those who have not been immersed in the mathematics of the languages to adapt to this style of specification. In our earlier work [Robertson et al., 1994] we defined a simplified refinement language following some of the ideas of [Levy, 1994] but with a final translation from our inequality expressions to standard Horn clauses. The basic idea is that the Prolog clauses are obtained by a translation from inequalities between sets of results obtainable from given argument positions in predicates. For example, the inequality:

$$D : \mathit{diagnosis}(S, D) \supseteq D1 : \mathit{diagnosis}(S1 : \mathit{hypothesise}(S, S1), D1)$$

might be used to denote that the set of results for the diagnosis, D, of diseases based on given symptoms, S, includes the set of diagnoses, $D1$, from hypothesising on those symptoms. This translates to the Horn clause:

$$\mathit{diagnosis}(S, D) \leftarrow \mathit{hypothesise}(S, S1) \wedge \mathit{diagnosis}(S1, D)$$

In [Robertson et al., 1994] we use the inequality definitions to supply a refinement lattice within which designers may navigate and select inequalities at levels of detail appropriate to their problem. These are then translated to Horn clauses. It would, in theory, be possible to combine this notion of refinement (or related notions of refinement expressed using higher order functions as described in, for example, [Uschold, 1990, Naish and Sterling, 1997]) with the refinement of inequalities between theories currently used in HANSEL. In practice, this introduces yet another level of mathematical sophistication into a framework which is already highly demanding of designers. It is not clear whether the combination of refinement methods would simplify or complicate matters.

This concludes our discussion of related work. We have introduced the four main approaches to synthesis of logic programs and identified the one closest to the work of this thesis: structural synthesis. We then explained why some notion of lifecycle is necessary in synthesis and described three contrasting views of lifecycles. The third of these, based on distributed tasks united by a shared language, is the motivation for the work described in the next part of this thesis. One benefit of having a lifecycle is that information may be obtained automatically in early stages of design with the intention of using it later, for example in testing. We briefly summarised some systems which perform testing in logic programming. We then looked more closely at systems for designing specifications, concentrating on those using structural synthesis. Finally, we introduced

the idea of set-based refinement which will feature later in this thesis, comparing it to a related form of refinement used previously by the author and others.

Part II

A Solution Allowing Diversity

Chapter 3

Overview of the LSS Distributed Design Model

The aim of LSS is to allow diverse styles of description to be used when representing different parts of a problem. This is done by constructing editing tools which are targeted at particular styles of description and have interfaces which reinforce that style. The tools do not interact directly with each other. Instead, they communicate through a shared formal language. Thus, one tool may construct a partial description and save it in the shared language, with this description then being read by another tool which may extend or refine it. This has the practical advantage that new tools can be added to LSS without the cost of building interfaces between them and the existing tools. This motivation is similar to that for the use of interlinguas in knowledge sharing (see [Uschold and Gruninger, 1996] for an overview).

The tools currently in LSS can be split into three groups:

- Those concerned with designing new predicates in different styles. The current set of these is:

  The techniques editor : describes predicates in a style which requires the partial program describing the flow of control first to be selected and then for it to be extended one argument at a time. This is described in detail in Sections 4.1 and 5.1.

  The process editor : allows designers to edit diagrams representing process sequences in a box and arrow notation and automatically converts these into Definite Clause Grammar (DCG) specifications, which in turn have a direct translation to normal Horn clauses. This is described in detail in Sections 4.2 and 5.2.

  The diagrammatic recursion editor : gives a diagrammatic style of description for partial definition of simple recursive predicates, which can be refined using the techniques editor. This is described in detail in Section 5.3.

The window editor : allows designers to "sketch" visual descriptions of windows and their components (edit fields, buttons, *etc.* and annotate these with links showing parts of the interaction between them (*e.g.* that a button click in one window creates another window). This description is then translated automatically into a partial Prolog definition of the interface predicates, which can be refined using the techniques editor.

- Those which transform existing predicate definitions into equivalent definitions which are more compact and/or more efficient computationally. Only one early prototype tool is in this group:

  The transformation editor : gives a menu of basic forms of correctness preserving transformations which the designer can apply to the predicates constructed by other editors.

- Those which provide a commentary on specifications which have been built by one or more of the other tools. The current set of these are:

  The overview editor : displays the call graph for a specification - nodes being predicate names and links connecting each predicate to the other predicates which it calls as subgoals.

  The argument editor : is a graphical editor for a semantic network in which the links are labelled from a small set of argument primitives (*e.g.* "supports" or "objects to") in the style of IBIS argumentation nets [Conklin and Begeman, 1988]. Some of the nodes in the network can point to predicate definitions, allowing the argument network to be used to describe part of the debate underlying the design of selected predicates. This is similar to the editor described in [Ramesh and Luqi, 1993].

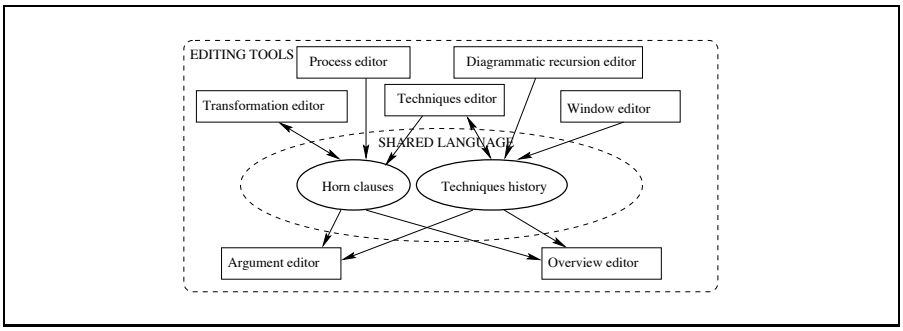


Figure 3.1: The LSS design model

More detailed overviews of LSS can be found in [Robertson, 1996a], which explains the idea of this form of distributed design, and in [Robertson, 1998a],

which describes some of the empirical testing of the system (summarised in Section 7.2. This thesis concentrates on the design of new specifications rather than on correctness preserving transformations or retrospective commentaries on designs so in the following chapters we focus our attention on this group of tools. We also wish to avoid lengthy discussion of the window editor because it does not go significantly beyond the capabilities available in commercially available dialogue generators, other than by its ability to produce specifications which can be refined by a techniques editor. Therefore Chapters 4, 5 and 6 are confined to discussing the techniques editor, process editor and diagrammatic recursion editor.

Chapter 4

Formal Concepts

The LSS toolset uses as its language of interchange between tools either standard Horn clauses or Horn clauses of the sort used in a form of techniques editing, resembling normal Horn clauses but with some additional varieties of subgoal to allow partial definitions of predicates. This is described in Section 4.1. All of the tools produce restricted classes of Horn clause definitions, as described in Chapter 5. One of the tools does this via Definite Clause Grammar notation, standard with most Prolog systems, and in Section 4.2 we explain how this relates to the description of a particular kind of process model.

4.1 Skeletons and Extensions

The basic idea of skeletons and extensions in logic programming comes from the observation that many designers think of part of the predicate as being responsible for the flow of control of the program during execution, with other parts building on top of this. The former is the skeleton of the predicate and the latter are extensions of that skeleton. For example, a predicate $count(L, C)$ which finds the cardinality, C, of list, L might be defined as:

$$count([], 0) \tag{4.1}$$

$$count([H|T], N) \leftarrow count(T, Nt) \wedge N \; is \; Nt + 1 \tag{4.2}$$

The skeleton for this is:

$$count([]) \tag{4.3}$$

$$count([H|T]) \leftarrow count(T) \tag{4.4}$$

and the extension is a transformation which adds the second argument as a counter:

$$\begin{array}{l} count([]) \\ count([H|T]) \leftarrow count(T) \end{array} \Rightarrow \begin{array}{l} count([], 0) \\ count([H|T], N) \leftarrow count(T, Nt) \wedge N \; is \; Nt + 1 \end{array} \tag{4.5}$$

We can generalise these notions by using partially defined predicates where the identity of the subgoals which test or update variables is undefined. Instead the appropriate subgoals contain placeholders which are of the form:

- $\mathcal{T}(X)$ if a test should be performed on variable X.
- $\mathcal{U}(X, Y)$ if variable Y is derived from variable X.

In our cardinality example, the general skeleton corresponding to clauses 4.3 and 4.4 might be:

$$P(X) \leftarrow \mathcal{T}(X) \tag{4.6}$$
$$P([H|T]) \leftarrow P(T) \tag{4.7}$$

and the general transformation corresponding to expression 4.5 might be the following, where each ϕ_i is a possibly empty conjunction of subgoals.

$$\begin{array}{lcl} P(X) & \leftarrow & \phi_1 \\ P(G1) & \leftarrow & \phi_2 P(G2)\phi_3 \end{array} \quad \Rightarrow \quad \begin{array}{lcl} P(X,F) & \leftarrow & \phi_1 \wedge \mathcal{T}(F) \\ P(G1,N) & \leftarrow & \phi_2 P(G2,Nt) \wedge \mathcal{U}(Nt,N)\phi_3 \end{array} \tag{4.8}$$

This gives a way of structuring the design of predicates. We first select and instantiate a skeleton. In the example, we would select the skeleton given by clauses 4.6 and 4.7 and instantiate P to *count* and $\mathcal{T}(X)$ to $X = []$. We then apply the extension defined in expression 4.8 to this skeleton giving the transformation:

$$\begin{array}{c} count(X) \leftarrow X = [] \\ count([H|T]) \leftarrow count(T) \end{array} \quad \Rightarrow \quad \begin{array}{c} count(X,F) \leftarrow X = [] \wedge \mathcal{T}(F) \\ count([H|T],N) \leftarrow count(T,Nt) \wedge \mathcal{U}(Nt,N) \end{array} \tag{4.9}$$

and finally we instantiate $\mathcal{T}(F)$ to $F = 0$ and $\mathcal{U}(Nt, N)$ to N *is* $Nt+1$, giving the equivalent of clauses 4.1 and 4.2. This style of construction is the basis for the LSS techniques editing tool described in Section 5.1 and is related to the use of HANSEL skeletons and argument slices in Section 9.5. A more lengthy introduction appears in Chapter 6 of [Robertson and Agusti, 1999].

4.2 Definite Clause Grammar Related to Processes

One of the aims of LSS is to provide tools for constructing restricted classes of logic programs, using design paradigms that originate in logic programming but which can be interpreted in terms of a domain-specific task. An example of this is where the task is to represent a process consisting of alternative sequences of steps which can be composed hierarchically. As we go through the sequence we may observe certain events that happen as a consequence of the process. for instance, in defining order processing in a business model we might want to represent two alternative sequences:

- An invoice is raised for the customer but subsequently cancelled; or
- An invoice is raised for the customer and subsequently filled.

and be able to report on completion of either one of these sequences that the appropriate events had taken place.

There is a natural parallel between this sort of task and Definite Clause Grammar (DCG) notation, with each sequence being represented as the parsing steps in the body of a DCG rule and the observed events being the elements of the sequence generated by the DCG. Our invoice processing example might then be represented as:

$$\begin{aligned} process\_invoice(Customer) \quad \Rightarrow \quad & raise\_invoice(Customer, N), \\ & [invoice\_raised(Customer, N)], \\ & cancel\_invoice(Customer, N), \\ & [invoice\_cancelled(Customer, N)] \end{aligned} \tag{4.10}$$

$$\begin{aligned} process\_invoice(Customer) \quad \Rightarrow \quad & raise\_invoice(Customer, N), \\ & [invoice\_raised(Customer, N)], \\ & fill\_invoice(Customer, N), \\ & [invoice\_filled(Customer, N)] \end{aligned} \tag{4.11}$$

where $process\_invoice(Customer)$ is the DCG for a sequence of events observed in processing an invoice for a $Customer$; $raise\_invoice(Customer, N)$ is the sequence for raising an invoice numbered N; $cancel\_invoice(Customer, N)$ is the sequence for cancelling order N; and $fill\_invoice(Customer, N)$ is the sequence for filling invoice N. We can also place conditions on the execution of parts of the sequence by enclosing them within curly braces, as in standard DCGs. This direct analogy between non-deterministic sequential processes and DCGs is the basis for the visual interface of the LSS process editor of Section 5.2.

This completes our discussion of the key formal concepts underpinning the LSS system. In the next chapter we introduce the tools which enable designers to use these formal concepts in describing specifications.

Chapter 5

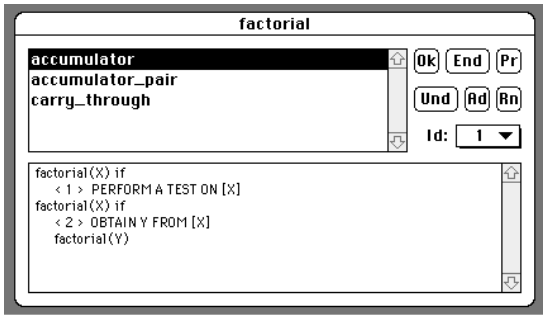
Tools Based on Different Paradigms

Chapter 3 gave an overview of the LSS toolset and Chapter 4 introduced the formal concepts, beyond those of conventional logic programming, used by the system. We now look in more detail at three of the editors used in LSS explaining how the visual interface provided by the tool enables predicates to be designed in the formal styles described in chapter 4. We make the examples in this chapter as simple as possible so that they convey the essence of each editor. Chapter 6 gives a larger example in which the editors share information in order to describe a larger problem.

5.1 Techniques Editor

Section 4.1 described the basic idea of techniques editing. Appendix B gives the library of skeletons currently used in LSS. There is no consensus on the ideal set of skeletons (see Section 2.4.1 for some examples) because the choice of these depends on the task in hand. The current library has, for flexibility, a set of skeletons for general forms of recursion (Appendix B.2) and, for ease in describing commonly occurring special cases, sets of skeletons for recursion on lists (Appendix B.3) and for recursion using arithmetic counters (Appendix B.4). We now show how the techniques library is applied to build a basic predicate, $factorial(X, F)$, which describes a recursive calculation of the factorial, F for a given number, X. The flow of control of this predicate is established by recursion on X so our first step is to select an appropriate definition of this from the library of skeletons. We choose skeleton 3 from Appendix B.2, giving us the initial definition shown in clauses 5.1 and 5.2 of Figure 5.1. The display above these in the figure shows the editor at this stage in design. Its edit field shows the current predicate definition and above this are the edit controls which we shall use later to extend the definition.

The lines in Figure 5.1 beginning with labels <1> and <2> correspond to the



$$factorial(X) \leftarrow \mathcal{T}(X) \quad (5.1)$$
$$factorial(X) \leftarrow \mathcal{U}(X,Y) \wedge factorial(Y) \quad (5.2)$$

Figure 5.1: Techniques editor tool example - first step

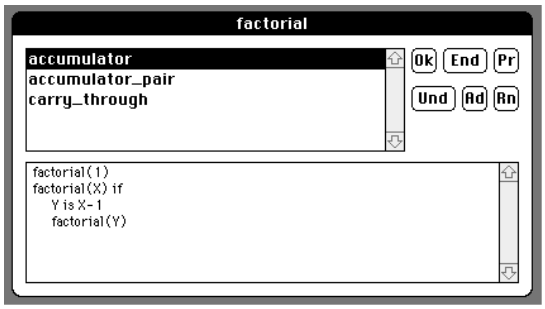
subgoals $\mathcal{T}(X)$ and $\mathcal{U}(X,Y)$ respectively. The menu shown opposite the `Id` label above the edit field allows these subgoals to be instantiated to the appropriate test and update: in the first case $\mathcal{T}(X)$ becomes $X = 1$ and in the second case $\mathcal{U}(X,Y)$ becomes Y *is* $X-1$. This takes us to the definition shown in Figure 5.2.

The menu at the top left of Figure 5.2 gives a list of the extensions which can be performed on this definition. We want the value of the factorial to be accumulated as we recurse down the sequence of integers established by the skeleton. Therefore we choose the `accumulator` extension. This operates in a similar way to the example of Section 4.1, adding an additional argument which is tested in the base case and carried up through the recursion via an update subgoal. The resulting definition is shown in Figure 5.3.

Finally, we instantiate the test and update subgoals in the definition: $\mathcal{T}(F)$ becoming $F = 1$ and $\mathcal{U}(P,F)$ becoming F *is* $P*X$. This completes our definition of *factorial*.

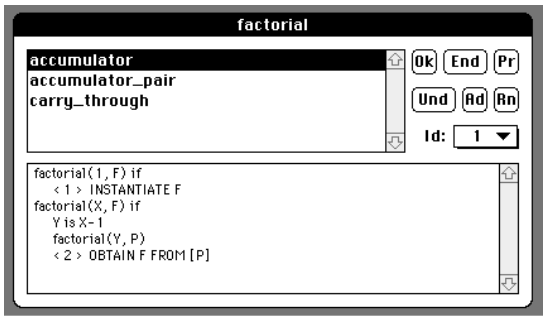
5.2 Process Editor

Section 4.2 described the use of Definite Clause Grammar (DCG) notation for describing a form of non-deterministic sequential process. We now explain how a diagrammatic notation is used to describe these DCGs. An artificial example is used in this section for simplicity but Chapter 6 contains a more realistic example. One of the principles underlying this editor is that every graphical description produced in the editor (even the earliest ones) have a translation into the underlying DCG notation, so we can view the developing logical description as we change the graphical one.



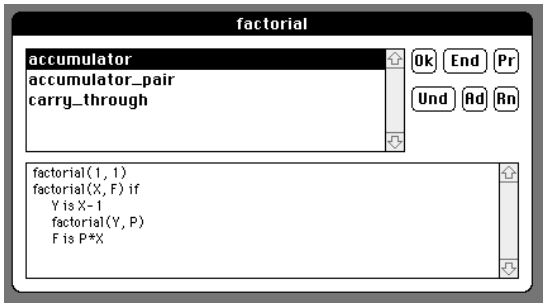
$$factorial(1) \tag{5.3}$$
$$factorial(X) \leftarrow Y \; is \; X - 1 \wedge factorial(Y) \tag{5.4}$$

Figure 5.2: Techniques editor tool example - second step



$$factorial(1, F) \leftarrow \mathcal{T}(F) \tag{5.5}$$
$$factorial(X, F) \leftarrow Y \; is \; X - 1 \wedge factorial(Y, P) \wedge \mathcal{U}(P, F) \tag{5.6}$$

Figure 5.3: Techniques editor tool example - third step

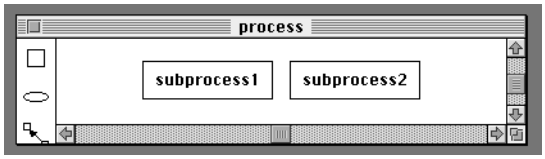


$$factorial(1,1) \tag{5.7}$$

$$factorial(X,F) \leftarrow Y\ is\ X-1 \wedge factorial(Y,P) \wedge F\ is\ P*X \tag{5.8}$$

Figure 5.4: Techniques editor tool example - final step

The editor uses a separate window for each DCG predicate. Figure 5.5 shows the window for a DCG called *process*. It contains two subsequences: *subprocess*1 and *subprocess*2 which we introduced by adding the two boxes shown in the diagram.



$$process \Rightarrow subprocess1 \tag{5.9}$$

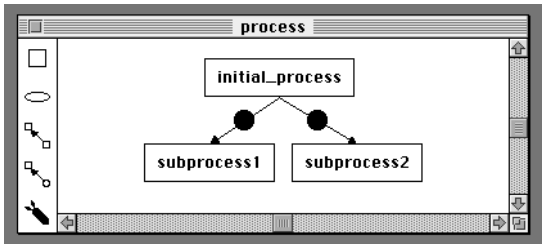
$$process \Rightarrow subprocess2 \tag{5.10}$$

Figure 5.5: Process tool example - first step

Subsequences are connected by links. Figure 5.6 shows the initial two subsequences connected to a common preceding subsequence, *initial_process*.

Similarly, we can describe convergences between processes by linking to a common successor sequence. Figure 5.7 converges on a subsequence called *process*, which is the same as the the DCG window name so we now have a recursive DCG which requires *process* to contain an *initial_process*, followed by either *subprocess*1 or *subprocess*2, followed by a further *process*.

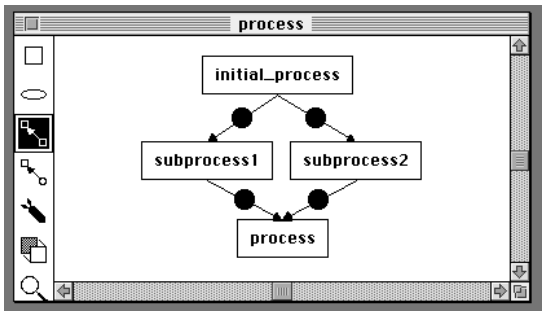
We may want to define arguments to the DCG which are used to pass infor-



$$process \Rightarrow initial\_process, subprocess1 \quad (5.11)$$

$$process \Rightarrow initial\_process, subprocess2 \quad (5.12)$$

Figure 5.6: Process tool example - second step

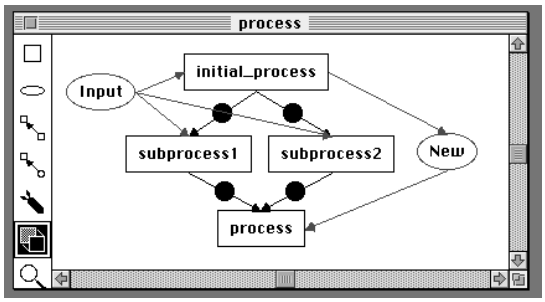


$$process \Rightarrow initial\_process, subprocess1, process \quad (5.13)$$

$$process \Rightarrow initial\_process, subprocess2, process \quad (5.14)$$

Figure 5.7: Process tool example - third step

mation around during processing. This is done graphically using an ellipse to represent each argument name and links connecting it to the appropriate predicates. In Figure 5.8 we have added the arguments *Input* (which is intended as an input argument to the head of the *process* DCG) and *New* (derived from *initial_process* and an input to the recursive call of *process*). Notice that in the translation we assume that *Input* is an argument to the head of the DCG because no link goes from a box to it (instead it is linked to three boxes).



$$process(Input) \Rightarrow initial\_process(Input, New), \; subprocess1(Input), \; process(New) \quad (5.15)$$

$$process(Input) \Rightarrow initial\_process(Input, New), \; subprocess2(Input), \; process(New) \quad (5.16)$$

Figure 5.8: Process tool example - final step

Other components of the DCG are the observed events in the process (corresponding to the items within square brackets in the DCG, as explained in Section 4.2) and conditions on clauses expressed as normal Prolog subgoals (corresponding to the subgoals within curly brackets in a DCG). These are added textually via windows which pop up when either a link or a box are clicked, allowing the additions to be placed at appropriate points in the sequence. This completes our introduction to the process editor.

5.3 Diagrammatic Recursion Editor

Section 5.1 described the techniques editor, which requires skeletal definitions to provide the basis for specification refinement. One way of viewing the diagrammatic recursion editor is as a way of defining a class of skeletons.

The form of editing done in this tool is, in one sense, more primitive than comparable diagrammatic editors. In particular, it constructs only a partial specification of a predicate, whereas the GraSP system [Agusti et al., 1998] can

represent complete Horn clause specification via more sophisticated use of similar graphical notation. In another sense the diagrammatic recursion editor is more sophisticated than GraSP because the partial specifications it produces can be loaded into the techniques editor of Section 5.1 and completed there. This allows a diagrammatic description of the overall structure of the recursive predicate, switching to textual description for the fine detail.

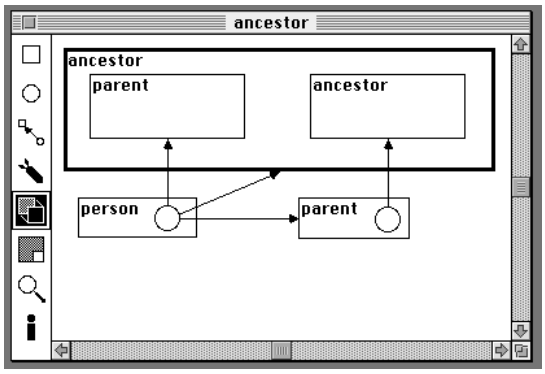
The mechanism of diagram construction in this editor is similar to that of the process editor (see Section 5.2) but the translation between diagrammatic symbols and formal notation is different. As before, each window defines a predicate. Within the window there must be a box drawn with a thick line which represents the head of the predicate. We shall call this the head box. The predicate is assumed to have an output argument and each box drawn within the head box allows the derivation of that output argument from its corresponding argument (via an update in the notation used in Section 5.1). Additional boxes drawn outside the head box describe predicates which are part of the conditions needed to derive the output of the head box but which do not directly derive the output. Instead, variables in these boxes (represented as circles) are linked to other boxes, denoting that the variables are arguments to the corresponding predicates. Figure 5.9 shows a diagram describing an *ancestor* example. The basic recursive structure of the definition is emphasised by focusing on the thick lined box which shows that the output of *ancestor* can either be from *parent* (the base case) or from *ancestor* itself (the recursive case). The other boxes give the additional conditions: that the ancestor may be the parent of the person appearing as the argument to *ancestor* or the ancestor of the parent of the person appearing as the argument to *ancestor*.

If we now use the techniques editor of Section 5.1 to instantiate all the tests ($\mathcal{T}_1 \ldots \mathcal{T}_5$) to be unifications between the pairs of variables (*e.g.* $\mathcal{T}_1(\{X_2, X_1\})$ becomes $X_2 = X_1$) then our logic program is:

$$\begin{aligned} ancestor(X_1, X_6) \quad &\leftarrow \quad person(X_1) \wedge \\ &\qquad\quad parent(X_1, X_4) \wedge \\ &\qquad\quad ancestor(X_4, X_6) \end{aligned} \tag{5.19}$$

$$\begin{aligned} ancestor(X_1, X_4) \quad &\leftarrow \quad person(X_1) \wedge \\ &\qquad\quad parent(X_1, X_4) \end{aligned} \tag{5.20}$$

This completes our introduction to the diagrammatic recursion editor, and to all the LSS tools of relevance to this thesis. The next chapter demonstrates how they can be used together.



$$ancestor(X_1, X_6) \leftarrow person(X_2) \wedge \mathcal{T}_1(\{X_2, X_1\}) \wedge parent(X_1, X_3) \wedge \mathcal{T}_2(\{X_3, X_4\}) \wedge ancestor(X_4, X_5) \wedge \mathcal{T}_3(\{X_5, X_6\})) \quad (5.17)$$

$$ancestor(X_1, X_4) \leftarrow person(X_2) \wedge \mathcal{T}_4(\{X_2, X_1\}) \wedge parent(X_1, X_3) \wedge \mathcal{T}_5(\{X_3, X_4\})) \quad (5.18)$$

Figure 5.9: Diagrammatic recursion tool example

Chapter 6

Worked Example

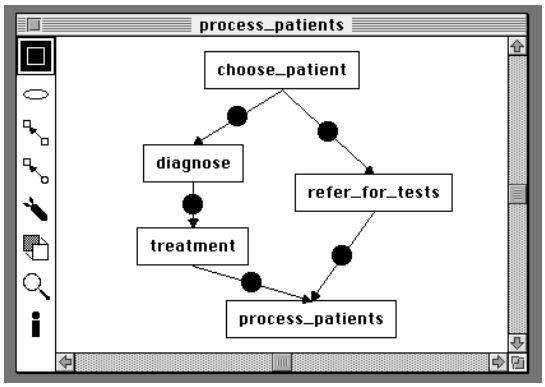
To illustrate how the LSS toolkit works we give a scenario in which a simple diagnostic system is built. The example is similar to the one used in [Robertson, 1996a]. As a starting point for the scenario we give the following informal problem description:

> A basic model of patient processing is when we have a group of patients, each of whom is either diagnosed and given an appropriate prescription or is referred for tests. Diagnosis is performed on each patient, based on his or her symptoms, by revising a set of hypotheses until these cover a sufficient subset of symptoms observed in the patient. Those patients which cannot be diagnosed are referred for tests. The interface to our basic processing system allows selections to be made from a menu of patient names and lists in a window the prescriptions and referrals.

We start by describing the overall processing using the process editor of Section 5.2. The screen snapshot of Figure 6.1 shows the stage where the basic process stages have been charted. The process begins by choosing a patient and ends by recursing. In between, we either go through diagnosis followed by treatment or referring the patient for tests. This translates to DCG expressions 6.1 and 6.2. Next we must add to the process definition the arguments needed to control it. To choose a patient we must know the set of patients being processed and we will identify the patient we wish to treat next, plus the remainder of patients left to be processed. Diagnosis for the selected patient will produce some diagnostic result, from which the subprocess of treatment produces a prescription.

By adding these arguments we construct the diagram shown in Figure 6.2, which translates to DCG expressions 6.3, 6.4 and 6.5.

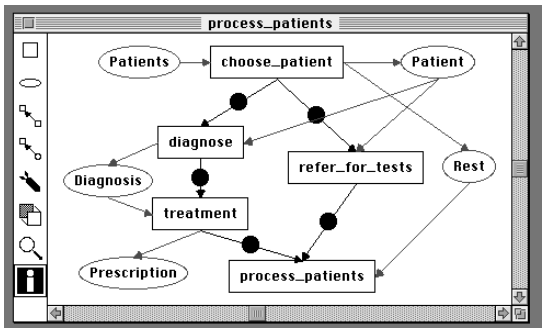
Next, we consider how to describe the diagnostic procedure. We build this using the diagrammatic recursion tool introduced in Section 5.3. This tool produces normal Prolog clauses, not DCG clauses, so we must construct (by hand)



$$process\_patients \Rightarrow choose\_patient, diagnose, treatment, process\_patients \quad (6.1)$$

$$process\_patients \Rightarrow choose\_patient, refer\_for\_tests, process\_patients \quad (6.2)$$

Figure 6.1: Early process diagram for our scenario



$$process\_patients(Patients) \Rightarrow choose\_patient(Patients, Patient, Rest) \wedge diagnose(Patient, Diagnosis) \wedge treatment(Diagnosis, Prescription) \wedge [recommendation(Patient, Prescription)] \wedge process\_patients(Rest) \quad (6.3)$$

$$process\_patients(Patients) \Rightarrow choose\_patient(Patients, Patient, Rest) \wedge refer\_for\_tests(Patient) \wedge [referred(Patient)] \wedge process\_patients(Rest) \quad (6.4)$$

$$process\_patients([]) \Rightarrow [] \quad (6.5)$$

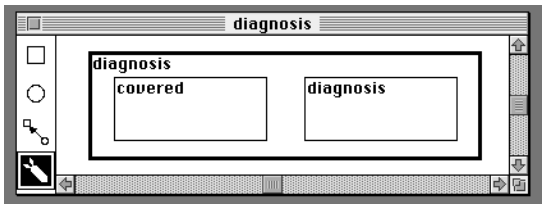
Figure 6.2: Final process diagram for our scenario

a clause which links the DCG for *diagnose* to the Prolog goal for *diagnosis*. This is:

$$diagnose(Patient, Diagnosis) \Rightarrow \{diagnosis(Patient, Symptoms, Hypotheses, Diagnosis)\} \wedge [diagnosed(Patient, Diagnosis)] \quad (6.6)$$

where the additional two arguments in diagnosis are the hypothesis and symptoms sets which are adapted when performing the diagnosis.

We now describe *diagnosis* using the diagrammatic recursion tool. We begin with a basic recursive diagram with the head box containing one base case (*covered*) and one recursive case. This is shown in Figure 6.3.



$$diagnosis(X_1) \leftarrow covered(X_2) \wedge \mathcal{T}_1(\{X_2, X_1\})) \quad (6.7)$$

$$diagnosis(X_1) \leftarrow diagnosis(X_2) \wedge \mathcal{T}_2(\{X_2, X_1\})) \quad (6.8)$$

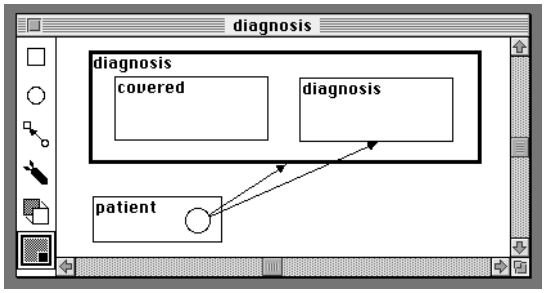
Figure 6.3: Diagrammatic recursion tool scenario - first step

We then introduce the first input argument to *diagnosis*, which is the patient to be diagnosed. This is an argument to both the head and recursive call of *diagnosis*, as shown in Figure 6.4.

Finally, we add the remaining arguments shown in Figure 6.5. The *symptoms* and *hypotheses* predicates generate sets of symptoms and hypotheses or tests these depending on whether the appropriate variables are instantiated, and the *revise* predicate revises the set of hypotheses for the given patient.

If we now use the techniques editor of Section 5.1 to instantiate all the tests except $\mathcal{T}_4$ to be unifications between the pairs of variables (*e.g.* $\mathcal{T}_1(\{X_2, X_1\})$ becomes $X_2 = X_1$). Test $\mathcal{T}_4$ needs a more complex test because it must protect against the set of diagnoses being empty. It therefore becomes $X_7 = X_8 \wedge not(X_8 = [])$ our logic program is:

$$diagnosis(X_1, X_6, X_3, X_8) \leftarrow hypotheses(X_3) \wedge patient(X_1) \wedge symptoms(X_6) \wedge covered(X_3, X_6, X_8) \wedge not(X_8 = []) \quad (6.13)$$



$$diagnosis(X_1, X_4) \leftarrow patient(X_2) \wedge \mathcal{T}_1(\{X_2, X_1\}) \wedge covered(X_3) \wedge \mathcal{T}_2(\{X_3, X_4\}) \quad (6.9)$$

$$diagnosis(X_1, X_4) \leftarrow patient(X_2) \wedge \mathcal{T}_3(\{X_2, X_1\}) \wedge diagnosis(X_1, X_3) \wedge \mathcal{T}_4(\{X_3, X_4\}) \quad (6.10)$$

Figure 6.4: Diagrammatic recursion tool scenario - second step

$$diagnosis(X_1, X_6, X_3, X_{10}) \leftarrow hypotheses(X_3) \wedge patient(X_1) \wedge symptoms(X_6) \wedge revise(X_1, X_6, X_3, X_8) \wedge diagnosis(X_1, X_6, X_8, X_{10}) \quad (6.14)$$

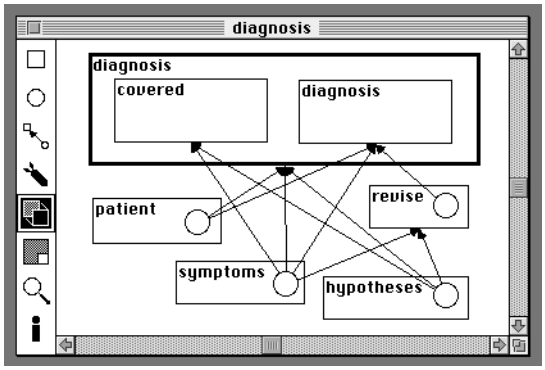
Renaming the variables and re-ordering the first three subgoals, purely for readability, gives us:

$$diagnosis(P, S, H, D) \leftarrow patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge covered(H, S, D) \wedge not(X_8 = []) \quad (6.15)$$

$$diagnosis(P, S, H, D) \leftarrow patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge revise(P, S, H, Hr) \wedge diagnosis(P, S, Hr, D) \quad (6.16)$$

Which can be transformed for efficiency by the transformation tool mentioned in Chapter 3 (but not described in detail in this thesis) into:

$$diagnosis(P, S, H, D) \leftarrow patient(P) \wedge symptoms(S) \wedge hypotheses(H) \wedge \left(\begin{array}{c} \left(\begin{array}{c} covered(H, S, D) \wedge \\ not(X_8 = []) \end{array} \right) \vee \\ \left(\begin{array}{c} revise(P, S, H, Hr) \wedge \\ diagnosis(P, S, Hr, D) \end{array} \right) \end{array} \right) \quad (6.17)$$

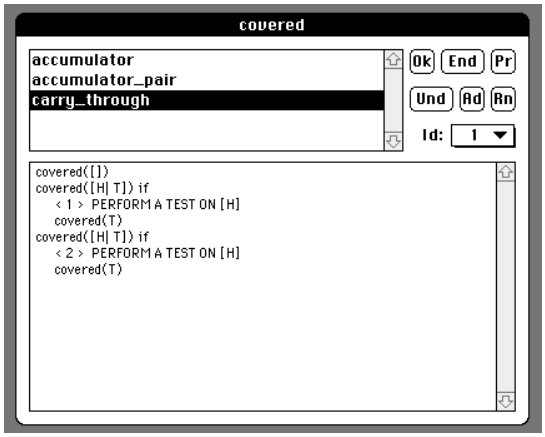


$$diagnosis(X_1, X_6, X_3, X_8) \leftarrow hypotheses(X_2) \wedge \mathcal{T}_1(\{X_2, X_3\}) \wedge patient(X_4) \wedge \mathcal{T}_2(\{X_4, X_1\}) \wedge symptoms(X_5) \wedge \mathcal{T}_3(\{X_5, X_6\}) \wedge covered(X_3, X_6, X_7) \wedge \mathcal{T}_4(\{X_7, X_8\}) \quad (6.11)$$

$$diagnosis(X_1, X_6, X_3, X_{10}) \leftarrow hypotheses(X_2) \wedge \mathcal{T}_5(\{X_2, X_3\}) \wedge patient(X_4) \wedge \mathcal{T}_6(\{X_4, X_1\}) \wedge symptoms(X_5) \wedge \mathcal{T}_7(\{X_5, X_6\}) \wedge revise(X_4, X_6, X_3, X_7) \wedge \mathcal{T}_8(\{X_7, X_8\}) \wedge diagnosis(X_1, X_6, X_8, X_9) \wedge \mathcal{T}_9(\{X_9, X_{10}\}) \quad (6.12)$$

Figure 6.5: Diagrammatic recursion tool scenario - final step

Finally, we use the techniques editor of Section 5.1 to define the way in which covering sets of hypotheses may be generated. We begin by choosing skeleton 18 from Appendix B.3, which traverses a list, choosing one or other of the two recursive clauses for each element depending on a test on that element in each clause. This is shown in Figure 6.6.



$$covered([]) \tag{6.18}$$

$$covered([H|T]) \leftarrow \mathcal{T}_1(H) \wedge covered(T) \tag{6.19}$$

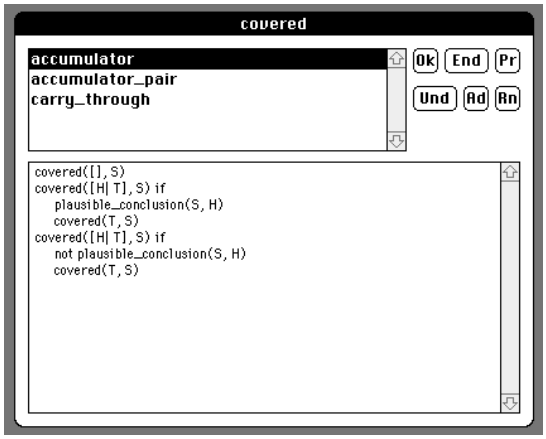
$$covered([H|T]) \leftarrow \mathcal{T}_2(H) \wedge covered(T) \tag{6.20}$$

Figure 6.6: Techniques editor tool scenario - first step

We then apply the *carry_through* extension from the menu at top left in Figure 6.6. This threads an argument through the predicate to carry the symptom information. We also instantiate $\mathcal{T}_1(H)$ to $plausible\_conclusion(S, H)$ and instantiate $\mathcal{T}_2(H)$ to $not(plausible\_conclusion(S, H))$. The resulting definition is shown in Figure 6.7.

Next we apply an *accumulator* extension from the menu top left in Figure 6.7. This adds an argument which passes the final diagnosis back up the recursion, having instantiated it in the base case.

Finally we instantiate the updates which we introduced with the accumulator extension. In the base case, where the list of hypotheses is empty, we want the set of diagnoses also to be empty so $\mathcal{T}(D)$ becomes $D = []$. In the recursive cases we want only hypotheses which are plausible conclusions to be added so $\mathcal{U}_2(P, D)$ becomes $D = [H|P]$ and $\mathcal{U}_3(P, D)$ becomes $D = P$. This gives us the final specification shown in Figure 6.9.

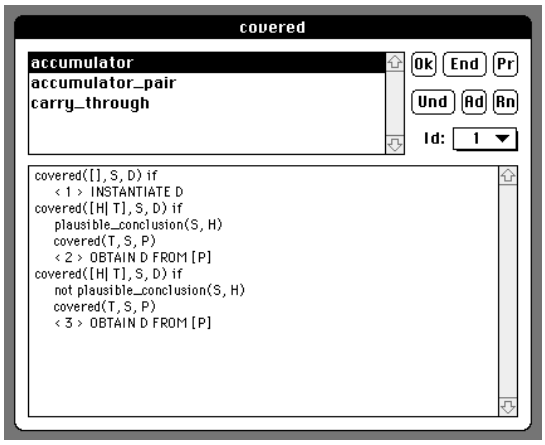


$$covered([], S) \tag{6.21}$$

$$covered([H|T], S) \leftarrow plausible\_conclusion(S, H) \wedge covered(T, S) \tag{6.22}$$

$$covered([H|T], S) \leftarrow not(plausible\_conclusion(S, H)) \wedge covered(T, S) \tag{6.23}$$

Figure 6.7: Techniques editor tool scenario - second step

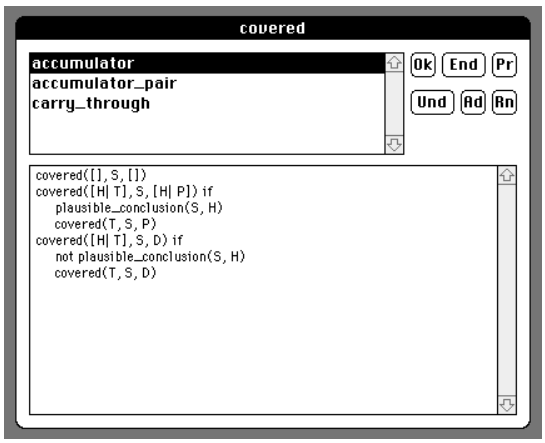


$$covered([], S, D) \leftarrow \mathcal{T}(D) \tag{6.24}$$

$$covered([H|T], S, D) \leftarrow plausible\_conclusion(S, H) \wedge covered(T, S, P) \wedge \mathcal{U}_2(P, D) \tag{6.25}$$

$$covered([H|T], S, D) \leftarrow not(plausible\_conclusion(S, H)) \wedge covered(T, S, D) \wedge \mathcal{U}_3(P, D) \tag{6.26}$$

Figure 6.8: Techniques editor tool scenario - third step



$$covered([], S, []) \tag{6.27}$$

$$covered([H|T], S, [H|P]) \leftarrow plausible\_conclusion(S, H) \wedge covered(T, S, P) \tag{6.28}$$

$$covered([H|T], S, D) \leftarrow not(plausible\_conclusion(S, H)) \wedge covered(T, S, D) \tag{6.29}$$

Figure 6.9: Techniques editor tool scenario - final step

Having come through this and the previous two chapters the reader should now understand the formal basis for the LSS system; have a feeling for the interface to each tool; and appreciate how the tools can work together on a design. In the next chapter we consider how effective this form of design can be.

Chapter 7

Evaluation

Effective use of heterogeneous toolsets like LSS requires that individual tools are accessible to target groups of designers and that large specifications can be assembled using combinations of tools. Section 7.1 explores the first requirement, using evidence from evaluations of techniques editors which were designed specifically for particular groups of designers. Section 7.2 addresses the second requirement through a case study in which LSS was used to re-implement one of its own tools - a moderately large task by logic programming standards. Finally, Section 7.3 highlights problems emerging from our use of LSS which motivated the construction of the HANSEL system.

7.1 Independent Evaluations of Techniques Editors

Techniques editing is an approach to design which can be described independently of the interface style used by designers when editing. Many different styles of interfaces are possible and these vary depending on the domain in which the editor is to be used. Usability tests depend crucially on getting this match right. This section describes two such tests, conducted on two different techniques editors which the author helped to construct. The first was used by novice programmers and the second was used by experts in ecological modelling.

7.1.1 Evaluation of a Techniques Editor for Novice Programmers

The most extensive evaluation of a techniques editor of which we have knowledge which was undertaken by psychologists at the University of Loughborough, using a techniques editor (named TEd) implemented by a group including the author[1].

[1]The principal engineer of this editor was A.Bowles and the rest of the team consisted of P.Brna and H.Pain.

The techniques used by this editor were simpler than the ones provided by the LSS editor because the target designers were novice programmers. The interface was also different, in general giving fewer options at each stage in editing and making smaller changes to the design at each step. Nevertheless, the basic approach to design is similar between TEd and the LSS techniques editor so the lessons learned from the Loughborough evaluation are instructive for LSS. A detailed report of methods and results appears in [Ormerod and Ball, 1996]. We summarise this below.

The method used in evaluation was to compare three groups of different students (each group containing ten or more). These students were all Prolog novices and were taking an introductory course in the design of basic recursive Prolog programs as part of their undergraduate degree. To protect against anomalies caused by differences in basic aptitude for Prolog all the students were pre-tested in designing elementary, non-recursive Prolog programs. The groups were then supported in their course in different ways. The first ("traditional") group were given a normal introductory course, using a standard text editor and a commercial Prolog environment. The second ("manual techniques") group were taught about techniques explicitly, in a style which encouraged them to approach the problem as if they were to use a techniques tool, but they used the standard editor and commercial environment. The third ("automated techniques") group were taught techniques explicitly and used TEd exclusively to design their programs. Each of the three groups was tested on six problems and three forms of quantitative results were obtained for each student on each problem: the number of technique selection errors for each group; the frequency of errors of different types; and time taken to find a solution.

In general, the "manual techniques" group performed little better than the "traditional" group on any of the measures, and tended to make slightly more errors. The "automated techniques" group, however, made approximately half as many errors as the other groups and produced solutions more quickly for most of the problems (although this difference is statistically significant at $P < 0.05$ for only two of the problems). Although this at first sight appears encouraging for techniques editing there is a caveat with important consequences for LSS. The advantages of tool support were greatest for "routine" problems where careful thought was not needed to devise the design of the solution to the problem. For more complex tasks it appeared that the benefit from tool support became a smaller proportion of the total time taken to solve the problem, so the tool became less valuable. There was also anecdotal evidence that students using TEd became dependent on the tool, making it difficult for them to shift later to a traditional Prolog design environment. In short, the evidence of this study is that a techniques editor can improve novice programming performance but it is unclear whether these benefits remain significant as novices become more experienced and tackle more challenging problems. This raises the issue of what it is like to use this sort of technology to tackle larger problems. We turn to this issue in Section 7.2.

7.1.2 Evaluation of a Techniques Editor for Expert Ecological Modellers

The TeMS system described in [Castro, 1999] applies techniques editing methods to a highly domain-specific problem: assisting ecological experts to construct animal population dynamics models. This sort of task is different from generic techniques editors like those used in LSS and in the evaluation of Section 7.1.1 because the skeletons and extensions used as building blocks for the specification are larger and are domain specific (for example, logistic growth of a population is a basic component in TeMS). The reason for this is to bring the editing operations of the editor closer to concepts which ecological experts (with no training in logic) are likely to understand. The evaluation of TeMS is relevant to LSS because it gives an indication of the likely benefits which might accrue from including more specific varieties of tools such as the techniques in the toolset. TeMS was evaluated primarily by presenting fifteen experimental participants with a standard modelling problem; observing them as they used TeMS to try to solve it; and obtaining feedback through interviewing and a questionnaire. All the participants had substantial ecological modelling experience and most were trained in ecology to postgraduate level. Only three had experience of Prolog and none had experience of techniques editing or TeMS.

The modelling problem given to the experimental participants was a textual description of a scenario which, essentially, involved an animal population divided into age classes which is invaded by a predator population - the task being to predict the change in the age structure of the prey over a given time span. This would have been beyond the ability of the participants to program quickly by conventional means but they all produced models to their satisfaction within the time allotted by the experiment (20 minutes) so TeMS certainly made it faster for them to produce appropriate code. Furthermore, the subjects claimed that the modelling approach was clear and that the models produced were, on the whole, well structured. This enthusiasm seems impressive but it must be remembered that TeMS builds a narrow class of programs and therefore it can engage in a highly domain-specific dialogue with designers. Thus, it gives a useful marker for the degree of acceptability which we could hope to achieve with a highly targeted system built on top of this generic design method. There were also points on which TeMS showed signs of weakness, although none of these appeared to be insurmountable. In particular, there was felt to be some loss of the broader picture of the whole of the developing model when working on a part of it under TeMS control. This resonates with problems experienced in LSS (see Section 7.3.2).

7.2 Tackling a Large Example using LSS

The evaluations of the previous section raise confidence that tools of the sort used in LSS can produce modest but measurable increases in the quality of specifications for target groups of designers, and that highly specialised tools can

make this sort of technology accessible to experts in domains far removed from logic programming. There remains, however, the issue of whether a collection of these tools (each of which normally builds a limited part of a specification) can work together in an effective way to build large programs. This issue was explored by attempting to use LSS to build one of its own tools: the techniques editor. Details of this evaluation appear in [Robertson, 1998a] and we summarise the main points in this section.

The aim of the evaluation was to take a substantial piece of existing Prolog code, which had been written using a normal text-based editor, and reconstruct it using LSS. This does not allow any conclusions about how well LSS assists in problem formulation or in its appropriateness for target groups of designers. It does help to identify problems in managing the interaction between LSS tools, contributing to the critical analysis of Section 7.3.

The LSS techniques editor code was chosen for this case study because it is moderately large and complex by logic programming standards. It consists of 85 predicate definitions, each of which varies in length from 2 to 68 lines of code (a total of approximately 850 lines). A wide variety of definitional styles appear (including "impure" styles such as failure-driven iteration) and the number of arguments for predicates varies from 0 to 8, so each predicate differs significantly from others in its structure. Roughly a quarter of the lines of code control the techniques editor interface. Although the LSS interface specification tool is not described at length in this document it was used in the reconstruction and all the interface code was included in the case study. Neither the techniques editor code nor the LSS tools were adapted during the evaluation and the temptation to force LSS tools into tasks to which they are unsuited was avoided.

Of the 85 predicates used in the techniques editor, 49 could be reconstructed using the LSS tools. This is an encouraging result because the code for the techniques editor was written without any thought of reconstructing it in this way. Of the 36 predicates which were not reconstructed by LSS, 10 of these could have been reconstructed but were ignored because they were too simple to warrant the use of a special-purpose tool - they were predicates of two or three lines which are easier to define by hand. The remaining 26 predicates resisted re-construction because they did not fit the paradigm of design of any of the LSS tools - either by not having an obvious skeleton for use in the techniques editor; or by not having the DCG-based structure required by the process tool; or by not having the recursive structure demanded by the diagrammatic recursion editor.

The general results described above appear encouraging but the case study revealed a number of problems with LSS. These are:

- There are recurring patterns of design which could have been represented within the LSS toolset but weren't. An example is in defining the interface code, which was done by using the interface tool to produce the code for generating the appropriate display objects and the techniques editor for filling the "gaps" left by the interface tool. These gaps are primarily the callbacks which occur upon selecting an active window component (such as a button or a menu item) and in many cases they have a standard

structure. These standard structures, however, are not available in any of the current LSS tools.

- A tool is present in the LSS system for performing correctness-preserving transformations to predicates, but it is merely an early prototype and provides only a small set of transformations. The current transformation tool was useful in some cases (particularly in packing 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 goals as disjuncts) but could have done more had it been better equipped.

- The libraries of skeletons and extensions in the techniques editor are small, simple and generic. A surprising number of the predicates in the case study could be reconstructed from these libraries but it was more time consuming to build them from small, generic components than it would have been from larger, more specialised ones. Adding more varieties of skeleton and technique is straightforward in theory but has practical limitations because the more we add the larger becomes the library and the longer it takes to browse through it. The current library contains less than 30 skeletons and, at a rough guess, the upper limit for browsing might be 100 so some extension is possible but not a great deal, unless a more sophisticated browsing system were invented.

- The process tool and the diagrammatic recursion editor were not used at all in the recursion. The main reason for this is that they are intended to assist in early conceptualisation of specifications and are not the most direct way to reconstruct a specification which is already known precisely. Although these tools could have been used for some of the predicates it was necessary in this experiment to adhere to the requirement that the most appropriate tool be used for the task in hand.

In addition to the problems above, there are a number of general problems with the distributed style of specification used in LSS which emerged from this and other smaller experiments with the toolset. These are the subject of the next section.

7.3 Problems with LSS

7.3.1 Choice of Style

One of the advantages of a diverse set of design tools is that it allows a choice of styles in which to describe a problem. This is normally achieved at the cost of restricting the range of specifications which each tool can describe. The most extreme example of this is the Process tool (Section 5.2) which synthesises only Definite Clause Grammars, so must always generate predicates with the additional pair of arguments obtained by translation from DCG clauses to standard

Prolog. For example, the translation of DCG clauses 5.9 and 5.10 from Figure 5.5 is:

$$process(S_i, S_f) \leftarrow subprocess1(S_i, S_f) \quad (7.1)$$
$$process(S_i, S_f) \leftarrow subprocess2(S_1, S_2) \quad (7.2)$$

A problem with tools which are restrictive in this way is that designers may not always be able to predict whether they are too restrictive for the problem in hand. Suppose that we want to make the choice of whether to use clause 7.1 or clause 7.2 be determined by some test on the input sequence (S_i). The extended predicate might look like this:

$$process(S_i, S_f) \leftarrow test(S_i) \wedge subprocess1(S_i, S_f) \quad (7.3)$$
$$process(S_i, S_f) \leftarrow not(test(S_i)) \wedge subprocess2(S_1, S_2) \quad (7.4)$$

The problem is that clauses 7.3 and 7.4 cannot be defined by the Process tool because it gives no access to the variables in the last two arguments of the predicate. These are generated automatically via the standard DCG to Prolog translation mechanism. The only way in the Process tool to obtain the control added in clauses 7.3 and 7.4 is to duplicate the input sequence as an additional argument to the DCG, which is then:

$$process(S'_i) \Rightarrow \{test(S'_i)\}, subprocess1 \quad (7.5)$$
$$process(S'_i, S_f) \Rightarrow \{not(test(S'_i))\}, subprocess2 \quad (7.6)$$

This translates to the Prolog clauses:

$$process(S'_i, S_i, S_f) \leftarrow test(S'_i) \wedge subprocess1(S_i, S_f) \quad (7.7)$$
$$process(S'_i, S_i, S_f) \leftarrow not(test(S'_i)) \wedge subprocess2(S_1, S_2) \quad (7.8)$$

Although this is a solution it is an inelegant one, since it requires us to maintain two versions of the input sequence and keep these in synchrony during the execution of the program. The Process tool really should not be used for this sort of specification but the representational requirement which made it inappropriate might not have been identified early in the design.

7.3.2 Maintaining an Overview

The use of Prolog as a *lingua franca* between tools means that there are no syntactic translation problems between parts of the specification which have been designed in different styles. Unfortunately, this does not absolve us from the need to understand the semantics of each predicate which must interact with the one we are defining. The distributed tools give no assistance with this. For

example, there is nothing to prevent a designer from implementing *subprocess*1, called as a goal from clause 7.1 in either of the following ways:

$$subprocess1(S_i, S_f) \leftarrow S_f \; is \; S_i + 1 \tag{7.9}$$
$$subprocess1(S_i, S_f) \leftarrow S_f = [a|S_i] \tag{7.10}$$
$$subprocess1(S_i, S_f) \leftarrow process(S_i, S_f) \tag{7.11}$$

Definition 7.9 is wrong because it introduces a type error, so this might be prevented by using a typed language instead of Prolog. Definition 7.10 is likely to be wrong because it makes S_f larger, rather than smaller as we would expect from the way it is called as a subgoal of clause 7.1. Definition 7.11 may not be wrong but it raises potential problems of non-termination because it introduces a mutual recursion with clause 7.1. The reason we can see these difficulties is that we know pertinent information about the design of all of the predicates. The problem is that, in the worst case, we may have to study all of the predicates in order to be sure that we have the relevant information. If so, then we lose the modularity of design which is one of the advantages of the distributed design method.

7.3.3 Saying Less

Communication between LSS tools is through Prolog clauses which may contain update and test goals used in techniques editing. This is the language of output for all the tools and, as such, it is close to conventional Prolog. The problem with this language of communication is that it allows only crude descriptions of partially developed designs. For example, if a designer wishes to transport a definition of a simple list traversal program between tools then this can be done either using the full specification, which might be:

$$p([]) \tag{7.12}$$
$$p([H|T]) \leftarrow test(H) \wedge p(T) \tag{7.13}$$

or it can be done using a specification which does not commit to the choice of test on H:

$$p([]) \tag{7.14}$$
$$p([H|T]) \leftarrow \mathcal{T}(H) \wedge p(T) \tag{7.15}$$

or it can be expressed more generally by leaving open the relationship between the arguments in the head and recursive subgoal.

$$p(S) \leftarrow \mathcal{T}(S) \tag{7.16}$$
$$p(S_1) \leftarrow \mathcal{U}(S_1, S_2) \wedge p(S_2) \tag{7.17}$$

We know there are other ways of describing this specification which cannot be expressed in the LSS communication language. For example, we could say that the recursive subgoal always takes a set[2] which is no larger than the set in the head of the clause. Formally, this might be:

$$p(S) \leftarrow \mathcal{T}(S) \tag{7.18}$$
$$p(S_1) \leftarrow S_1 \supseteq S_2 \wedge p(S_2) \tag{7.19}$$

This is less restrictive than the partial program defined by clauses 7.14 and 7.15 because the inequality between sets need not be achieved only by removing the first element from the set. It is more restrictive than the partial program defined by clauses 7.16 and 7.17 because it tells us more about the sort of relation we must have during the recursion. These sorts of shades of meaning in communication are not achievable in the LSS toolset, regardless of the sophistication of individual tools.

7.3.4 Transformations Over Many Predicates

The LSS tools focus on individual predicates as the units of design. This is a common assumption in logic programming - the idea being that we make sure we define each predicate correctly and, since there are no global variables to create hidden interactions between predicates, the correct definitions can simply be combined. Module systems make this sort of combination even safer by clarifying the points of interaction between groups of predicates and preventing unanticipated use of predicates internal to a module. The self-contained nature of each predicate is one of the strengths of logic programming but there is a problem in assuming that each LSS tool can manipulate individual predicates independently from those in other tools, as the following example demonstrates.

Suppose that LSS is being used to build a simple program which currently consists of the definition of predicate p which was given in clauses 7.12 and 7.13 and the predicate q below, which calls p.

$$q(X) \leftarrow p(X) \tag{7.20}$$

The definition of p is not completed and, using a techniques editor, we add a second argument to it, changing it to:

$$p([], F) \leftarrow \mathcal{T}(F) \tag{7.21}$$
$$p([H|T], F_1) \leftarrow \mathit{test}(H) \wedge p(T, F_2) \wedge \mathcal{U}(F_2, F_1) \tag{7.22}$$

The problem is that clause 7.20 is now out of step with the extended version of p. Ideally, we would like to revise this clause automatically by adding the extra argument to its subgoal and perhaps even adding a test to remind the designer that attention should be paid to it. The resulting clause might be:

[2]The argument in this example is a list, not a set, but the distinction isn't important here.

$$q(X) \leftarrow p(X,Y) \ \wedge \ \mathcal{T}(Y) \tag{7.23}$$

This sort of change is difficult to perform automatically in a distributed toolset like LSS because the choice of revision is not unique. In our example there is a second possible revision to q which is:

$$q(X) \leftarrow p(Y,X) \ \wedge \ \mathcal{T}(Y) \tag{7.24}$$

This alternative is possible because there is no way of knowing whether the $p(X)$ in clause 7.20 was intended to refer to the first argument of p (which appeared originally in clauses 7.12 and 7.13) or to the second argument of p (which was added in clauses 7.21 and 7.22).

7.3.5 Maintaining Properties During Use and Revision

Interaction between the design of predicates is not only through changes in arity but also through expectations of the properties of those predicates. For example, the following is a predicate, $u(S)$, which recurses over set S via the relation, r, and terminates when S is empty:

$$u([]) \tag{7.25}$$

$$u(S_1) \quad \leftarrow \quad r(S_1,S_2) \ \wedge \ u(S_2) \tag{7.26}$$

Termination of this predicate depends on r, with the most obvious termination property being that $r(S_1,S_2)$ should ensure that S_2 is no larger than S_1. The problem is that LSS does not communicate information about desirable properties between tools, so there is nothing to stop r being defined as (for example):

$$r(S_1,[a|S_1]) \tag{7.27}$$

We can spot this as a bad design decision only because we have been told a property which r must possess in order to satisfy the needs of u. The LSS tools provide no mechanism for describing or deriving such properties. Nor does LSS have the ability to communicate property information between tools.

This completes our evaluation of LSS and, with it, the part of this thesis which concerns distributed forms of design. We have shown that, in the right hands, this form of design can be used for significantly sized tasks. There are, however, problems associated with the architecture which leave it weak in some important areas. In the next part of this thesis we concentrate on these problems, which are related primarily to the need for a stronger notion of lifecycle uniting the design tools.

Part III

A Solution Imposing Structured Design

Chapter 8

Overview of the HANSEL Lifecycle Model

HANSEL is intended to be applied to problems which can be idealised as operations over sets of axioms, some of these sets being inputs to the specification and the others being outputs. Early refinement in HANSEL is purely in terms of these sets, abstracting away from their contents. Later refinement puts in place the mechanisms needed to access and change elements of these sets. This, and its link to subsequent testing, produces a form of lifecycle which is unlike those of conventional software engineering but which is analogous to some aspects of those conventional lifecycles.

Figure 2.1 of Section 2.2 shows a diagram of the "V" lifecycle model from software engineering. The key idea in this form of lifecycle is that commitments made in the design stages (on the descending arm of the "V") are carried across to be used in validation and verification studies (on the ascending arm of the "V"). The aim of HANSEL is to demonstrate parallels to this sort of design style using logic for design stages and automating the transfer of a limited range of commitments, which in the case of HANSEL are properties of the specifications at each stage. The objective is *not* to describe an entire lifecycle model to compete with those in industrial software process control. This would be a huge and (arguably) fruitless undertaking because it would require formal representation of parts of the process, such as requirements analysis, which are highly resistant to formal treatment. Our objective is much more modest: to show that while staying close to comparatively simple notions of formal specification and refinement we can develop routes of transfer of some commitments from design to testing, and that use of formality allows the transfer to be automated.

Having made clear that the analogy must be drawn with caution, Figure 8.1 shows the diagram of Figure 2.1 re-interpreted for HANSEL. The descending arm of the "V" shows the stages of refinement of a HANSEL specification: an initial template is chosen from a library; then it is refined using a set based notation; this is used to cue the introduction of skeletal definitions for the predicates in

the specification; and once these have been fully fleshed put we have an executable[1] specification in Prolog. Refinement through these stages is done using an interactive design tool which (automatically and behind the scenes) records properties of the specification which are implied by the chosen refinement steps and which can be used to test the executable specification (the ascending arm of the "V"). The dotted arrows on the diagram represent this transfer of testing information. Notice that there is no dotted line from the very highest level of design to testing the validity of a template. This is because the properties carried across in the current HANSEL inplementation are related closely to the structure of the executable specification and there is a big gap between the information obtained from testing these properties and higher level considerations of validity of the overall design.

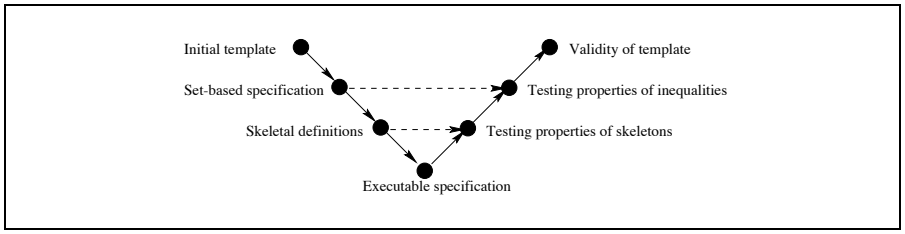


Figure 8.1: The HANSEL lifecycle model

On descending through the levels of specification in HANSEL a designer moves from generic to domain-specific forms of description. The highest levels of initial templates and set-based specification are task-independent and domain-independent. The skeletal definitions of the next level are task-specific but domain-independent. The final details supplied to form an executable specification are domain-specific. This means that although the highest levels of HANSEL and its framework for design are general the lower levels are not. Having said this, the sorts of tasks described in the current library of skeletal definitions are flexible enough to cover a wide variety of problems. The strategy in HANSEL has been to make refinement steps as general as possible without making them so general that they offer little pragmatic support for designers. This is a question of balance – we could have more support by reducing the generality of the refinements. If HANSEL were to be made into a practical tool we believe this sacrifice of generality would be necessary but the aim of this thesis is to provide an adaptable tool for experiment, not to build a development system.

HANSEL is a comparatively small system but it possesses a number of novel features which we shall describe in detail in later chapters. These are:

Representation of specifications using constraints on theories. The specifications constructed by HANSEL are unusual because they consist of predicates which relate sets of axioms. Each axiom set is itself a restricted set

[1]HANSEL does not guarantee that the specification is executable. In particular, the ordering of goals in the body of a clause may not be the right one for a standard Prolog interpreter.

of Horn clauses (see Appendix A for the restrictions). This means that we have to think of specifications as manipulations on sets of clauses rather than taking the more conventional view of logic programs as relations between objects in some universe of discourse. In return for this shift in view we are able to treat the axioms of theories themselves as objects which may be manipulated. This is important when, within the deductive framework of logic programming, we need to specify problem solving methods which are not purely deductive. Such situations are frequent - for example when we hypothesise causes for events or when we partition an inconsistent set of axioms into separate consistent subsets.

Integration of techniques-based design with set-based refinement. When automating the design of specifications there is always a tension between the desire to provide a general, abstract system capable of dealing with a broad range of problems and the need to give guidance in appropriate forms of design for specific tasks. The set-based refinements used in HANSEL early design are an example of the former, while the skeletons and extensions used in HANSEL later design are an example of the latter. We have found a direct and simple way of integrating these, by having refinements to set inequalities introduce "hooks" for the introduction of binary relations, then using skeletons to define these relations and then extending from binary to n-ary relations through the addition of argument slices.

Association of properties with refinements in a lightweight style. The dominant methods of formal design within the logic programming community are based on constructive and deductive synthesis (as described in Section 2.1). The most attractive way of performing this sort of design is to begin by specifying the properties which a logic program should possess and then use these to construct a program which satisfies exactly these properties. The fundamental problem with this approach is that describing appropriate properties for non-trivial problems, and knowing that we have described all properties which are relevant to our problem, is difficult, time-consuming and boring (see quote in Section 2.1). A more lightweight[2] approach, with greater similarity to conventional software engineering, is to associate properties with structural features of the design (so we engage from the beginning in structural synthesis as described in Section 2.1) and use these either to guide structural choices (not covered in this thesis but see Section 14.2) or to assist in retrospective testing (addressed in this thesis in a limited way). In doing this we sacrifice the guarantee that the logic programs we produce are correct with respect to those properties. The gain is that we can be selective in the properties we associate with structural changes, making it possible to confine our attention to those which we think are useful rather than those which we were obliged to include because we were locked into a method which required guarantees.

[2]A discussion of the benefits of lightweight uses of formality in design appears in [Robertson, 1995]

A direct route from design to a limited form of automated testing. The HANSEL system allows properties to be associated with structural features of a logic program in a way which need not intrude on structural synthesis. The design tool itself does not make use of this information (although it could, as outlined in Section 14.2). We do, however, demonstrate a direct means of making use of property information to generate test conditions for the logic programs produced by HANSEL. This goes nowhere near far enough to provide a convincing testing regime for these sorts of programs but it is an additional weapon in the armoury of designers and the method for generating test conditions is automatic, via a simple meta-interpreter, thus placing little overhead on designers.

In Chapter 9 we explain the formal concepts upon which HANSEL is based. We then summarise, in Chapter 10, how these formal methods are combined in the HANSEL tool. This paves the way for Chapter 11 which gives a short worked example of HANSEL being applied to part of the problem on which we demonstrated LSSin Chapter 6. Finally, we show some examples in Chapter 12 of HANSEL at work on more realistic problems.

Chapter 9

Formal Concepts

This chapter describes the formal system on which HANSEL is based. Section 9.1 defines what is meant by "axiom" set and "theory", which are the building blocks of early specifications. Section 9.2 introduces the use of Horn clauses containing set expressions as an early specification language. Section 9.4 describes the system of rewrite rules which is used in early refinement of these specifications. Section 9.5 explains how a form of "skeleton" notation (related to that of Section 4.1) is used to connect more detailed definitions into early specifications. Finally, Section 9.3 explains how properties (used in the ascending arm of the lifecycle model of Chapter 8) are associated with clauses and with predicates.

9.1 Axiom Sets and Theories

An axiom set is a set of Horn clauses without negation. For example $\{p(X,Y) \leftarrow q(X) \wedge r(Y), q(a) \leftarrow true, r(b) \leftarrow true\}$ is a valid axiom set but $\{\neg p(X) \leftarrow q(X)\}$ and $\{p(X) \leftarrow \neg q(X)\}$ are both invalid as axiom sets. The HANSEL predicates which manipulate axioms sets do not commit to a particular data structure for representing them but, where this needs to be done for explanation or for execution of the specifications we shall enclose such sets within curly brackets as in the examples above.

We write $\tau(S)$ to denote the theory for an axiom set, S. This contains all the Horn clauses which can be deduced from S. For example, $\tau(\{p(X,Y) \leftarrow q(X) \wedge r(Y), q(a) \leftarrow true, r(b) \leftarrow true\})$ is the set: $\{p(X,Y) \leftarrow q(X) \wedge r(Y), p(X,b) \leftarrow q(X), p(a,Y) \leftarrow r(Y), p(a,b) \leftarrow true, q(a) \leftarrow true, r(b) \leftarrow true\}$. These theory definitions are used exclusively in early specification, where they are related by inequalities over the sets they represent. The theory sets themselves are never derived, since they need not be finite, but the inequality definitions are consistent with the sets. For example, $\tau(\{p(s(X)) \leftarrow p(X), p(i)\})$ is the infinite set $\{p(i), p(s(i)), p(s(s(i))), \ldots\}$ and $\tau(\{p(s(X)) \leftarrow p(X), p(i), q\})$ is the infinite set $\{q, p(i), p(s(i)), p(s(s(i))), \ldots\}$ but we can reason with inequalities such as $\tau(\{p(s(X)) \leftarrow p(X), p(i)\}) \subseteq \tau(\{p(s(X)) \leftarrow p(X), p(i), q\})$ without

enumerating the elements of those theories.

9.2 Horn Clauses Using Inequalities between Theories

We shall be using inequalities between theories, as described above, as a basis for refinement of early specifications. This works only for specifications which can be expressed as definitions over axiom sets. We now introduce this concept; explain by example the class of specification problems that it covers[1]; and relate it to the refinement rules which appear in detail in Section 9.4.

The Horn clauses used in HANSEL are standard (see Appendix A for the permitted syntax), except that they may contain as subgoals a number of reserved expressions denoting relations between theories. These are used to describe the constraints imposed by these subgoals and are part of the system of refinement described in detail in Section 9.4. We motivate this form of design using a simple example. Suppose that we want to specify a method for hypothesising a diagnostic theory given a set of permitted observations and a set of diagnostic rules describing how symptoms may be inferred from causes. We shall define a predicate $diagnosis(Observations, Theory, DiagnosticTheory)$, where:

- *Observations* is an axiom set describing the permitted observations. For our example, this might be:

$$\{(oil\_warning\_light \leftarrow true), (engine\_dead \leftarrow true)\}$$

- *Theory* is an axiom set containing the diagnostic rules. For our example, this might be:

$$\left\{ \begin{array}{c} (oil\_warning\_light \leftarrow no\_oil), (engine\_dead \leftarrow no\_spark), \\ (no\_oil \leftarrow broken\_sump), (no\_spark \leftarrow broken\_wiring), (no\_spark \leftarrow bad\_plugs) \end{array} \right\}$$

- *DiagnosticTheory* is a diagnostic theory which allows at least one of the symptoms in the *Theory* to be deduced by adding one or more of the axioms in *Observations*. One instance of this for our example is:

$$\left\{ \begin{array}{c} (oil\_warning\_light \leftarrow no\_oil), (engine\_dead \leftarrow no\_spark), \\ (no\_oil \leftarrow broken\_sump), (no\_spark \leftarrow broken\_wiring), (no\_spark \leftarrow bad\_plugs), \\ (engine\_dead \leftarrow true), (broken\_wiring \leftarrow true) \end{array} \right\}$$

  where $engine\_dead \leftarrow true$ is a symptom appearing in *Observations* (above) and $broken\_wiring \leftarrow true$ is a potential cause which, when added to the *Theory* allows us to deduce that symptom.

[1] We are discussing here the coverage possible in theory, not what we obtain in practice with a support tool, which we discuss in Chapter 12.

Without saying exactly how diagnostic theories are derived, we can give a high level description of key relations between axiom sets in terms of set inequalities. In the base case, we know that the final diagnostic theory must include the current theory. In the recursive case, we want to reduce the size of the observation theory as we go down through the recursion (so we "use up" observations) and we want to increase the size of the current diagnostic theory (by adding observations to it). Formally this is:

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\qquad \tau(\mathit{Theory}) \subseteq \tau(\mathit{DiagnosticTheory}) \end{aligned} \tag{9.1}$$

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\quad \tau(\mathit{Observations}) \supseteq \tau(\mathit{RemainingObservations}) \wedge \\ &\quad \tau(\mathit{Theory}) \subseteq \tau(\mathit{NewTheory}) \wedge \\ &\quad \mathit{diagnosis}(\mathit{RemainingObservations}, \mathit{NewTheory}, \mathit{DiagnosticTheory}) \end{aligned} \tag{9.2}$$

We now can add a little more detail by describing the kind of relation which we will use to satisfy the inequalities above. This involves a classification of types of binary relation that we describe in detail in Sections 9.4.2 and 9.4.3 but is sketched here. The inequality in clause 9.1 is satisfied by a relation which generates from the theory new axioms which are hypothesised causes and adds these to the theory to get the final theory. The first inequality in clause 9.2 is satisfied by a relation which removes axioms from the observation set; while the second inequality in clause 9.2 is satisfied by a relation which adds axioms to the theory set. Formally this is:

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\qquad \mathit{Theory} \overset{new}{\rightsquigarrow} \mathit{Causes} \wedge \\ &\qquad \mathit{union}(\mathit{Theory}, \mathit{Causes}, \mathit{DiagnosticTheory}) \end{aligned} \tag{9.3}$$

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\quad \mathit{Observations} \overset{remove}{\rightsquigarrow} \mathit{RemainingObservations} \wedge \\ &\quad \mathit{Theory} \overset{add}{\rightsquigarrow} \mathit{NewTheory} \wedge \\ &\quad \mathit{diagnosis}(\mathit{RemainingObservations}, \mathit{NewTheory}, \mathit{DiagnosticTheory}) \end{aligned} \tag{9.4}$$

Finally we can be specific about our choice of predicates for the relations introduced above. The relation in clause 9.3 is defined as abductive inference from *Theory* to obtain *Causes* consistent with that theory. The first relation in clause 9.4 is defined as a predicate that generates a singleton set, *ChosenObservations*, containing an element from *Observations* (and generating each element on backtracking). The second relation in clause 9.4 is defined simply by making the *NewTheory* be the union of the *ChosenObservations* and the *Theory*.

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\qquad \mathit{diagnostic\_hypotheses}(\mathit{Theory}, \mathit{Causes}) \wedge \\ &\qquad \mathit{union}(\mathit{Theory}, \mathit{Causes}, \mathit{DiagnosticTheory}) \end{aligned} \tag{9.5}$$

$$\begin{aligned} &\mathit{diagnosis}(\mathit{Observations}, \mathit{Theory}, \mathit{DiagnosticTheory}) \leftarrow \\ &\quad \mathit{select\_observations}(\mathit{Observations}, \mathit{ChosenObservations}, \mathit{RemainingObservations}) \wedge \\ &\quad \mathit{union}(\mathit{Theory}, \mathit{ChosenObservations}, \mathit{NewTheory}) \wedge \\ &\quad \mathit{diagnosis}(\mathit{RemainingObservations}, \mathit{NewTheory}, \mathit{DiagnosticTheory}) \end{aligned} \tag{9.6}$$

This example illustrates two of the basic concepts underpinning HANSEL. These are:

- Specifications are described as Horn Clauses which manipulate sets of axioms, appearing as arguments to the goals in these clauses.
- Inequalities between sets are used to anchor sequences of refinement, ending in conventional Horn clause specifications, which no longer contain inequalities.

A third basic concept in HANSEL is described in the next section. It is that the properties of expressions (such as inequalities) appearing in earlier specifications can help us to check whether we have made appropriate design choices in later, more detailed specifications.

9.3 Use of Properties in Testing

Figure 9.1 illustrates the method used to assist in testing logic programs produced by the HANSEL system. At the start of the method we use the HANSEL tool (which we summarise in Chapter 10) to produce a logic program and associated properties which we expect to hold for the program. We then use an execution system (currently a simple meta-interpreter described in Section 9.7) to answer queries from the program and, in the process of doing this, generate a set of test properties relevant to each query. These are instances of the properties originally associated with the logic program. These test properties are then passed to a testing system (currently a standard Prolog interpreter) to determine whether or not they can be established from the logic program.

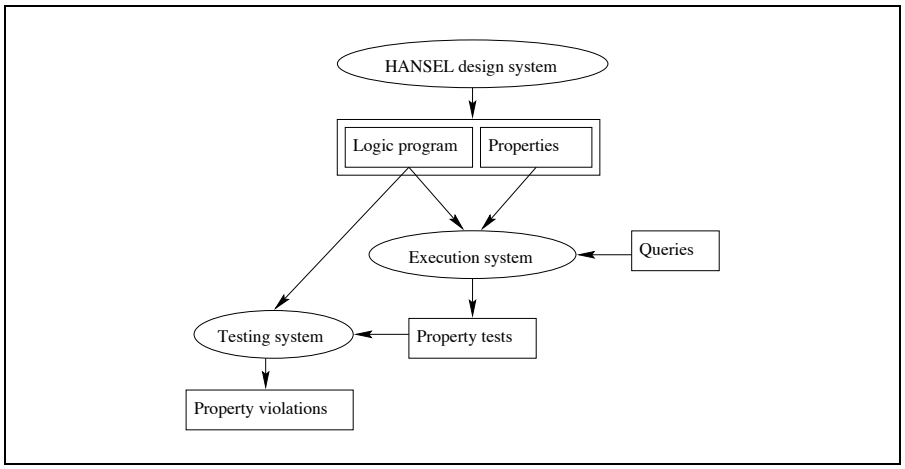


Figure 9.1: Overview of the HANSEL testing method

There are two main limitations to this testing method. The first is insurmountable: we derive tests from executing the specification to produce instances of existentially quantified goals so we only generate those tests which correspond to each particular execution. In other words, we need to choose the right goals to get revealing tests. The second limitation is flexible: although there is no difficulty in generating tests for complex properties it can sometimes be difficult to prove whether or not these properties hold. In the current version of HANSEL we simply use Prolog to test for property violation (see Section 12.3 for examples). We could have used a more sophisticated theorem prover but this strays into specialist areas outside this thesis.

HANSEL recognises two types of property: properties which apply to variables in individual clauses, which we call "clause properties" and properties which apply to predicates, which we call "predicate properties". Later in this chapter, clause properties are associated with rewrites in Section 9.4; predicate properties are associated with skeletons in Section 9.5; and a basic mechanism for using properties in testing is defined in Section 9.7. In Section 12.3 we explain how this method is used on the examples of Chapter 12. The difference between clause and predicate properties is best explained by an example.

The predicate, $sort\_segments(L1, L2)$, is given a list, $L1$, which may contain elements which are integers and generates the list, $L2$, which is identical to $L1$ except that the segments of $L1$ which are integers are sorted in ascending order within that segment. For example, $sort\_segments([4, 3, 5, a, b, 2, 1, c], L2)$ instantiates $L2$ to $[3, 4, 5, a, b, 1, 2, c]$. We might define it as follows; where $sort(S, Ss)$ is true when Ss is the sorted version of S, and *append* is the normal list concatenation predicate. The definition of *sort_segments* uses *segment* to find each integer segment in turn (skipping over any non-integers in the recursion via the third clause). $segment(L, S, R)$ is true if S is an unbroken sequence of integers from the first element of L to the first non-integer in L, with R being the rest of the elements in L.

$$sort\_segments([], []) \tag{9.7}$$

$$sort\_segments(L1, L2) \leftarrow segment(L1, S, R) \wedge sort(S, Ss) \wedge sort\_segments(R, L3) \wedge append(Ss, L3, L2) \tag{9.8}$$

$$sort\_segments([H|T], [H|R]) \leftarrow not(integer(H)) \wedge sort\_segments(T, R) \tag{9.9}$$

$$segment(L, S, R) \leftarrow segment1(L, S, R) \wedge not(S = []) \tag{9.10}$$

$$segment1([], [], []) \tag{9.11}$$

$$segment1([H|T1], [H|T2], R) \leftarrow integer(H) \wedge segment1(T1, T2, R) \tag{9.12}$$

$$segment1([H|T], [], [H|T]) \leftarrow not(integer(H)) \tag{9.13}$$

Suppose that we have written only the definition of the *sort_segments* clauses (*i.e.* clauses 9.7 to 9.9) and we intend to give the task of writing *segment* to someone else. We would like to specify formally (either partially or fully, according to our degree of rigour) what *segment* should do and we would like to place constraints on the way the clauses of *sort_segments* are used, to protect it against misuse. Our partial specification of $segment(L, S, R)$ might require that S contains only integers and that there shouldn't be an integer as the first element of R (if so, we haven't got all of the segment in S).

$$\forall L, S, R.segment(L, S, R) \rightarrow \left(\begin{array}{c} (\forall Xs.Xs \in S \rightarrow integer(Xs)) \wedge \\ not(\exists Xr, T.(R = [Xr|T] \wedge integer(Xr))) \end{array} \right) \qquad (9.14)$$

By checking the definition of *segment* against this requirement we can prevent some incorrect definitions. For instance, if it is defined simply as:

$$segment([H|T], [H], T)$$

then we can demonstrate that this does not meet the requirement of expression 9.14 by showing that the negation of the property can be proved for instances such as $segment([1,2],[1],[2])$ which succeed according to the (incorrect) predicate definition. A proof for this example, done by hand for the purposes of explanation only, is given below. In Section 12.3 we describe the form of proof we actually allow in the current version of HANSEL which, as we mentioned earlier, relies on the standard Prolog interpreter.

$$not\left(\forall L, S, R.segment(L, S, R) \rightarrow \left(\begin{array}{c} (\forall Xs.Xs \in S \rightarrow integer(Xs)) \wedge \\ \uparrow \\ not(\exists Xr, T.(R = [Xr|T] \wedge integer(Xr))) \end{array} \right)\right)$$

$$\exists L, S, R.segment(L, S, R) \quad \wedge \quad not\left(\begin{array}{c} (\forall Xs.Xs \in S \rightarrow integer(Xs)) \wedge \\ not(\exists Xr, T.(R = [Xr|T] \wedge integer(Xr))) \end{array} \right)$$

$$\uparrow \qquad\qquad\qquad\qquad \uparrow$$

$$segment([1,2],[1],[2]) \qquad not(not(\exists Xr, T.([2] = [Xr|T] \wedge integer(Xr))))$$

$$\uparrow$$

$$\exists Xr, T.([2] = [Xr|T] \wedge integer(Xr))$$

$$\uparrow$$

$$[2] = [2|[]] \wedge integer(2)$$

This use of predicate properties gives a way of preventing some ways in which *segment* might be incorrectly defined but does not give complete protection. One particularly nasty form of incorrect definition is to allow *segment* to return as its third argument, R, a list which is not smaller than the input list, $L1$, because this may create an infinite recursion in *sort_segments*, which relies for termination on a well founded ordering from $L1$ through R to the empty list. This mistake is easy to make because if we simply omit from the definition of *segment* in clause 9.10 the test that S is not empty, giving:

$$segment(L, S, R) \leftarrow segment1(L, S, R)$$

then this is capable of generating infinite recursions for goals such as $sort\_segments([a], X)$, since a subgoal in solving this goal is $segment([a], [], [a])$, which would (correctly)

have failed in the original definition. We could protect against this by defining another predicate property for *segment* but it is (arguably) more natural to associate the requirement with the clause where the recursion occurs. We write this as a binary predicate $hansel\_clause(C, \mathcal{T})$ where C is a clause in the specification and $\mathcal{T}$ is a set of properties required to hold for any instance of C used to solve a goal. For our example, the appropriate definition of this is:

$$\begin{array}{lll} hansel\_clause(& sort\_segments(L1, L2) \leftarrow & segment(L1, S, R) \wedge \\ & & sort(S, Ss) \wedge \\ & & sort\_segments(R, L3) \wedge \\ & & append(Ss, L3, L2), \\ & \{\exists X.X \in L1 \ \wedge \ not(X \in R)\} \ ) & \end{array} \tag{9.15}$$

Notice that variables are shared between the clause (in the first argument) and its properties (in the second argument). Constraints like this one come into effect when clauses are used to solve goals, hence instantiating properties. For instance, if the goal $sort\_segments([a], X)$ is matched to the head of the clause in expression 9.15 and we have solved its first subgoal using the erroneous definition of *segment* given above then the entire expression (including the property) is instantiated to:

$$\begin{array}{lll} hansel\_clause(& sort\_segments([a], L2) \leftarrow & segment([a], [], [a]) \wedge \\ & & sort([], Ss) \wedge \\ & & sort\_segments([a], L3) \wedge \\ & & append(Ss, L3, L2), \\ & \{\exists X.X \in [a] \ \wedge \ not(X \in [a])\} \ ) & \end{array}$$

At this point we can see there is a problem because the instantiated property cannot be satisfied: $\exists X.X \in [a] \ \wedge \ not(X \in [a])$ has no solution. This system of clause properties (like expression 9.15) and predicate properties (like expression 9.14) is flexible, since properties can be associated either with predicates or with the variables in any clause of a predicate. It raises an issue, however, of what to do with the properties which we represent. Section 9.7 describes a simple use of properties in testing.

9.4 Refinement Rules

Having introduced in Section 9.2 the basic idea of specification refinement in HANSEL, we now describe the system of rewrites currently used to refine predicate definitions. Along with each rewrite is a set of properties which we anticipate that testers might wish to check in verifying the specification. Rewrites are divided into groups associated with each set-related operator. Section 9.4.1 describes the rewrites from generic relations to theory equalities and inequalities. Sections 9.4.2, 9.4.3 and 9.4.4 describe rewrites from equalities, specialisation inequalities and generalisation inequalities, respectively. Sections 9.4.5 and 9.4.6 describe rewrites for element selection and addition expressions.

Notice that properties associated with rewrites are not required to be complete in the sense that they fully specify all relevant properties of the rewritten

expression. By allowing partiality we give designers of the rewrites the chance to stipulate just those properties which seem of greatest importance. We also do not require designers of rewrites to consider the entire rewrite system when choosing properties of a particular rewrite rule - each is described individually. This introduces some redundancy because later rewrites may include properties defined for earlier ones – for example the property associated with Rewrite 1 (below) also appears in Rewrite 12 (below) even though it can only be reached via Rewrite 1.

Throughout this section, variables beginning with the letter S refer to axiom sets and variables beginning with the letter X refer to elements of those sets. Each rewrite is accompanied by an informal description of the intuition behind it. In these description we use the phrase "$S1$ specialises to $S2$" when the theory of $S1$ is a superset of the theory of $S2$ ($\tau(S1) \supseteq \tau(S2)$) and the phrase "$S1$ generalises to $S2$" when the theory of $S1$ is a subset of the theory of $S2$ ($\tau(S1) \subseteq \tau(S2)$). Notice that the direction of inequalities matters because when introducing predicates to manipulate the axiom sets we assume that the first set in the pair deriving from an inequality is an input to the predicate. Thus $\tau(S1) \supseteq \tau(S2)$ will be refined differently from $\tau(S2) \subseteq \tau(S1)$ despite the fact that these two expressions have the same semantics. This is the reason why there are separate (symmetrical) sets of refinements for theory specialisation (Section 9.4.3) and for theory generalisation (Section 9.4.4).

9.4.1 Refinements from General Relations to Theory Comparators

The main role of refinements in this section is to allow generic relations, which denote only that two axiom sets interact, to be rewritten into inequalities or equalities between axiom sets. To allow for the possibility that the interaction between axiom sets may not be in either of these forms there is also a rewrite to an unconstrained binary relation, allowing us later to introduce a skeleton directly to relate the axiom sets.

Rewrite 1 *If we have a general relation between $S1$ and $S2$ then $S1$ might specialise to $S2$. If so, we expect anything provable from $S2$ to be provable from $S1$.*

$$\begin{array}{lcl} \textit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \tau(S1) \supseteq \tau(S2) \\ \textit{Properties} & : & \{\forall X.S2 \vdash X \rightarrow S1 \vdash X\} \end{array}$$

Rewrite 2 *If we have a general relation between $S1$ and $S2$ then $S2$ might specialise to $S1$. If so, we expect anything provable from $S1$ to be provable from $S2$.*

$$\begin{array}{lcl} \textit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \tau(S2) \supseteq \tau(S1) \\ \textit{Properties} & : & \{\forall X.S1 \vdash X \rightarrow S2 \vdash X\} \end{array}$$

Rewrite 3 *If we have a general relation between $S1$ and $S2$ then $S1$ might generalise to $S2$. If so, we expect anything provable from $S1$ to be provable from*

$S2$.

$$\begin{array}{lcl} \mathit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \tau(S1) \subseteq \tau(S2) \\ \mathit{Properties} & : & \{\forall X.S1 \vdash X \rightarrow S2 \vdash X\} \end{array}$$

Rewrite 4 *If we have a general relation between $S1$ and $S2$ then $S2$ might generalise to $S1$. If so, we expect anything provable from $S2$ to be provable from $S1$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \tau(S2) \subseteq \tau(S1) \\ \mathit{Properties} & : & \{\forall X.S2 \vdash X \rightarrow S1 \vdash X\} \end{array}$$

Rewrite 5 *If we have a general relation between $S1$ and $S2$ then $S1$ might equal $S2$. If so, we expect anything provable from $S1$ to be provable from $S2$ and vice versa.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \tau(S1) = \tau(S2) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X.S1 \vdash X \rightarrow S2 \vdash X, \\ \forall X.S2 \vdash X \rightarrow S1 \vdash X \end{array} \right\} \end{array}$$

Rewrite 6 *If we have a general relation between $S1$ and $S2$ then this may be via general relations to a third set, $S3$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; \left(\begin{array}{c} S1 \approx S3 \wedge \\ S3 \approx S2 \end{array} \right) \\ \mathit{Properties} & : & \{\} \end{array}$$

Rewrite 7 *A general relation between $S1$ and $S2$ might be obtained through a predicate directly relating $S1$ and $S2$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & S1 \approx S2 \;\Rightarrow\; S1 \overset{unconstrained}{\rightsquigarrow} S2 \\ \mathit{Properties} & : & \{\} \end{array}$$

9.4.2 Refinements of Theory Equalities

The refinements of this section allow equalities between axiom sets to be defined either in terms of a pair of inequalities or by direct comparison of the sets. In direct comparison we distinguish between the case where the sets have the same elements but with no commitment to ordering of those elements (denoted by $\mathit{equal}(S1, S2)$ below) and the case where the two sets have identical data structures (denoted by $\mathit{identical}(S1, S2)$ below). The latter case is provided because it is common when writing specifications to want to say that two variables are identical, in which case HANSEL automatically binds the two variables into a single variable and the $\mathit{identical}(S1, S2)$ subgoal disappears from the specification. This is considerably neater than accumulating numerous explicit set equalities in the bodies of clauses.

Rewrite 8 *An equality between $S1$ and $S2$ may be established by showing that $S1$ specialises to $S2$ and $S1$ generalises to $S2$. If so, we expect anything provable*

from $S1$ to be provable from $S2$ and vice versa.

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) = \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} \tau(S1) \supseteq \tau(S2) \wedge \\ \tau(S1) \subseteq \tau(S2) \end{array}\right) \\ \mathit{Properties} & : & \left\{\begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S2 \vdash X1, \\ \forall X2.S2 \vdash X2 \rightarrow S1 \vdash X2 \end{array}\right\} \end{array}$$

Rewrite 9 *An equality between $S1$ and $S2$ may be established by showing that $S1$ and $S2$ contain the same elements. If so, we expect any element of one set to appear in the other.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) = \tau(S2) \;\Rightarrow\; \mathit{equal}(S1, S2) \\ \mathit{Properties} & : & \left\{\begin{array}{c} \forall X1.X1 \in S1 \rightarrow X1 \in S2 \\ \forall X2.X2 \in S2 \rightarrow X2 \in S1 \end{array}\right\} \end{array}$$

Rewrite 10 *An equality between $S1$ and $S2$ may be established by showing that $S1$ and $S2$ are identical data structures.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) = \tau(S2) \;\Rightarrow\; \mathit{identical}(S1, S2) \\ \mathit{Properties} & : & \{\} \end{array}$$

9.4.3 Refinements of Theory Specialisation

In this section we deal with refinements of specialisation relations between axiom sets. The first four of these rewrites introduce "hooks" for predicate definition via skeleton introduction (the hooks being the $S1 \overset{R}{\rightsquigarrow} S3$ expressions). The rest are consistent with standard set theory. Notice that it would have been possible to extend this set of rewrites (and their duals in the next section) with rewrites to inequalities between interpretations[2] of the axiom sets, since these are always subsets of the corresponding theory. This, however, introduces yet another subtlety into the refinement process so we leave it for further work.

Rewrite 11 *A specialisation from $S1$ to $S2$ may be established by a predicate which creates a new set, $S3$, by adding axioms which are logical consequences of $S1$ to $S1$ (thus not changing its theory), then showing that $S3$ specialises to $S2$. If so, we expect everything provable from $S1$ to be provable from $S3$ and vice versa; we expect $S3$ to contain an element which is not in $S1$; and we expect everything provable from $S2$ to be provable from $S3$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} S1 \overset{expand}{\rightsquigarrow} S3 \wedge \\ \tau(S3) \supseteq \tau(S2) \end{array}\right) \\ \mathit{Properties} & : & \left\{\begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S3 \vdash X1, \\ \forall X2.S3 \vdash X2 \rightarrow S1 \vdash X2, \\ \exists X3.X3 \in S3 \;\wedge\; \mathit{not}(X3 \in S1), \\ \forall X4.S2 \vdash X4 \rightarrow S3 \vdash X4 \end{array}\right\} \end{array}$$

[2] An interpretation being the set of ground goals deducible from the set of Horn clauses.

Rewrite 12 *A specialisation from $S1$ to $S2$ may be established by a predicate which restricts the theory of $S1$ to give $S2$. If so, everything provable from $S2$ should be provable from $S1$ and there should be something which is provable from $S1$ but not $S2$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; S1 \overset{specialise}{\rightsquigarrow} S2 \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S2 \vdash X1 \rightarrow S1 \vdash X1, \\ \exists X2.S1 \vdash X2 \;\wedge\; not(S2 \vdash X2) \end{array} \right\} \end{array}$$

Rewrite 13 *A specialisation from $S1$ to $S2$ may be established by a predicate which removes axioms from $S1$ to give $S2$. If so, there should be some element of $S1$ which is not in $S2$ but all elements of $S2$ should be in $S1$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; S1 \overset{remove}{\rightsquigarrow} S2 \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.X1 \in S2 \rightarrow X1 \in S1, \\ \exists X2.X2 \in S1 \;\wedge\; not(X2 \in S2) \end{array} \right\} \end{array}$$

Rewrite 14 *A specialisation from $S1$ to $S2$ may be established by removing from $S1$ a subset of it, $S3$, which is determined by some predicate. If so, the elements in $S3$ should appear in $S1$ but not in $S2$, and all the elements of $S2$ should appear in $S1$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} S1 \overset{subset}{\rightsquigarrow} S3 \wedge \\ remove(S3, S1, S2) \end{array} \right) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.X1 \in S3 \rightarrow X1 \in S1, \\ \forall X2.X2 \in S3 \rightarrow not(X2 \in S2), \\ \forall X3.X3 \in S2 \rightarrow X3 \in S1) \end{array} \right\} \end{array}$$

Rewrite 15 *A specialisation from $S1$ to $S2$ may be established by showing that an intermediate set, $S3$, is a specialisation of $S1$ and that $S3$ specialises to $S2$. If so, we expect anything provable from $S3$ to be provable from $S1$ and anything provable from $S2$ to be provable from $S3$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} \tau(S1) \supseteq \tau(S3) \wedge \\ \tau(S3) \supseteq \tau(S2) \end{array} \right) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S3 \vdash X1 \rightarrow S1 \vdash X1, \\ \forall X2.S2 \vdash X2 \rightarrow S3 \vdash X2 \end{array} \right\} \end{array}$$

Rewrite 16 *A specialisation from $S1$ to $S2$ may be established by showing that $S2$ is the union of two sets, $S3$ and $S4$, which are independent specialisations of $S1$. If so, anything provable from $S3$ or $S4$ should be provable in $S1$ and $S2$*

should contain exactly those elements that appear in either $S3$ or $S4$.

$$\begin{array}{lcl} \textit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow \left(\begin{array}{c} \tau(S1) \supseteq \tau(S3) \wedge \\ \tau(S1) \supseteq \tau(S4) \wedge \\ \textit{union}(S3, S4, S2) \end{array} \right) \\ \textit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S3 \vdash X1 \rightarrow S1 \vdash X1, \\ \forall X2.S4 \vdash X2 \rightarrow S1 \vdash X2, \\ \forall X3.X3 \in S3 \rightarrow X3 \in S2, \\ \forall X4.X4 \in S4 \rightarrow X4 \in S2, \\ \forall X5.X5 \in S2 \rightarrow (X5 \in S3 \;\vee\; X5 \in S4) \end{array} \right\} \end{array}$$

Rewrite 17 *A specialisation from $S1$ to $S2$ may be established by showing that $S1$ equals $S2$. If so, anything provable in $S1$ should be provable in $S2$ and vice versa.*

$$\begin{array}{lcl} \textit{Rewrite} & : & \tau(S1) \supseteq \tau(S2) \;\Rightarrow\; \tau(S1) = \tau(S2) \\ \textit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S2 \vdash X1, \\ \forall X2.S2 \vdash X2 \rightarrow S1 \vdash X2 \end{array} \right\} \end{array}$$

9.4.4 Refinements of Theory Generalisation

In this section we deal with refinements of generalisation relations between axiom sets. These are the duals of the refinements in Section 9.4.3.

Rewrite 18 *A generalisation from $S1$ to $S2$ may be established by a predicate which creates a new set, $S3$, by adding axioms to $S1$ without changing its theory, then showing that $S3$ generalises to $S2$. If so, we expect everything provable from $S1$ to be provable from $S3$ and vice versa; we expect $S3$ to contain an element which is not in $S1$; and we expect everything provable from $S3$ to be provable from $S2$.*

$$\begin{array}{lcl} \textit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow \left(\begin{array}{c} S1 \overset{expand}{\rightsquigarrow} S3 \wedge \\ \tau(S3) \subseteq \tau(S2) \end{array} \right) \\ \textit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S3 \vdash X1, \\ \forall X2.S3 \vdash X2 \rightarrow S1 \vdash X2, \\ \exists X3.X3 \in S3 \;\wedge\; \textit{not}(X3 \in S1), \\ \forall X4.S3 \vdash X4 \rightarrow S2 \vdash X4 \end{array} \right\} \end{array}$$

Rewrite 19 *A generalisation from $S1$ to $S2$ may be established by a predicate which enlarges the theory of $S1$ to give $S2$. If so, everything provable from $S1$ should be provable from $S2$ and there should be something which is provable from $S2$ but not $S1$.*

$$\begin{array}{lcl} \textit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; S1 \overset{generalise}{\rightsquigarrow} S2 \\ \textit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S2 \vdash X1, \\ \exists X2.S2 \vdash X2 \;\wedge\; \textit{not}(S1 \vdash X2) \end{array} \right\} \end{array}$$

Rewrite 20 *A generalisation from $S1$ to $S2$ may be established by a predicate which adds axioms to $S1$ to give $S2$. If so, there should be some element of $S2$ which is not in $S1$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; S1 \overset{add}{\rightsquigarrow} S2 \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.X1 \in S1 \rightarrow X1 \in S2, \\ \exists X2.X2 \in S2 \;\wedge\; not(X2 \in S1) \end{array} \right\} \end{array}$$

Rewrite 21 *A generalisation from $S1$ to $S2$ may be established by adding to $S1$ a new axiom set, $S3$, which is determined by some predicate. If so, $S3$ should be non-empty; no element of $S3$ should appear in $S1$; and $S2$ should contain exactly those elements which are in $S1$ and $S3$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} S1 \overset{new}{\rightsquigarrow} S3 \wedge \\ union(S3, S1, S2) \end{array} \right) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \exists X1.X1 \in S3, \\ \forall X2.X2 \in S3 \rightarrow not(X2 \in S1), \\ \forall X3.X3 \in S3 \rightarrow X3 \in S2, \\ \forall X4.X4 \in S1 \rightarrow X4 \in S2, \\ \forall X5.X5 \in S2 \rightarrow (X5 \in S1 \;\vee\; X5 \in S3) \end{array} \right\} \end{array}$$

Rewrite 22 *A generalisation from $S1$ to $S2$ may be established by showing that an intermediate set, $S3$ is a generalisation of $S1$ and that $S3$ generalises to $S2$. If so, we expect anything provable from $S1$ to be provable from $S3$ and anything provable from $S3$ to be provable from $S2$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} \tau(S1) \subseteq \tau(S3) \wedge \\ \tau(S3) \subseteq \tau(S2) \end{array} \right) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S3 \vdash X1, \\ \forall X2.S3 \vdash X2 \rightarrow S2 \vdash X2 \end{array} \right\} \end{array}$$

Rewrite 23 *A generalisation from $S1$ to $S2$ may be established by showing that $S2$ is the intersection of two sets, $S3$ and $S4$, which are independent generalisations of $S1$. If so, anything provable from $S1$ should be provable in $S3$ and in $S4$; and $S2$ should contain exactly those elements that appear in both $S3$ or $S4$.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; \left(\begin{array}{c} \tau(S1) \subseteq \tau(S3) \wedge \\ \tau(S1) \subseteq \tau(S4) \wedge \\ intersection(S3, S4, S2) \end{array} \right) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S3 \vdash X1, \\ \forall X2.S1 \vdash X2 \rightarrow S4 \vdash X2, \\ \forall X3.(X3 \in S3 \;\wedge\; X3 \in S4) \rightarrow X3 \in S2, \\ \forall X4.X4 \in S2 \rightarrow (X4 \in S3 \;\wedge\; X4 \in S4) \end{array} \right\} \end{array}$$

Rewrite 24 *A generalisation from $S1$ to $S2$ may be established by showing that $S1$ equals $S2$. If so, anything provable in $S1$ should be provable in $S2$ and vice versa.*

$$\begin{array}{lcl} \mathit{Rewrite} & : & \tau(S1) \subseteq \tau(S2) \;\Rightarrow\; \tau(S1) = \tau(S2) \\ \mathit{Properties} & : & \left\{ \begin{array}{c} \forall X1.S1 \vdash X1 \rightarrow S2 \vdash X1, \\ \forall X2.S2 \vdash X2 \rightarrow S1 \vdash X2 \end{array} \right\} \end{array}$$

9.4.5 Refinements of Element Selection

The refinements of this section are all rewrites for the expression $\mathcal{S}(X1, S1, S2)$ which denotes that an element, X is selected from axiom set $S1$ to give the remaining axioms $S2$. These expressions do not appear in the rewrites of Sections 9.4.1 to 9.4.4, so they do not arise in the early stages of HANSEL design. They appear in the definitions of skeletons, as we shall show in Section 9.5. We supply three sorts of selection: by taking the first element of the axiom set; by taking some element from the set (regardless of position); and selecting some element which passes a test, $\mathcal{T}(X)$ (see Section 4.1 for a discussion of test subgoals).

Rewrite 25 *If we wish to select an element, X, from $S1$ to give $S2$ then we might select the first element of $S1$.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{S}(X, S1, S2) \Rightarrow \textit{select\_first\_element}(X, S1, S2) \\ \textit{Properties} & : & \{\} \end{array}$$

Rewrite 26 *If we wish to select an element, X, from $S1$ to give $S2$ then we might select any element of $S1$.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{S}(X, S1, S2) \Rightarrow \textit{select\_some\_element}(X, S1, S2) \\ \textit{Properties} & : & \{\} \end{array}$$

Rewrite 27 *If we wish to select an element, X, from $S1$ to give $S2$ then we might select any element of $S1$ which satisfies some test.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{S}(X, S1, S2) \Rightarrow \left(\begin{array}{c} \textit{select\_some\_element}(X, S1, S2) \wedge \\ \mathcal{T}(X) \end{array} \right) \\ \textit{Properties} & : & \{\} \end{array}$$

9.4.6 Refinements of Element Addition

The refinements of this section are duals of those in Section 9.4.5. They all are rewrites for the expression $\mathcal{A}(X1, S1, S2)$ which denotes that an element, X is added to axiom set $S1$ to give the larger axiom set $S2$.

Rewrite 28 *If we wish to add an element, X, to $S1$ to give $S2$ then we might add the first element of $S1$.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{A}(X, S1, S2) \Rightarrow \textit{add\_element\_first}(X, S1, S2) \\ \textit{Properties} & : & \{\} \end{array}$$

Rewrite 29 *If we wish to add an element, X, to $S1$ to give $S2$ then we might add any element of $S1$.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{A}(X, S1, S2) \Rightarrow \textit{add\_element\_somewhere}(X, S1, S2) \\ \textit{Properties} & : & \{\} \end{array}$$

Rewrite 30 *If we wish to add an element, X, to $S1$ to give $S2$ then we might add any element of $S1$ which satisfies some test.*

$$\begin{array}{lll} \textit{Rewrite} & : & \mathcal{A}(X, S1, S2) \Rightarrow \left(\begin{array}{c} \textit{add\_element\_somewhere}(X, S1, S2) \wedge \\ \mathcal{T}(X) \end{array} \right) \\ \textit{Properties} & : & \{\} \end{array}$$

9.5 Skeletons for Axiom Sets

Some of the rewrites of Section 9.4 introduce into clauses subgoals of the form $S1 \overset{R}{\rightsquigarrow} S2$, where $S1$ and $S2$ are axiom sets and R is the type of relation which holds between them. These are the "hooks" to which we attach binary predicates, and HANSEL assists in defining these predicates by providing a library of partially completed definitions intended to provide the basic flow of control of the relation. We call these "skeletons" because the aim is similar to the partial programs by the same name used in Section 4.1 - although in HANSEL they are being used within a different design method and are defined over axiom sets rather than the usual Prolog data structures.

Each skeleton is described using the following three components:

Parameters : A term of the form $(P, S1, S2, Rs)$, where P is the predicate name; $S1$ and $S2$ are the left and right hand sides of the relation $S1 \overset{R}{\rightsquigarrow} S2$ which the skeleton is defining; and Rs is a list of predicate names used as subgoals in the skeleton.

Code : A set of Horn clauses defining the partial specification supplied by the skeleton.

Properties : A set of properties associated with the skeleton.

Skeletons are instantiated through their parameters, with the instantiations being propagated through the code and properties via shared variable names. Parameters $S1$ and $S2$ are instantiated by matching to the relation $S1 \overset{R}{\rightsquigarrow} S2$, and the other parameters are instantiated by the designer via a dialogue described in Section 10.4. Skeletons are partial specifications because they may contain HANSEL expressions which need further refinement[3] and they may be extended by adding argument slices, as described in Section 10.6.

Since skeletons are task or domain specific it is necessary to have a methodical way of populating the library of skeletons. Section 9.5.1 summarises the main groups of skeletons currently in the HANSEL library. Section 9.5.2 shows how one of these groups was designed, explaining how a particular class of skeletons and their properties is covered methodically.

[3]In particular the expressions $\mathcal{S}(X, S, Sr)$ or $\mathcal{A}(X, S, Sa)$ are used to denote that some predicate should select an element from or add an element to set S, as described in detail in Sections 9.4.5 and 9.4.6.

9.5.1 Groups of Skeletons

The skeletons in HANSEL are task specific and are organised into groups around standard types of logic programming task, each understood as a relation between the given axiom set in the first argument of the binary relation and the derived axiom set in the second argument. Full definitions appear in Appendix C. The current groups of tasks are:

Traversal through all the elements of a set (Appendix C.1) where the relation applies a mapping from the first set to the second set by mapping each element of the first set individually. An example of this type of skeleton (which is skeleton 2 in Appendix C.1) is:

$$\begin{aligned}
P(S1,S2) \leftarrow\ & \mathit{empty}(S1)\wedge \\
& \mathit{empty}(S2), \\
P(S1,S2) \leftarrow\ & \mathcal{S}_1(X1,S1,T1)\wedge \\
& R(X1,X2)\wedge \\
& P(T1,T2)\wedge \\
& \mathcal{A}_1(X2,T2,S2), \\
P(S1,S2) \leftarrow\ & \mathcal{S}_2(X1,S1,T1)\wedge \\
& \mathit{not}(R(X1,X2))\wedge \\
& P(T1,T2)\wedge \\
& \mathcal{A}_2(X1,T2,S2)
\end{aligned}$$

which maps each element, $X1$, of $S1$ which satisfies $R(X1,X2)$ to an element, $X2$ of $S2$ and if $X1$ does not satisfy $R(X1,X2)$ then $X1$ itself appears in $S2$. An instance of this skeleton is a predicate *pseudonyms* which maps each name in a set, represented as a list, to the pseudonym for that name if it exists or to the original name if a pseudonym does not exist. An instance of the skeleton above for this (with P bound to *pseudonyms* and R bound to *has_pseudonym*) is:

$$\begin{aligned}
\mathit{pseudonyms}(S1,S2) \leftarrow\ & \mathit{empty}(S1)\wedge \\
& \mathit{empty}(S2), \\
\mathit{pseudonyms}(S1,S2) \leftarrow\ & \mathcal{S}_1(X1,S1,T1)\wedge \\
& \mathit{has_pseudonym}(X1,X2)\wedge \\
& \mathit{pseudonyms}(T1,T2)\wedge \\
& \mathcal{A}_1(X2,T2,S2), \\
\mathit{pseudonyms}(S1,S2) \leftarrow\ & \mathcal{S}_2(X1,S1,T1)\wedge \\
& \mathit{not}(\mathit{has_pseudonym}(X1,X2))\wedge \\
& \mathit{pseudonyms}(T1,T2)\wedge \\
& \mathcal{A}_2(X1,T2,S2)
\end{aligned}$$

If we then refine the selection and addition subgoals using Rewrites 25 and 28; then define $\mathit{select\_first\_element}(X,S1,S2)$ as $S1 = [X1|S2]$; then define $\mathit{add\_element\_first}(X,S1,S2)$ as $S2 = [X1|S1]$ we end up with the following changes to subgoals:

- $\mathcal{S}_1(X1, S1, T1)$ becomes $S1 = [X1|T1]$.
- $\mathcal{A}_1(X2, T2, S2)$ becomes $S2 = [X2|T2]$.
- $\mathcal{S}_2(X1, S1, T1)$ becomes $S1 = [X1|T1]$.
- $\mathcal{A}_2(X1, T2, S2)$ becomes $S2 = [X2|T2]$.

and if these unifications are pushed into the head of each clause we have a conventional list traversal.

$$
\begin{array}{ll}
pseudonyms([], []) & \\
pseudonyms([X1|T1], [X2|T2]) \leftarrow & has\_pseudonym(X1, X2) \wedge \\
 & pseudonyms(T1, T2) \\
pseudonyms([X1|T1], [X1|T2]) \leftarrow & not(has\_pseudonym(X1, X2)) \wedge \\
 & pseudonyms(T1, T2)
\end{array}
$$

Searching through some of the elements of a set (Appendix C.2) where the relation considers some of the elements of the first set, eventually finding one where a mapping to the second set can be obtained. An example of this skeleton (which is skeleton 15 in Appendix C.2) is:

$$
\begin{array}{ll}
P(S1, S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1) \wedge \\
 & singleton(X1, S2), \\
P(S1, S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & not(R(X1)) \wedge \\
 & P(T1, S2)
\end{array}
$$

which selects elements of set $S1$ until one is selected which satisfies the test $R(X1)$, in which case $S2$ is the singleton set containing $X1$. An instance of this skeleton is a predicate *search_for_integer* which searches a set, represented using a list, for the first integer found. We can derive this from the skeleton following a procedure similar to that for the previous example, the result being:

$$
\begin{array}{ll}
search\_for\_integer([X1|T1], [X1]) \leftarrow & integer(X1) \\
search\_for\_integer([X1|T1], S2) \leftarrow & not(integer(X1)) \wedge \\
 & search\_for\_integer(T1, S2)
\end{array}
$$

Applying deductive inference to a set (Appendix C.3) where a system of deductive inference is applied to the first axiom set, allowing one or more instances of a selected goal to be obtained and either be added to the first set or independently form the second set. An example of this skeleton (which is skeleton 24 in Appendix C.3) is:

$$
\begin{array}{ll}
P(S1, S2) \leftarrow & R(S1, G) \wedge \\
 & deduce(D, S1 \vdash G) \wedge \\
 & singleton(G, S3) \wedge \\
 & union(S3, S1, S2)
\end{array}
$$

which selects a goal, G, relevant to axiom set $S1$ and finds a proof of G from $S1$ using deductive system D, adding the instantiated goal to $S1$ to produce $S2$. This link to deductive systems is explained in more detail in Section 9.6.

Applying abductive inference to a set (Appendix C.4) where a system of abductive inference is applied to the first axiom set, allowing the axioms necessary to conclude one or more instances of a selected goal to be obtained and either be added to the first set or independently form the second set. An example of this skeleton (which is skeleton 28 in Appendix C.4) is:

$$\begin{aligned} P(S1, S2) \leftarrow\ & R(S1, G) \wedge \\ & \mathit{abduce}(A, S1 \vdash G, Ae) \wedge \\ & \mathit{union}(Ae, S1, S2) \end{aligned}$$

which selects a goal, G, relevant to axiom set $S1$ and finds, using abductive inference system A, a set of additional axioms, Ae, needed to prove G from $S1$, adding these new axioms to $S1$ to give $S2$. Further details of the link to abductive systems are given in Section 9.6.

Applying a relation directly to a set (Appendix C.5) where a relation maps the first set, in total, to one or more elements of the second set. An example of this skeleton (which is skeleton 34 in Appendix C.5) is:

$$\begin{aligned} P(S1, S2) \leftarrow\ & R(S1, X) \wedge \\ & \mathit{singleton}(X, S3) \wedge \\ & \mathit{union}(S3, S1, S2) \end{aligned}$$

which constructs set $S2$ by adding to set $S1$ the element, X, obtained by solving $R(S1, X)$. Since this is a skeleton in a "catch-all" group of tasks, it can be used to obtain similar effects to other, more specific, forms of skeleton. For instance, the deduction skeleton described earlier can be obtained using the generic skeleton above and defining R as:
$R(S1, X) \leftarrow R1(S1, X) \wedge \mathit{deduce}(D, S1 \vdash X)$.

9.5.2 Constructing a Group of Skeletons

The previous section summarised the groups of skeletons currently in the HANSEL library. We now examine the first of those groups in more detail to explain how it was populated. This group concerns predicates which map elements of a given set onto elements of a derived set. The completed group of skeletons appears in Appendix C.1. The sorts of mappings we consider on elements are either single mappings via relation R from element $X1$ to element $X2$, written $R(X1, X2)$ in the skeleton definitions, or exhaustive mappings of R from $X1$ to the Set of corresponding $X2$ elements, written $\mathit{setof}(X2, R(X1, X2), Set)$ in the skeleton

definitions. Later in this section we depict single mappings using the picture and exhaustive mappings using the picture . In some skeletons the mapping relation need not hold for all elements, in which case we can either choose to carry the original element across to the new set or leave it behind. We depict these situations as and respectively.

Given that these are the forms of element mapping that we allow and that our skeletons will apply these by traversing the set, there are twelve possible combinations, which we depict in Figure 9.2. There are two subgroups of these: on the left are the ones where the mappings between elements place only the mapped elements in the derived set (so a single mapping $R(X1, X2)$ places $X2$ in the derived set if $X1$ is in the given set); on the right are the ones where mappings place both original and mapped elements in the derived set (so a single mapping $R(X1, X2)$ places both $X1$ and $X2$ in the derived set). Within each of these subgroups we have a split between single mappings (numbers 1, 2 and 9 in the first group and 5, 6 and 11 in the second) or exhaustive mappings (numbers 3, 4 and 10 in the first group and 7, 8 and 12 in the second). Within these smaller subgroups we can have either a mapping for every element; for some elements with non-mapped elements carried across; or for some elements with non-mapped elements left out. This gives us skeletons 1 to 12 in Appendix C.1.

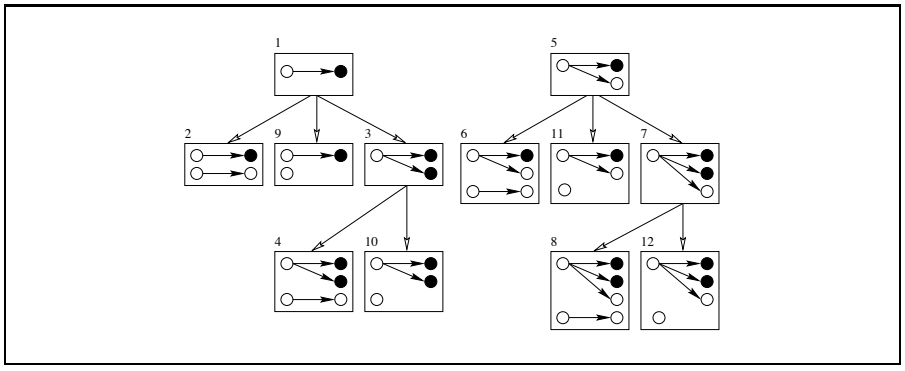


Figure 9.2: Different forms of traversal skeleton

We now look at these skeletons more formally, with the aim of assigning properties to them and using these properties to raise our confidence that we have considered all the possibilities for this type of task. To structure our analysis we can divide these properties into two groups: those which might hold if we observe an element in the given set (which we name $S1$) and those which hold if an element belongs to the derived set (which we name $S2$). We consider each of these in turn:

For an element ($X1$) of the given set ($S1$) there are three sets of constraints:

- A mapping, R, may hold from $X1$ to $X2$ and this must either put

$X2$ into $S2$ or put both $X1$ and $X2$ into $S2$. Formally:

$$X1 \in S1 \rightarrow \exists X2.R(X1, X2) \wedge X2 \in S2 \quad (9.16)$$
$$X1 \in S1 \rightarrow \exists X2.R(X1, X2) \wedge X1 \in S2 \wedge X2 \in S2 \quad (9.17)$$

All the skeletons in the group must satisfy either constraint 9.16 or constraint 9.17. In Figure 9.2 the group on the left satisfies constraint 9.16 while the group on the right satisfies constraint 9.17.

- A mapping, R, may not hold from $X1$ to $X2$, in which case we may or may not require $X1$ to appear in $S2$. Formally:

$$X1 \in S1 \rightarrow not(\exists X2.R(X1, X2)) \wedge X1 \in S2 \quad (9.18)$$
$$X1 \in S1 \rightarrow not(\exists X2.R(X1, X2)) \quad (9.19)$$

We can combine either constraint 9.18 or 9.19 disjunctively with either of constraints 9.16 or 9.17.

- If a mapping holds from $X1$ to $X2$ (as per the constraints above) then we may require it to be exhaustive. Formally:

$$X1 \in S1 \rightarrow (\forall X2.R(X1, X2) \rightarrow X2 \in S2) \quad (9.20)$$

This constraint applies only when we know a mapping exists so it can be combined only with constraint 9.16 or constraint 9.17.

For an element ($X2$) of the given set ($S2$) there are two sets of constraints:

- A mapping may exist which put $X2$ there by applying R to some $X1$ in $S1$ and, if so, $X1$ may or may not appear in $S2$ along with $X2$. Formally:

$$X2 \in S2 \rightarrow \exists X1.R(X1, X2) \wedge X1 \in S1 \quad (9.21)$$
$$X2 \in S2 \rightarrow \exists X1.R(X1, X2) \wedge X1 \in S1 \wedge X1 \in S2 \quad (9.22)$$

Constraint 9.21 applies only if constraint 9.16 applies. Constraint 9.22 applies only if constraint 9.17 applies.

- A mapping may not exist but $X2$ appears in $S1$. Formally:

$$X2 \in S2 \rightarrow not(\exists X1.R(X1, X2)) \wedge X2 \in S1 \quad (9.23)$$

This constraint applies only if constraint 9.17 or 9.18 applies and is disjunctive with either constraint 9.21 or constraint 9.22.

The conditions stipulated above allow only twelve combinations of constraints 9.16 to 9.23. Each of these corresponds to one of the twelve skeletons described on the diagram of Figure 9.2. The table below shows the combinations, the left column giving the constraints on $X1$, the middle column giving the corresponding constraints on $X2$ and the right column identifying the skeleton which should satisfy the constraints.

Constraint on $X1 \in S1$	Constraint on $X2 \in S2$	Skeleton
9.16	9.21	1
9.16 $\vee$ 9.18	9.21 $\vee$ 9.23	2
9.16 $\vee$ 9.19	9.21	9
9.16 $\wedge$ 9.20	9.21	3
(9.16 $\wedge$ 9.20) $\vee$ 9.18	9.21 $\vee$ 9.23	4
(9.16 $\wedge$ 9.20) $\vee$ 9.19	9.21	10
9.17	9.22 $\vee$ 9.23	5
9.17 $\vee$ 9.18	9.22 $\vee$ 9.23	6
9.17 $\vee$ 9.19	9.22 $\vee$ 9.23	11
9.17 $\wedge$ 9.20	9.22 $\vee$ 9.23	7
(9.17 $\wedge$ 9.20) $\vee$ 9.18	9.22 $\vee$ 9.23	8
(9.17 $\wedge$ 9.20) $\vee$ 9.19	9.22 $\vee$ 9.23	12

The final step is to define the partial program for each skeleton, such that it performs the mapping sketched in Figure 9.2 and satisfies the properties given above. As an example we choose skeleton 2 from Appendix C.1, the definition of which was discussed earlier in Section 9.5.1. This is the leftmost box in Figure 9.2 and, by assembling the constraints given in the table above, its properties are:

$$X1 \in S1 \quad \rightarrow \quad \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2 \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \ \wedge \ X1 \in S2 \end{array} \right) \tag{9.24}$$

$$X2 \in S2 \quad \rightarrow \quad \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right) \tag{9.25}$$

These properties require for each element of $S1$ that:

- If the mapping relation, R, from $X1$ to $X2$ holds then $X2$ appears in $S2$. This can be ensured by defining a recursive clause which under the given conditions selects each appropriate $X1$ from $S1$ and adds the corresponding $X2$ to $S2$. Formally:

$$\begin{array}{rl} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & R(X1,X2) \wedge \\ & P(T1,T2) \wedge \\ & \mathcal{A}_1(X2,T2,S2), \end{array} \tag{9.26}$$

- If the mapping relation, R, doesn't apply to $X1$ then $X1$ itself appears in $S2$. This can be ensured by defining a recursive clause which under the given conditions selects each appropriate $X1$ from $S1$ and adds it to $S2$. Formally:

$$\begin{array}{rl} P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & \mathit{not}(R(X1,X2)) \wedge \\ & P(T1,T2) \wedge \\ & \mathcal{A}_2(X1,T2,S2) \end{array} \tag{9.27}$$

Finally, we need to define the base case where no elements appear in $S1$, which is:

$$P(S1, S2) \leftarrow \begin{array}{l} empty(S1) \wedge \\ empty(S2), \end{array} \tag{9.28}$$

These clauses give the partial specification for skeleton 2 as it appears in Appendix C.1.

9.6 Specialised Sets of Inference Rules

One of the devices used to give flexibility to some of the skeletons summarised in Section 9.5 is to define general purpose deduction or abduction predicates which take as an argument a "flag" identifying a set of inference rules to be used in the deductive or abductive reasoning. This gives a way of separating specialised forms of inference relevant to different parts of a specification while retaining a standard approach to applying the inference rules.

All use of deductive proof rules is through the predicate $deduce(F, S \vdash X)$ which is true if there is a deductive proof of X from axiom set S using the inference rules indexed by F. This is the case if there is a deduction rule of type F which can establish $S \vdash X$ given conditions, C, and sub-proofs, Sp, and C holds and all of the subproofs are obtainable (via recursive application of the deduction definition to each sub-proof).

$$deduce(F, Proof) \leftarrow \begin{array}{l} deduction\_rule(F, Proof, C, Sp) \wedge \\ C \wedge \\ deduce\_subproofs(F, Sp) \end{array} \tag{9.29}$$

$$deduce\_subproofs(F, []) \tag{9.30}$$

$$deduce\_subproofs(F, [Proof|T]) \leftarrow \begin{array}{l} deduce(F, Proof) \wedge \\ deduce\_subproofs(F, T) \end{array} \tag{9.31}$$

How one defines each set of deductive rules depends on the problem to be solved but there is a basic set which gives a deductive behaviour typically described for pure logic programs and therefore particularly appropriate to the axiom sets used in HANSEL. It has four rules: the first saying that the atom *true* is always provable; the second proving a conjunction of goals if subproofs exist for each of the conjuncts; the third giving a proof of X if a clause exists in the axiom set which allows it to be deduced from Y and Y can be deduced as a sub-proof; and the fourth allowing built-in predicates to be considered true if they can be satisfied directly by the Prolog system.

$$deduction\_rule(basic, A \vdash true, true, []) \tag{9.32}$$

$$deduction\_rule(basic, A \vdash (X1 \wedge X2), true, [A \vdash X1, A \vdash X2]) \tag{9.33}$$

$$deduction\_rule(basic, A \vdash X, (X \leftarrow Y) \in A, [A \vdash Y]) \tag{9.34}$$

$$deduction\_rule(basic, A \vdash X, (system\_predicate(X) \wedge X), []) \tag{9.35}$$

Use of abductive inference rules is done similarly to that for deductive rules, but is a little more complex because abductive rules must generate axioms which could be added to the given axiom set to establish a given goal. Our abductive inference rules are thus of the form $abd\_rule(F, S \vdash X, E, C1, Sp, C2)$ where F is the type of abduction rule; $S \vdash X$ is the proof which the rule makes possible through generation of new axioms, E; $C1$ is a pre-condition which must hold for the rule to apply; and $C2$ is a post-condition which is used to construct E if the sub-proofs in Sp can be established. The mechanism used to apply abduction rules (similar to that for deductive rules) is then:

$$abduce(F, Proof, E) \leftarrow abd\_rule(F, Proof, E, C1, Sp, C2) \wedge C1 \wedge abduce\_subproofs(F, Sp) \wedge C2 \tag{9.36}$$

$$abduce\_subproofs(F, []) \tag{9.37}$$

$$abduce\_subproofs(F, [(Proof, E)|T]) \leftarrow abduce(F, Proof, E) \wedge abduce\_subproofs(F, T) \tag{9.38}$$

As for deduction, the choice of rules depends on the problem in hand but clauses 9.39 to 9.42 give a basic set. The first rule gives an empty abduction set for the atom *true*, representing a goal which is deducible without evidence. The second rule allows us to add $X \leftarrow true$ to axiom set S if we wish to prove unit goal X and there is no clause capable of deducing X in S. The third rule allows us to add the abduced axiom sets for goals $X1$ and $X2$ if we are trying to solve a goal which is the conjunction of $X1$ and $X2$. The fourth rule allows us to add the abduced axioms for Y if the goal we are looking for is X and Y would allow X to be concluded from the clauses in the axiom set, S.

$$abd\_rule(basic, A \vdash true, [], true, [], true) \tag{9.39}$$

$$abd\_rule(basic, S \vdash X, [(X \leftarrow true)], \begin{pmatrix} not(X = (C1, C2)) \wedge \\ not(X = true) \wedge \\ not((X \leftarrow P) \in S) \end{pmatrix}, [], true) \tag{9.40}$$

$$abd\_rule(basic, S \vdash (X1 \wedge X2), E, true, \begin{bmatrix} (S \vdash X1, E1), \\ (S \vdash X2, E2) \end{bmatrix}, union(E1, E2, E)) \tag{9.41}$$

$$abd\_rule(basic, S \vdash X, E, \begin{pmatrix} (X \leftarrow Y) \in S \wedge \\ not(Y = true) \end{pmatrix}, [(S \vdash Y, E)], true) \tag{9.42}$$

9.7 A Meta-Interpreter for Producing Tests

Section 9.3 introduced the use of properties associated with clauses and predicates in verifying that designs have been used correctly. Section 9.4 gives examples of properties associated with rewrite rules and Section 9.5 gives examples of properties for skeletons. We shall show in Chapters 11 and 12 how these are accumulated along with the specification. This section gives a straightforward way

of using them, thus instantiating the "Execution system" element of Figure 9.1 in Section 9.3.

One way of using properties associated with predicates and clauses is in testing. Testing of logic programs involves running the programs to determine whether or not they give the expected answers on sets of test goals. By including property information in testing, we have the opportunity to check not only whether the answers were right but also that they were obtained in ways which conform to the expected use of the predicates (as indicated by their properties). We can know what the appropriate instances of the properties are by replicating the proof procedure of the solver used to execute the logic program and ensuring that this collects all the properties relevant to that particular execution of the program. An easy way to do this for Horn clause programs is by using a meta-interpreter, such as the one defined in clauses 9.43 to 9.47 for the predicate $solve(G, \mathcal{T})$ below, where G is a given goal and $\mathcal{T}$ is a set of properties relevant to the testing of that goal which have been accumulated during its solution. This meta-interpreter is designed so that the solution of G is performed in a similar way to standard meta-interpreters for pure Prolog (see for example Chapter 19 of [Sterling and Shapiro, 1986]) and the appropriate tests, instantiated through the solution of goals in the proof tree, are accumulated in $\mathcal{T}$ without influencing the search during proof. This is similar to the standard way in which information such as explanations of proofs are accumulated in meta-interpreters (again, see Chapter 19 of [Sterling and Shapiro, 1986]). To make the formal definitions more compact we have used functional representations for set union in the definitions below, which are otherwise normal Prolog.

$$solve((A \wedge B), \mathcal{T}_a \cup \mathcal{T}_b) \leftarrow solve(A, \mathcal{T}_a) \wedge solve(B, \mathcal{T}_b) \quad (9.43)$$

$$solve((A \vee B), \mathcal{T}) \leftarrow solve(A, \mathcal{T}) \vee solve(B, \mathcal{T}) \quad (9.44)$$

$$solve(X, \mathcal{T}) \leftarrow hansel\_clause((X \leftarrow P), \mathcal{T}_x) \wedge solve(P, \mathcal{T}_p) \wedge \mathcal{T} = \mathcal{T}_x \cup \mathcal{T}_p \cup \{T | hansel\_prop(X, T)\} \quad (9.45)$$

$$solve(X, \mathcal{T}) \leftarrow not(hansel\_clause((X \leftarrow P), \mathcal{T}')) \wedge not(external(X)) \wedge clause(X, P) \wedge solve(P, \mathcal{T}_p) \wedge \mathcal{T} = \mathcal{T}_p \cup \{T | hansel\_prop(X, T)\} \quad (9.46)$$

$$solve(X, \{\}) \leftarrow external(X) \wedge X \quad (9.47)$$

This method of test collection via meta-interpretation is similar to that used for checking specifications against ontological constraints in [Kalfoglou and Robertson, 1999]. The main difference is that in [Kalfoglou and Robertson, 1999] the tests associated with subgoals in a proof are executed while the proof is being performed. This has the advantage of being parsimonious, since only those tests which reveal problems need be returned when the proof is complete. However, it has the disadvantage of requiring the tests to be decidable in sufficiently short computation times that the overhead on the computation of the main proof is not

excessive. This was practical for the ontological constraints which were the subject of [Kalfoglou and Robertson, 1999] but is impractical for many of the tests which are generated by HANSEL for example those which require that everything provable from one axiom set should be provable from another. Too much computation is involved for these to be performed during the execution of a specification. It is therefore a better policy to use the meta-interpreter simply to return all the tests and use a separate (possibly interactive) system to deal with proving or disproving them. This is a significant piece of research in its own right because the proofs may not be trivial and connecting them to the original design in a way which conveys meaning to designers is non-trivial. This topic is outside the scope of this thesis but is discussed briefly in Section 12.5.1.

We have now completed the introduction to the formal basis for HANSEL. In the next chapter we describe the way we have implemented it as a design tool.

Chapter 10

The HANSEL Structured Design Method

Figure 10.1 gives an overview of the interaction between the main windows in the current HANSEL system. The details of each window are given in the corresponding sections below. At the centre of the interaction is the main specification window (Section 10.1, which records the current state of the specification and from which all other windows are accessed. The first window accessed from the main window is normally the choice of initial templates (Section 10.2), which offers a menu of specifications expressed at the highest level of generality. This is normally followed by choice of refinements (Section 10.3) where a menu of refinements is offered for selected subgoals in the main specification window. At the lowest level of detail is the choice of skeletons (Section 10.4) which allows standard skeletal definitions to be selected for predicates appearing as subgoals in the main specification window. Some refinements introduce the need for subgoals which apply a test to a variable (Section 10.5). No window is shown in Figure 10.1 for adding an argument to a predicate (Section 10.6) because this happens without interaction upon pressing the "Add slice" button accompanying each defined predicate.

10.1 Specification Window

Figure 10.2 shows the main specification window in HANSEL. On the right is a window showing the current specification. On the left is a collection of controls, including the usual controls for quitting the system, clearing the specification, undoing each edit, saving the specification, plus the following controls which need more detailed explanation:

- The "New predicate" button is used to introduce a new predicate, with the name given in the edit box to its right. In Figure 10.2 the predicate

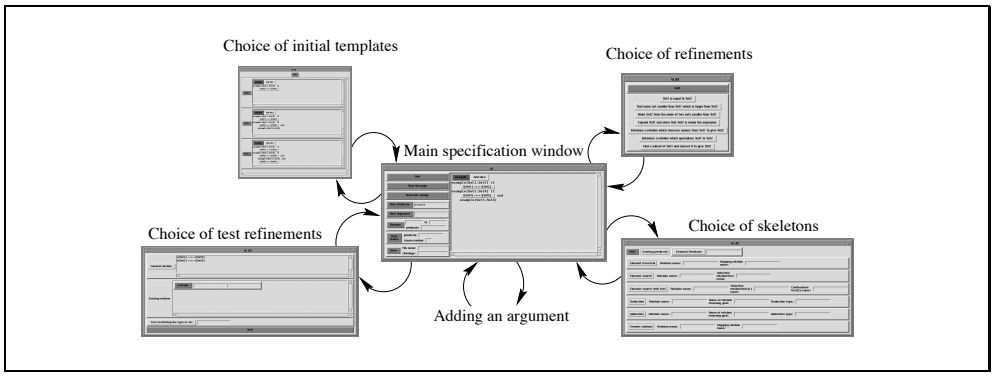


Figure 10.1: How the HANSEL windows interact

example has been added and has been specified at the highest level using an initial template (see Section 10.2).

- The "New argument" button is used to add an argument to a predicate appearing as a subgoal in the specification but which does not have a definition in the specification (*i.e* it is an externally defined predicate). This simply adds an additional argument to every occurrence of the named predicate along with a subgoal to test that variable (see Section 10.5).
- The "Rename" button renames the given variable name in the given predicate to some new name. The new name is not allowed to be a name currently used in the predicate, thus preventing variables being unified "covertly". Instead, they must be renamed "overtly" by using equality in the refinement choices (Section 10.3). This design choice is arguable - it is often convenient to use renaming to identify variables but unconstrained renaming can be used to change the semantics of the specification radically.
- The "Copy clause" button makes a duplicate of a chosen clause in the given predicate. This is most commonly used immediately after selecting an initial template, since the initial templates have only a single base and recursive clause, and some problems require several of these.

The specification displayed on the right side of Figure 10.2 corresponds to the formal expression:

$$\begin{aligned} example(Set1, Set2) &\leftarrow Set1 \approx Set2 \\ example(Set3, Set4) &\leftarrow Set3 \approx Set5 \wedge \\ &\quad\ example(Set5, Set4) \end{aligned}$$

In the display, the subgoals corresponding to $Set1 \approx Set2$ and $Set3 \approx Set5$ appear as buttons. This makes it easy to see where are the "gaps" in the current

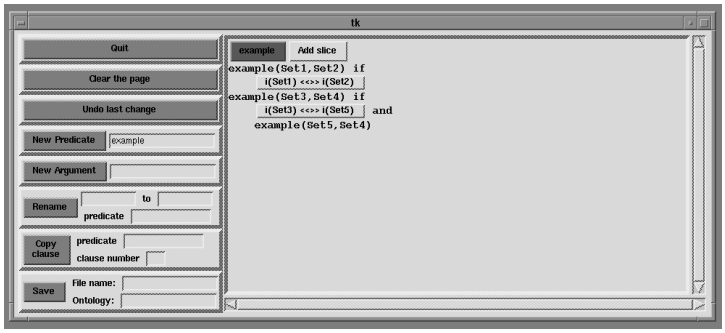


Figure 10.2: The main specification window

specification. Clicking on one of these buttons prompts for refinements for the appropriate subgoal: a menu like the one in Section 10.3 if the rewrite rules of Section 9.4 apply; a window like the one in Section 10.5 if the subgoal is a general test; or a window like the one in Section 10.4 if the subgoal stipulates a relation to be defined. Above each predicate appears a button named "Add slice" which, when clicked, adds an additional argument slice to that predicate (see Section 10.6). As the specification develops we accumulate properties associated with the rewrites and skeletons we select. Designers can choose whether or not they wish to see these properties displayed in the specification window. In the examples of this thesis we chose not to display them because they are discussed in the accompanying text.

10.2 Initial Templates

Figure 10.3 shows the window in which initial templates are chosen for a named predicate (in this case the name is *example*). The templates are shown as a scrolling menu, where clicking the "Select" button beside a template selects that template. The current version of HANSEL has only three general templates which can be adapted by duplicating clauses (see previous section) but this can be adapted to suit the domain. The definitions in each template are shown as they will appear in the specification.

10.3 Applying Refinements

Section 10.1 explained that clicking on a subgoal in the main specification window which is capable of refinement by the rewrites in Section 9.4 will present a menu of those refinements. An example, appears in Figure 10.4. This has been generated by matching the left-hand side of rewrite rules 11 to 17 of Section 9.4.3

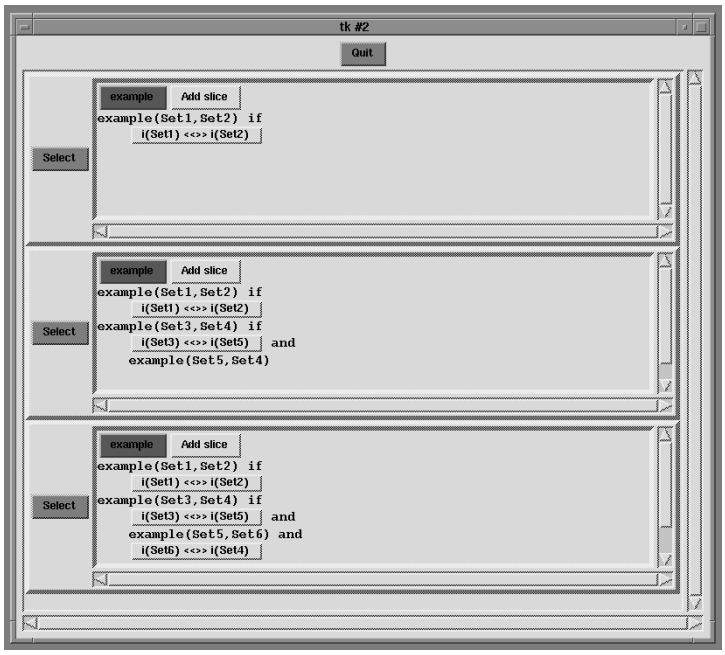


Figure 10.3: Choice of initial templates

and using a simple text template to give a less formal looking description of the rewrite. For instance, the menu item second from the top in Figure 10.4 is generated from rewrite 15 of Section 9.4.3, with the generated text describing the transitivity introduced by that refinement.

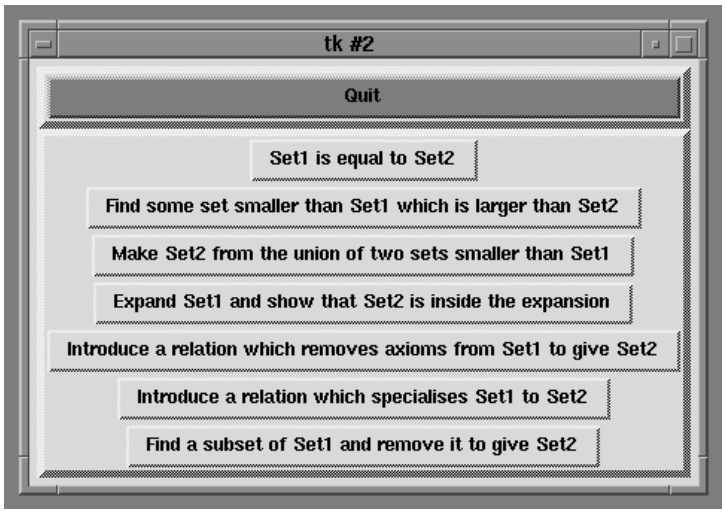


Figure 10.4: Refinements for $Set1 \supseteq Set2$

10.4 Introducing Skeletons

If the subgoal clicked in the main specification window (Section 10.1) is of the form $S1 \overset{R}{\rightsquigarrow} S2$, denoting that a predicate needs to be defined to relate $S1$ and $S2$, then a window offering choices of skeletons for predicate definition is shown, as in Figure 10.5. In the top row of buttons in this window are buttons allowing the predicate either to be identified with an existing predicate (a menu of these appears when that button is pressed) or stipulated as an external predicate (in which case the given name appears in the subgoal but no definition of it is added to the specification). Each of the other rows in the window describe a class of skeletons, parameterised through the edit fields in the corresponding row (see Section 9.5). There is a row for each of the skeleton groups identified in Section 9.5.1. The interface is flexible to allow for changes to the library of skeletons: if new groups of skeletons are added the window automatically produces the additional rows.

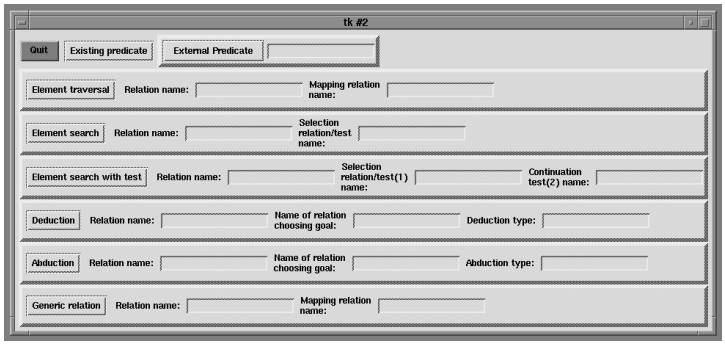


Figure 10.5: Skeleton choice window

10.5 Refining General Tests

If the subgoal clicked in the main specification window (Section 10.1) is of the form $\mathcal{T}(S)$, denoting that some test may be applied to axiom set S, then the window shown in Figure 10.6 appears. This gives three ways of refining the test: by relating the axiom set to some other set currently mentioned in the clause containing the test (top row of the window); by choosing a predicate already appearing in the specification (middle row) or by making the test be a unary predicate placing a type restriction on the set (lower row). In the example of Figure 10.6 the test was $\mathcal{T}(Set3)$ and it appeared in a clause containing sets $Set1$ and $Set2$ so the menu of possible relations to other sets contains $Set1 \approx Set3$ and $Set2 \approx Set3$. Only one predicate has been defined, *example* of arity 2, so this is given as an existing predicate with an edit field for each of the variable names which should appear in the rewrite for the test (these are checked to ensure that at least one of them is the set being tested and the others are sets mentioned in the clause).

10.6 Extending Skeletons

At the end of Section 10.1 we saw that each predicate in the main specification window has above it a button named "Add slice" which, when clicked, adds an additional argument slice to that predicate. This requires no additional window to be created because the additional argument and the subgoals required to connect it to the rest of the clause are added automatically. In each non-recursive clause the additional subgoal is a test on the argument. In each recursive clause the additional subgoal is a relation between the argument name in the head and the argument name in the recursive subgoal. The new argument is always added in the position before the last argument of the predicate, following the convention

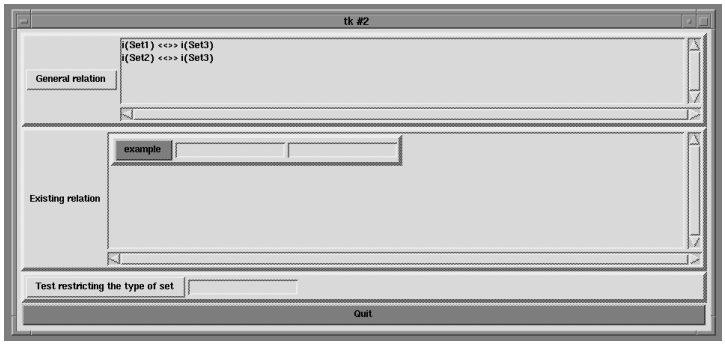


Figure 10.6: Test refinement window

that the last argument of a predicate is normally the output of the predicate so we shouldn't push it further to the left as we add slices. For example, if we add an argument slice to the predicate:

$$\begin{aligned} p(X, X) &\leftarrow q(X) \\ p(X, Z) &\leftarrow r(X, Y) \wedge p(Y, Z) \end{aligned}$$

then the resulting definition is:

$$\begin{aligned} p(X, A, X) &\leftarrow q(X) \wedge \mathcal{T}_1(A) \\ p(X, A_1, Z) &\leftarrow r(X, Y) \wedge p(Y, A_2, Z) \wedge A_1 \approx A_2 \end{aligned}$$

This is a simpler mechanism for extension than we needed in the techniques editor of LSS which had three different forms of extension. This is because the general relation $A_1 \approx A_2$ which is available in HANSEL but not in LSS allows a single form of extension which is general enough to subsume the three in LSS. One might argue that more specific forms of extension are beneficial because they make useful conceptual distinctions. In HANSEL we have chosen the most parsimonious solution in the knowledge that others are legitimate.

Chapter 11

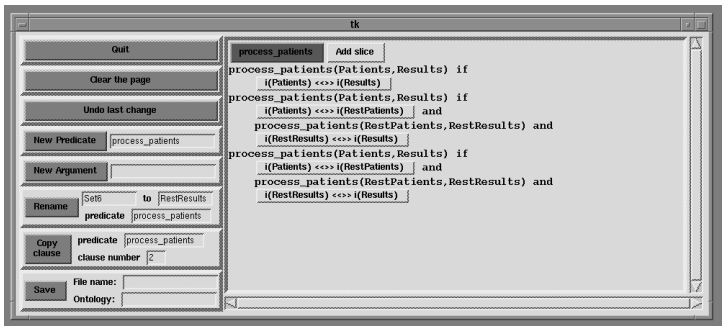
Worked Example

To illustrate HANSEL in use we develop in this chapter a version of the patient processing example which we used in Chapter 6 to demonstrate the LSS Process tool. This example is not ideally suited to HANSEL because it makes little use of the notion of axiom sets as arguments, which is what gives HANSEL its versatility, but it has the advantage of familiarity. The scenario begins by selecting a template describing the overall pattern of recursion for the predicate, as shown in Figure 11.1. The predicate takes two arguments: the theory describing the patients and the theory describing the results. In the base case (clause 11.1) the patient theory is related to that for the results. We also have two identical recursive clauses (11.2 and 11.3) which connect the theories in the heads of the clauses to new theories in the recursive subgoal. These will be the clauses which drive the processing of patients, with one case dealing with subsets of patients which can be diagnosed and the other dealing with patients which must be referred for tests.

Our next step (shown in Figure 11.1 is to replace the generic relations in the template with more specific relations. In the base case we choose simply to introduce an external predicate which is expected to define the relation. In the recursive cases we introduce inequalities between the theories corresponding to each axioms set. For *Patients* we stipulate that the theory taken down through the recursion (as *RestPatients*) is a specialisation of the original patient theory. By contrast the *Results* theory is a generalisation of the *RestResults* theory. Intuitively, the set of relevant information about patients gets smaller as we go down through the recursion, while the set of things we can conclude about the results gets larger as we go back up through the recursion.

We then refine the inequality between *Patients* and *RestPatients* in each recursive clause. Our choice is to select a subset of *Patients* and remove that from the original set to give *RestResults*. This introduces a new axiom set, *Subjects*, which describes the patients to be diagnosed or referred. This step is shown in Figure 11.3.

Our next step is to define the relation for choosing patients. This is done by selecting a skeletal definition for searching a set recursively, terminating when an

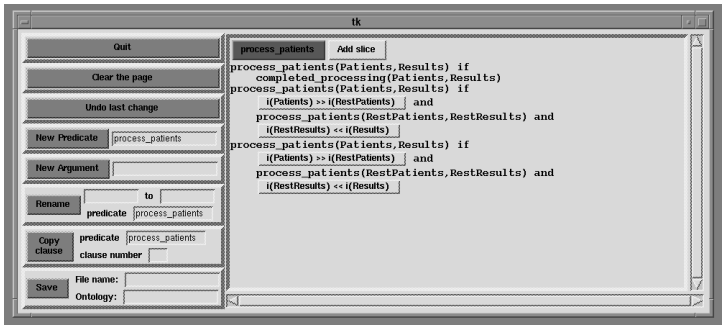


$$process\_patients(Patients, Results) \leftarrow \tau(Patients) \approx \tau(Results) \quad (11.1)$$

$$process\_patients(Patients, Results) \leftarrow \tau(Patients) \approx \tau(RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \approx \tau(Results) \quad (11.2)$$

$$process\_patients(Patients, Results) \leftarrow \tau(Patients) \approx \tau(RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \approx \tau(Results) \quad (11.3)$$

Figure 11.1: HANSEL first scenario - first step

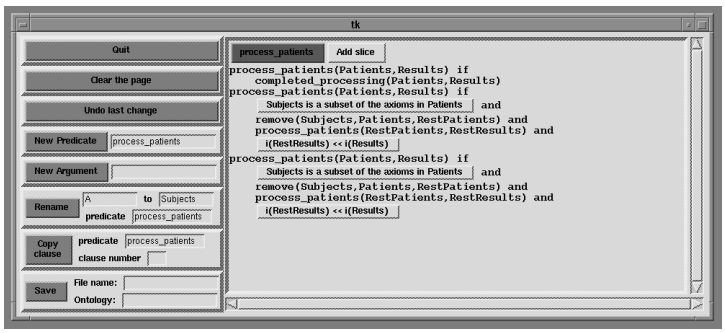


$$process\_patients(Patients, Results) \leftarrow completed\_processing(Patients, Results) \quad (11.4)$$

$$process\_patients(Patients, Results) \leftarrow \tau(Patients) \supseteq \tau(RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.5)$$

$$process\_patients(Patients, Results) \leftarrow \tau(Patients) \supseteq \tau(RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.6)$$

Figure 11.2: HANSEL first scenario - second step



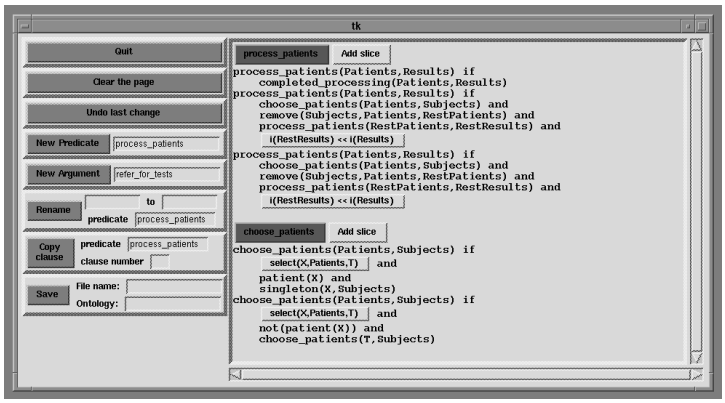
$$process\_patients(Patients, Results) \leftarrow completed\_processing(Patients, Results) \quad (11.7)$$

$$process\_patients(Patients, Results) \leftarrow Patients \overset{subset}{\rightsquigarrow} Subjects \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.8)$$

$$process\_patients(Patients, Results) \leftarrow Patients \overset{subset}{\rightsquigarrow} Subjects \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.9)$$

Figure 11.3: HANSEL first scenario - third step

axiom describing a patient is found (skeleton 15 in Appendix C). Thus the set of *Subjects* is a singleton set containing the axiom found during search through the *Patients* set. Figure 11.4 shows the state of the system after these refinements.



$$process\_patients(Patients, Results) \leftarrow completed\_processing(Patients, Results) \quad (11.10)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.11)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge \tau(RestResults) \subseteq \tau(Results) \quad (11.12)$$

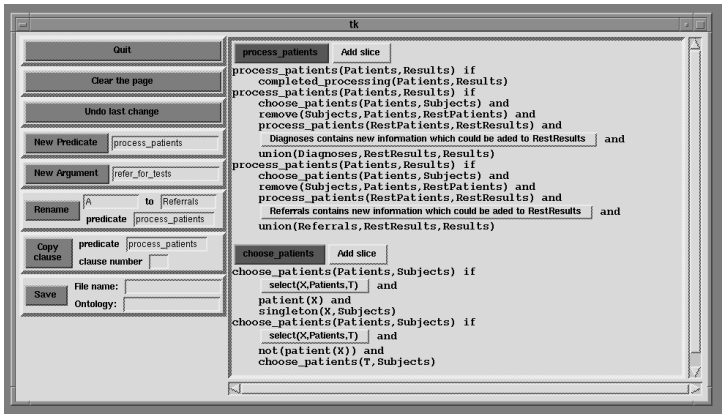
$$choose\_patients(Patients, Subjects) \leftarrow \mathcal{S}(X, Patients, T) \wedge patient(X) \wedge singleton(X, Subjects) \quad (11.13)$$

$$choose\_patients(Patients, Subjects) \leftarrow \mathcal{S}(X, Patients, T) \wedge not(patient(X)) \wedge choose\_patients(T, Subjects) \quad (11.14)$$

Figure 11.4: HANSEL first scenario - fourth step

We now turn to the inequality between *RestResults* and *Results*. This should make *Results* at least as large as *RestResults* so we stipulate a relation which generates new axioms consistent with *RestResults* and add these to it to obtain *Results*. This is shown in Figure 11.5.

Finally, we introduce externally defined predicates to perform the diagnosis and refer patients for tests. This gives us the complete definition shown in Figure 11.6.



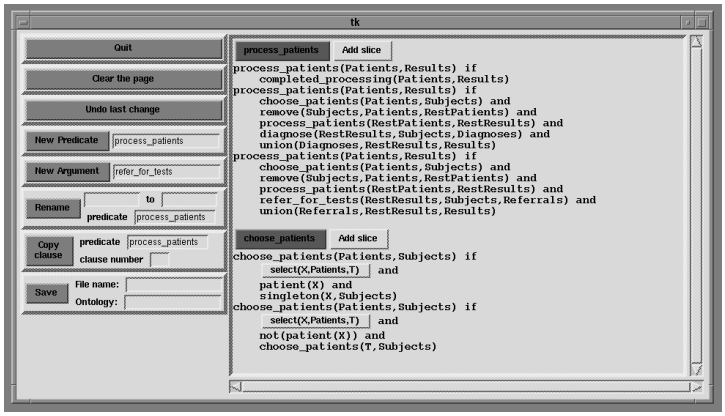
$$process\_patients(Patients, Results) \leftarrow completed\_processing(Patients, Results) \quad (11.15)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge RestResults \overset{new}{\rightsquigarrow} Diagnoses \wedge union(Diagnoses, RestResults, Results) \quad (11.16)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge RestResults \overset{new}{\rightsquigarrow} Referrals \wedge union(Referrals, RestResults, Results) \quad (11.17)$$

Plus *choose_patients*/2 as in Figure 11.4.

Figure 11.5: HANSEL first scenario - fifth step



$$process\_patients(Patients, Results) \leftarrow completed\_processing(Patients, Results) \quad (11.18)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge diagnose(RestResults, Subjects, Diagnoses) \wedge union(Diagnoses, RestResults, Results) \quad (11.19)$$

$$process\_patients(Patients, Results) \leftarrow choose\_patients(Patients, Subjects) \wedge remove(Subjects, Patients, RestPatients) \wedge process\_patients(RestPatients, RestResults) \wedge refer\_for\_tests(RestResults, Subjects, Referrals) \wedge union(Referrals, RestResults, Results) \quad (11.20)$$

Plus *choose_patients*/2 as in Figure 11.4.

Figure 11.6: HANSEL first scenario - sixth step

We are now at the same stage in our discussion of HANSEL as we were at the end of Chapter 6 when discussing LSS. Chapter 9 introduced the formal basis for the method and the current chapter has demonstrated how the tools within HANSEL interact during design. In the next chapter we assess the effectiveness of HANSEL on some larger problems and compare it to LSS.

Chapter 12

Evaluation

Unlike some of the work described in Chapter 7, the HANSEL tool was not built with the aim of being easy for a particular group of people to use. On the contrary, it was built to be as generic as possible and experience suggests that this makes it unlikely to be accessible to specification designers without training them in the general (and therefore abstract) theory on which it is based. Nevertheless, there are empirical issues which are germane to the current HANSEL tool. These are:

- To what extent can examples from established (possibly informal) design methods be described using HANSEL? The empirical issue here is whether the sort of design method which HANSEL supports is one which might actually occur. We evaluate this by tackling examples from the KADS design method in Sections 12.1 and 12.2.
- The degree to which HANSEL prescribes how to tackle a design problem. If we take a particular description of a problem, not expressed in a formal language, then it is clear that HANSEL can describe this problem in different ways but how different might these be in practice ? We give an example of this in Section 12.2 by taking the same KADS inference model and specifying it in two different ways in HANSEL.

Finally, there is the more general issue of the benefits obtained by having a formal lifecycle of the sort used in HANSEL compared to the distributed style of specification described in Chapters 3 to 7. To assess this, we return, in Section 12.4, to the problems raised in Section 7.3 when evaluating the LSS system.

12.1 An Example from KADS: Assessment

Figure 12.1 shows the diagrammatic inference structure for an assessment task. This takes as inputs a case description (describing the situation being assessed)

and a system model (describing the principles used in classifying situations in the chosen domain). The output of the assessment task is a description of the decision class to which the case has been assigned. To obtain this, the case description is selectively generalised to obtain appropriate features which are compared to norms selected from the system model.

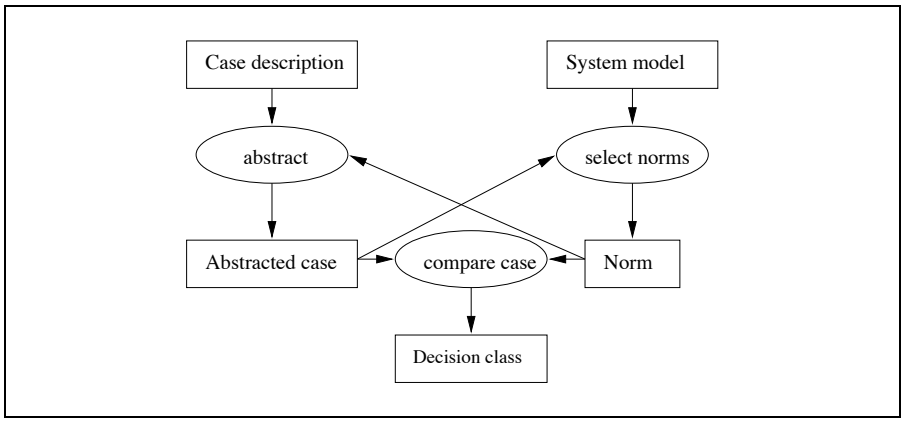


Figure 12.1: KADS inference structure diagram for an assessment task(adapted from [Tansley and Hayball, 1993] page 294).

12.1.1 Deriving an Assessment Model

We begin by selecting the simplest initial template provided in HANSEL, which states that assessment relates a case description to a decision class.

$$assessment(CaseDescription, DecisionClass) \leftarrow \tau(CaseDescription) \approx \tau(DecisionClass) \quad (12.1)$$

We then refine this by applying Rewrite 6 to say that the relationship between the case description and decision class is through an abstracted case.

$$assessment(CaseDescription, DecisionClass) \leftarrow \tau(CaseDescription) \approx \tau(AbstractedCase) \wedge \tau(AbstractedCase) \approx \tau(DecisionClass) \quad (12.2)$$

Now we apply Rewrite 3 to say that the abstracted case is a generalisation of the case description. This introduces the property given in expression 12.4, which requires anything provable from the case description to be provable from the abstracted case.

$$assessment(CaseDescription, DecisionClass) \leftarrow \tau(CaseDescription) \subseteq \tau(AbstractedCase) \wedge \tau(AbstractedCase) \approx \tau(DecisionClass) \quad (12.3)$$

$$\forall X.CaseDescription \vdash X \rightarrow AbstractedCase \vdash X \quad (12.4)$$

The relation between abstracted case and decision class cannot be classified as a specialisation or as a generalisation so we simply introduce a new predicate to describe the appropriate relation.

$$\begin{array}{l} assessment(CaseDescription, DecisionClass) \leftarrow \\ \quad \tau(CaseDescription) \subseteq \tau(AbstractedCase) \wedge \\ \quad compare\_case(AbstractedCase, DecisionClass) \end{array} \tag{12.5}$$

We now add the missing knowledge sources from the KADS diagram of Figure 12.1. These are the system model, used in selecting the norm from the abstracted case, and the norm itself. The system model is an input to the assessment predicate so it is added as an extra argument slice to the predicate (with $\mathcal{T}_2(SystemModel)$ added automatically to supply the "hook" for connecting the system model to other arguments in the predicate if necessary). The norm is used in comparing the case so it is added as an additional argument to *compare_case* (with $\mathcal{T}_1(Norm)$ added automatically to enable it to be related to other axiom sets).

$$\begin{array}{l} assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ \quad \tau(CaseDescription) \subseteq \tau(AbstractedCase) \wedge \\ \quad \mathcal{T}_1(Norm) \wedge \\ \quad compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ \quad \mathcal{T}_2(SystemModel) \end{array} \tag{12.6}$$

The tests we just introduced are now refined. The first is rewritten to a relation between the system model and the norm. The second simply checks that the system model is of the appropriate type.

$$\begin{array}{l} assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ \quad \tau(CaseDescription) \subseteq \tau(AbstractedCase) \wedge \\ \quad \tau(SystemModel) \approx \tau(Norm) \wedge \\ \quad compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ \quad system\_model(SystemModel) \end{array} \tag{12.7}$$

We intend the norm to be a specialisation of the system model (since our idea of a norm is that it describes a set of features which can be deduced from the system model) so we apply Rewrite 1 to introduce the appropriate inequality. This introduces the property given in expression 12.9, which requires anything provable from the norm to be provable from the system model.

$$\begin{array}{l} assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ \quad \tau(CaseDescription) \subseteq \tau(AbstractedCase) \wedge \\ \quad \tau(SystemModel) \supseteq \tau(Norm) \wedge \\ \quad compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ \quad system\_model(SystemModel) \end{array} \tag{12.8}$$

$$\forall X. Norm \vdash X \rightarrow SystemModel \vdash X \tag{12.9}$$

Notice that this property is specific to clause 12.8 and HANSEL actually stores the clause and its properties together in a single predicate, $hansel\_clause(C, \mathcal{T})$, where C is a clause in the specification and $\mathcal{T}$ is a set of properties required to hold for any instance of C used to solve a goal) as described in Section 9.3. Packaging the clauses together in this way makes it a little more clumsy to read them

so we keep them separate in the examples of this chapter but the correspondence between variables is important when we generate instances of properties for testing in Section 12.3. Thus, the variables $Norm$ and $SystemModel$ in clause 12.8 and property 12.9 (above) form a connection between these two expressions during testing in Section 12.3. The same principle holds for other clause properties in this chapter.

We now commit to introducing a relation for generalising from the case description to the abstracted case and a relation for specialising the system model to obtain the norm. This is done by applying rewrites 19 and 12. As a consequence, we introduce properties 12.11 and 12.12 which have the effect of tightening the constraints already given by properties 12.4 and 12.9.

$$\begin{array}{l} assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ \quad CaseDescription \overset{generalise}{\rightsquigarrow} AbstractedCase \wedge \\ \quad SystemModel \overset{specialise}{\rightsquigarrow} Norm \wedge \\ \quad compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ \quad system\_model(SystemModel) \end{array} \tag{12.10}$$

$$\exists X. SystemModel \vdash X \ \wedge \ not(Norm \vdash X) \tag{12.11}$$

$$\exists X. AbstractedCase \vdash X \ \wedge \ not(CaseDescription \vdash X) \tag{12.12}$$

Our next step is to introduce skeletal definitions for the generalisation and specialisation relations of the previous step. These correspond to the *abstract* and *select_norms* ovals in the diagram of Figure 12.1. We intend that the *abstract* predicate should generalise from a case description to an abstracted case by applying all the generalisations possible to each axiom in the case description. Therefore we choose Skeleton 7 for this relation. The *select_norms* predicate is intended to filter the system model for parts of it establishing the norm, so we use Skeleton 9 for it. In selecting these skeletons we also introduce properties 12.19 to 12.22.

$$\begin{array}{l} assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ \quad abstract(CaseDescription, AbstractedCase) \wedge \\ \quad select\_norms(SystemModel, Norm) \wedge \\ \quad compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ \quad system\_model(SystemModel) \end{array} \tag{12.13}$$

$$\begin{array}{l} abstract(CaseDescription, AbstractedCase) \leftarrow \\ \quad empty(CaseDescription) \wedge \\ \quad empty(AbstractedCase) \end{array} \tag{12.14}$$

$$\begin{array}{l} abstract(CaseDescription, AbstractedCase) \leftarrow \\ \quad \mathcal{S}(X, CaseDescription, T) \wedge \\ \quad setof(X1, generalisation(X, X1), Set) \wedge \\ \quad abstract(T, T1) \wedge \\ \quad \mathcal{A}(X, Set, Set1) \wedge \\ \quad union(Set1, T1, AbstractedCase) \end{array} \tag{12.15}$$

$$\begin{array}{l} select\_norms(SystemModel, Norm) \leftarrow \\ \quad empty(SystemModel) \wedge \\ \quad empty(Norm) \end{array} \tag{12.16}$$

$$\begin{array}{l} select\_norms(SystemModel, Norm) \leftarrow \\ \quad \mathcal{S}_1(X, SystemModel, T) \wedge \\ \quad relevant(X, X1) \wedge \\ \quad select\_norms(T, T1) \wedge \\ \quad \mathcal{A}(X1, T1, Norm) \end{array} \tag{12.17}$$

$$\begin{array}{l} select\_norms(SystemModel, Norm) \leftarrow \\ \quad \mathcal{S}_2(X, SystemModel, T) \wedge \\ \quad not(relevant(X, X1)) \wedge \\ \quad select\_norms(T, Norm) \end{array} \tag{12.18}$$

$$abstract(CaseDescription, AbstractedCase) \rightarrow \forall X1.X1 \in CaseDescription \rightarrow \left(\begin{array}{c} \exists X2. \left(\begin{array}{c} generalisation(X1, X2) \wedge \\ X1 \in AbstractedCase \wedge \\ X2 \in AbstractedCase \end{array} \right) \\ \wedge \\ \forall X2'.generalisation(X1, X2') \rightarrow X2' \in AbstractedCase \end{array} \right) \tag{12.19}$$

$$abstract(CaseDescription, AbstractedCase) \rightarrow \forall X2.X2 \in AbstractedCase \rightarrow \left(\begin{array}{c} \exists X1. \left(\begin{array}{c} generalisation(X1, X2) \wedge \\ X1 \in CaseDescription \wedge \\ X1 \in AbstractedCase \end{array} \right) \\ \vee \\ \left(\begin{array}{c} not(\exists X1.generalisation(X1, X2)) \wedge \\ X2 \in CaseDescription \end{array} \right) \end{array} \right) \tag{12.20}$$

$$select\_norms(SystemModel, Norm) \rightarrow \forall X1.X1 \in SystemModel \rightarrow \left(\begin{array}{c} \exists X2.relevant(X1, X2) \ \wedge \ X2 \in Norm \\ \vee \\ not(\exists X2.relevant(X1, X2)) \end{array} \right) \tag{12.21}$$

$$select\_norms(SystemModel, Norm) \rightarrow \\ \forall X2.X2 \in Norm \rightarrow (\ \exists X1.relevant(X1, X2)\ \wedge\ X1 \in SystemModel\ ) \quad (12.22)$$

The skeletal definitions introduced in the previous step require elements to be selected from and added to sets but do not commit to the means of selection or addition. We now stipulate that the first elements of appropriate sets should be selected or added in each case.

$$assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow \\ abstract(CaseDescription, AbstractedCase) \wedge \\ select\_norms(SystemModel, Norm) \wedge \\ compare\_case(AbstractedCase, Norm, DecisionClass) \wedge \\ system\_model(SystemModel) \quad (12.23)$$

$$abstract(CaseDescription, AbstractedCase) \leftarrow \\ empty(CaseDescription) \wedge \\ empty(AbstractedCase) \quad (12.24)$$

$$abstract(CaseDescription, AbstractedCase) \leftarrow \\ select\_first\_element(X, CaseDescription, T) \wedge \\ setof(X1, generalisation(X, X1), Set) \wedge \\ abstract(T, T1) \wedge \\ add\_element\_first(X, Set, Set1) \wedge \\ union(Set1, T1, AbstractedCase) \quad (12.25)$$

$$select\_norms(SystemModel, Norm) \leftarrow \\ empty(SystemModel) \wedge \\ empty(Norm) \quad (12.26)$$

$$select\_norms(SystemModel, Norm) \leftarrow \\ select\_first\_element(X, SystemModel, T) \wedge \\ relevant(X, X1) \wedge \\ select\_norms(T, T1) \wedge \\ add\_element\_first(X1, T1, Norm) \quad (12.27)$$

$$select\_norms(SystemModel, Norm) \leftarrow \\ select\_first\_element(X, SystemModel, T) \wedge \\ not(relevant(X, X1)) \wedge \\ select\_norms(T, Norm) \quad (12.28)$$

Finally we account for the remaining connection in the diagram in Figure 12.1 which requires *select_norms* to take the abstracted case into account. We do this by adding an additional argument slice to *select_norms* (with this automatically adding to *assessment* an additional variable in the *select_norms* subgoal) and carrying this through as an additional argument to the *relevant* relation which determines how *select_norms* filters the axioms.

$$assessment(CaseDescription, SystemModel, DecisionClass) \leftarrow abstract(CaseDescription, AbstractedCase) \wedge select\_norms(SystemModel, AbstractedCase, Norm) \wedge compare\_case(AbstractedCase, Norm, DecisionClass) \wedge system\_model(SystemModel) \quad (12.29)$$

$$abstract(CaseDescription, AbstractedCase) \leftarrow empty(CaseDescription) \wedge empty(AbstractedCase) \quad (12.30)$$

$$abstract(CaseDescription, AbstractedCase) \leftarrow select\_first\_element(X, CaseDescription, T) \wedge setof(X1, generalisation(X, X1), Set) \wedge abstract(T, T1) \wedge add\_element\_first(X, Set, Set1) \wedge union(Set1, T1, AbstractedCase) \quad (12.31)$$

$$select\_norms(SystemModel, AbstractedCase, Norm) \leftarrow empty(SystemModel) \wedge empty(Norm) \quad (12.32)$$

$$select\_norms(SystemModel, AbstractedCase, Norm) \leftarrow select\_first\_element(X, SystemModel, T) \wedge relevant(X, AbstractedCase, X1) \wedge select\_norms(T, AbstractedCase, T1) \wedge add\_element\_first(X1, T1, Norm) \quad (12.33)$$

$$select\_norms(SystemModel, AbstractedCase, Norm) \leftarrow select\_first\_element(X, SystemModel, T) \wedge not(relevant(X, AbstractedCase, X1)) \wedge select\_norms(T, AbstractedCase, Norm) \quad (12.34)$$

12.1.2 Running the Assessment Model

Our instance of the assessment model is now complete, in the sense that the predicates corresponding to elements of the diagram in Figure 12.1 are defined. There are, however, a number of predicates which HANSEL left undefined. These refer to parts of the design where we make commitments to the contents of axiom sets and decide on the data structures used to represent them. We choose the standard Prolog list data structures for set representation and give below the definitions of each of the missing predicates.

An empty set is represented by the constant '[]'.

$$empty([]) \quad (12.35)$$

The first element of a set corresponds to the head of a list.

$$select\_first\_element(X, [X|T], T) \quad (12.36)$$

Elements are added to the head of a list.

$$add\_element\_first(X, T, [X|T]) \quad (12.37)$$

An axiom, $H \leftarrow B$ is relevant to a given *Case* if there is an axiom in the case which unifies with H (roughly speaking, the case talks about H).

$$relevant((H \leftarrow B), Case, (H \leftarrow B)) \leftarrow (H \leftarrow B1) \in Case. \tag{12.38}$$

We have the following library of domain specific generalisations from an axiom in the case description (first argument) to an additional axiom in the abstracted case (second argument). The first three of these give axioms which we can add if the case gives the age of a person, allowing the person to be classified as young, middle aged or old. The next two allow a person who leads a research group to be classified as having experience and management skills.

$$generalisation\left(\begin{array}{l}(age(X,N) \leftarrow true),\\ (young(X) \leftarrow age(X,N1), N1 < 30)\end{array}\right) \tag{12.39}$$

$$generalisation\left(\begin{array}{l}(age(X,N) \leftarrow true),\\ (middle\_aged(X) \leftarrow age(X,N1), N1 \geq 30, N1 < 50)\end{array}\right) \tag{12.40}$$

$$generalisation\left(\begin{array}{l}(age(X,N) \leftarrow true),\\ (old(X) \leftarrow age(X,N1), N1 \geq 50)\end{array}\right) \tag{12.41}$$

$$generalisation\left(\begin{array}{l}(leads\_group(X) \leftarrow true),\\ (experienced(X) \leftarrow leads\_group(X))\end{array}\right) \tag{12.42}$$

$$generalisation\left(\begin{array}{l}(leads\_group(X) \leftarrow true),\\ (management\_skills(X) \leftarrow leads\_group(X))\end{array}\right) \tag{12.43}$$

Finally, we define the way in which a norm is compared to an abstracted case, giving us axioms describing a decision class. This is done by having the decision class contain each axiom in the norm which can be deduced from the abstracted case.

$$compare\_case(AbstractedCase, [(H \leftarrow P)|T], [(H \leftarrow P)|R]) \leftarrow deduce(basic, AbstractedCase \vdash H) \wedge compare\_case(AbstractedCase, T, R) \tag{12.44}$$

$$compare\_case(AbstractedCase, [(H \leftarrow B)|T], R) \leftarrow not(deduce(basic, AbstractedCase \vdash H)), compare\_case(AbstractedCase, T, R) \tag{12.45}$$

$$compare\_case(AbstractedCase, [], []) \tag{12.46}$$

We can now run our logic program by calling an appropriate *assessment* goal. For example, we might have a case where *dave* is aged 36 and leads a research group, and a system model which requires us to look for middle aged people with management skills. Our goal is:

$$assessment(\left[\begin{array}{c}(age(dave,36) \leftarrow true),\\ (leads\_group(dave) \leftarrow true)\end{array}\right], \left[\begin{array}{c}(middle\_aged(X) \leftarrow true),\\ (management\_skills(X) \leftarrow true),\end{array}\right], D)$$

and this gives the decision classification confirming that *dave* does indeed have all the skills stipulated in the system model:

$$D = [(middle\_aged(dave) \leftarrow true), (management\_skills(dave) \leftarrow true)]$$

12.2 An Example from KADS: Qualitative Prediction

The aim of this section is to demonstrate how different HANSEL specifications can be designed from the same (informal) starting point. Figure 12.2 shows the diagrammatic inference structure for a qualitative prediction task. The basic idea of this task is to allow envisionments of the qualitative behaviour of a system, given an initial state description and envisioning the predicted behaviour at some later time. This involves the instantiation and computation of influence relations to augment the state (top of the diagram) and the derivation of new landmark values and influences to produce the predicted state (lower part of the diagram).

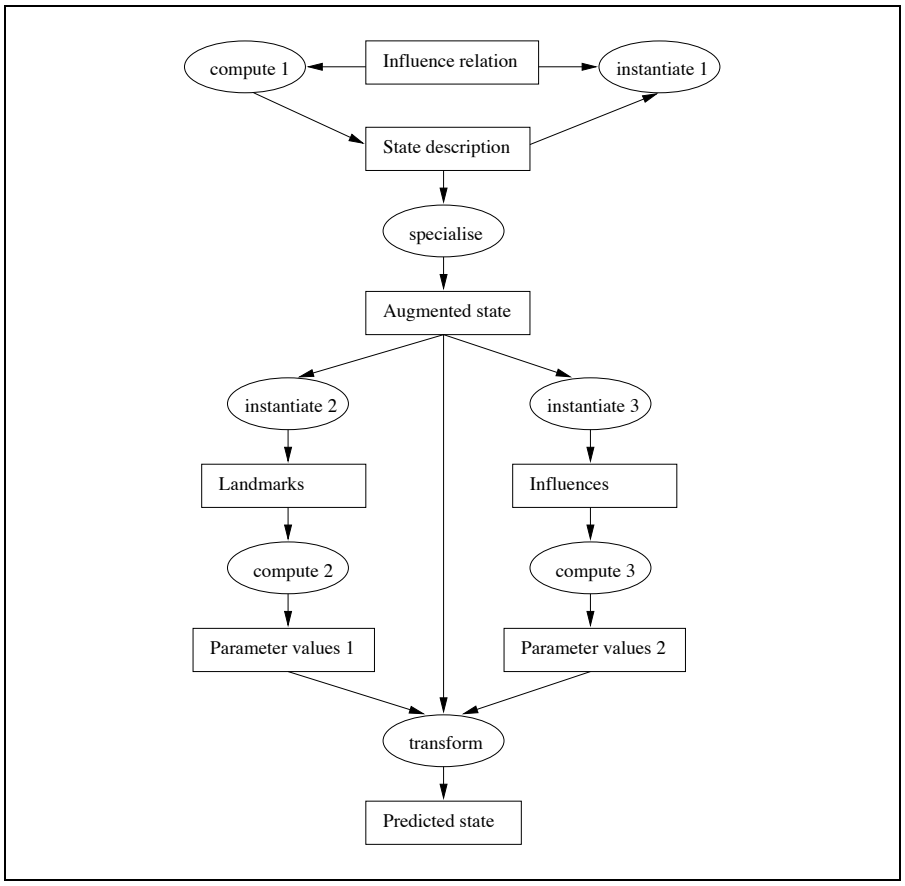


Figure 12.2: KADS inference structure diagram for a qualitative prediction task(adapted from [Tansley and Hayball, 1993] page 318).

A classic qualitative simulation example is a spring anchored at one end which is modelled qualitatively in terms of the position of its free end; the directional velocity of its free end; and the directional acceleration of its free end. The landmark values for each of these parameters are:

Position : Positive when fully extended. Zero when at rest. Negative when fully compressed.

Velocity : Positive when travelling in the direction of extension. Zero when not moving. Negative when travelling in the direction of compression.

Acceleration : Positive when accelerating in the direction of extension. Zero when not accelerating. Negative when accelerating in the direction of compression.

The continuous change of position, velocity and acceleration as the spring oscillates is shown by the cycles in the diagram on the left of Figure 12.3[1]. The numbered stages in this diagram indicate the approximate points at which a significant qualitative change has occurred. We map these to the transitions between qualitative states shown on the right. For example, at state number 5 the spring's position is negative, its velocity is zero and its acceleration is positive. The examples of Section 12.2.1 and Section 12.2.2 use HANSEL to construct a two different models capable of generating this sort of behaviour: the first taking a simpler route than the one suggested in the KADS diagram of Figure 12.2 and the second conforming more closely to the KADS diagram.

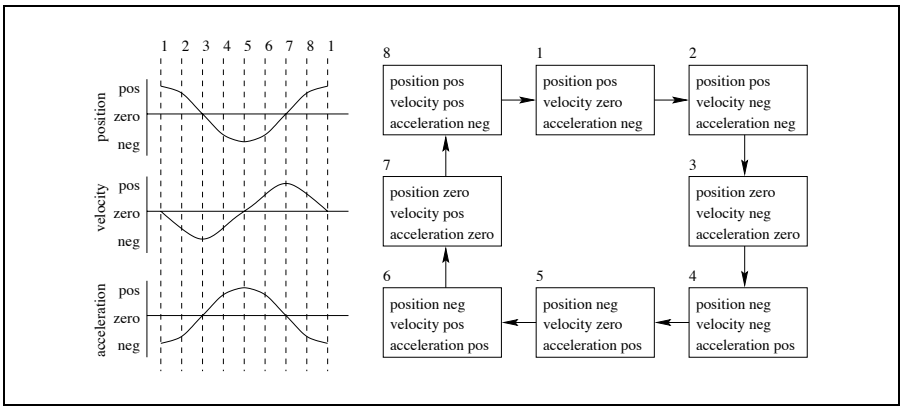


Figure 12.3: Example behaviour of an oscillating spring

[1]More sophisticated simulations are possible, for example modelling of damped oscillations through the invention of new landmark values, but we do not consider these here.

12.2.1 Deriving the First Qualitative Prediction Model

We begin by choosing a basic recursive template, in which the base case relates the current state to the predicted state and the recursive state relates the current state to the next state and from there to the predicted state.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.47)$$

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.48)$$

We then rewrite our general relations to inequalities, with the predicted state being included in the current state in the base case (rewrite 3) and the next state including the current state in the recursive case (rewrite 1). This introduces properties 12.51 and 12.52.

$$qp(State, Predicted) \leftarrow \tau(State) \supseteq \tau(Predicted) \quad (12.49)$$

$$qp(State, Predicted) \leftarrow \tau(State) \subseteq \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.50)$$

$$\forall X. State \vdash X \rightarrow NextState \vdash X \quad (12.51)$$

$$\forall X. Predicted \vdash X \rightarrow State \vdash X \quad (12.52)$$

Next we commit to relations for specialising the current state in the base case (rewrite 12) and generalising the current state in the recursive case (rewrite 19). These introduce properties 12.55 and 12.56.

$$qp(State, Predicted) \leftarrow State \overset{specialise}{\rightsquigarrow} Predicted \quad (12.53)$$

$$qp(State, Predicted) \leftarrow \tau(State) \overset{generalise}{\rightsquigarrow} \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.54)$$

$$\exists X. State \vdash X \ \wedge \ not(Predicted \vdash X) \quad (12.55)$$

$$\exists X. NextState \vdash X \ \wedge \ not(State \vdash X) \quad (12.56)$$

We now define the generalisation of the state in the recursive case using skeleton 25, which introduces the predicate *attribute_goal* to determine the goal used to derive the new landmark for an attribute of the system and applies an appropriate form of deduction (*basic*) exhaustively to obtain all the new attribute landmarks. The skeleton introduces properties 12.60 to 12.62.

$$qp(State, Predicted) \leftarrow State \overset{specialise}{\rightsquigarrow} Predicted \quad (12.57)$$

$$qp(State, Predicted) \leftarrow envision(State, NextState) \wedge qp(NextState, Predicted) \quad (12.58)$$

$$envision(State, NextState) \leftarrow attribute\_goal(State, Goal) \wedge setof(Goal, deduce(basic, proof(State, Goal)), Set) \wedge union(Set, State, NextState) \quad (12.59)$$

$$envision(State, NextState) \rightarrow \forall X1.X1 \in State \rightarrow X1 \in NextState \quad (12.60)$$

$$envision(State, NextState) \rightarrow \exists G.attribute\_goal(State, G) \wedge \forall X.((X = G \wedge deduce(D, State \vdash G)) \rightarrow X \in NextState) \quad (12.61)$$

$$envision(State, NextState) \rightarrow \forall G.G \in NextState \rightarrow (G \in State \vee (attribute\_goal(State, G) \wedge deduce(D, State \vdash G))) \quad (12.62)$$

Finally, we define the specialisation of the current state to the predicted state in the base case using skeleton 9, which allows only attributes of the state to be passed through to the final state (the test determining this mapping being the *is_attribute* subgoal). The skeleton introduces properties 12.69 and 12.70.

$$qp(State, Predicted) \leftarrow final\_attributes(State, Predicted) \quad (12.63)$$

$$qp(State, Predicted) \leftarrow envision(State, (NextState) \wedge qp(NextState, Predicted) \quad (12.64)$$

$$envision(State, NextState) \leftarrow attribute\_goal(State, Goal) \wedge setof(Goal, deduce(basic, proof(State, Goal)), Set) \wedge union(Set, State, NextState) \quad (12.65)$$

$$final\_attributes(State, Predicted) \leftarrow empty(State) \wedge empty(Predicted) \quad (12.66)$$

$$final\_attributes(State, Predicted) \leftarrow select\_first\_element(X, State, T) \wedge is\_attribute(X, X1) \wedge final\_attributes(T, T1) \wedge add\_element\_first(X1, T1, Predicted) \quad (12.67)$$

$$final\_attributes(State, Predicted) \leftarrow select\_first\_element(X, State, T) \wedge not(is\_attribute(X, X1)) \wedge final\_attributes(T, Predicted) \quad (12.68)$$

$$final\_attributes(State, Predicted) \rightarrow \\ \forall X1.X1 \in State \rightarrow \left(\begin{array}{c} \exists X2.is\_attribute(X1, X2) \ \wedge \ X2 \in Predicted \\ \vee \\ not(\exists X2.is\_attribute(X1, X2)) \end{array} \right) \quad (12.69)$$

$$final\_attributes(State, Predicted) \rightarrow \\ \forall X2.X2 \in Predicted \rightarrow (\ \exists X1.is\_attribute(X1, X2) \ \wedge \ X1 \in State \ ) \quad (12.70)$$

12.2.2 Running the First Qualitative Prediction Model

To run the qualitative model we must supply the definitions for the predicates not built directly in HANSEL. The definitions of *empty*, *select_first_element* and *add_element_first* are identical to those in Section 12.1.2 (clauses 12.1.2, 12.1.2 and 12.1.2). To design the other predicates we must decide how axioms describing attributes will be represented and how the deduction mechanism for deriving new states will work. The first of these tasks is straightforward. We use the expression $att(A, V, S)$ to denote that attribute A has value V in state S, where S is either *init* (denoting the initial state) or $s(S')$ denoting the successor state of state S'. The predicate used to select appropriate attribute expressions is then:

$$is\_attribute((att(A, V, S) \leftarrow true), (att(A, V, S) \leftarrow true)) \quad (12.71)$$

and the predicate used to choose an appropriate goal for deriving an attribute's landmark value in the next state is:

$$attribute\_goal(State, att(A, V, s(S))) \leftarrow last\_state\_step(State, S) \quad (12.72)$$

where $last\_state\_step(State, S)$ is true if state S is the latest state currently appearing in the axiom set, defined below as the state expression S for which there exists no attribute with a successor state ($s(S)$).

$$last\_state\_step(State, S) \leftarrow \\ (att(A, V, S) \leftarrow true) \in State \wedge not((att(A1, V1, s(S)) \leftarrow true) \in State) \quad (12.73)$$

Finally, we must define the proof rules for deducing new landmark values for attributes. These are the basic deduction rules of Section 9.6 (clauses 9.32 to 9.35).

We can now generate qualitative behaviours corresponding to the example in Figure 12.3 by finding solutions to the goal given in expression 12.74 below. The axiom set in the first argument of the *qp* goal contains four kinds of information. The first three clauses define the initial state of the spring (corresponding to the state on the left in Figure 12.3). The next three clauses define transition rules which derive values for attributes in successor states from attribute values in the

current state - for example, the first of these says that the position of the end of the spring in successor state, $s(S)$, will be determined by a transition, *trans*, on its current position and velocity. These transition definitions are given in the next nine clauses - for instance, the first of these says that if a value (*e.g.* position) is positive and its derivative (*e.g.* velocity) is positive then the value stays positive. Finally come the clauses defining the qualitative opposing values between position and acceleration at the next landmark.

$$
\begin{array}{ll}
qp([& (att(position, pos, init) \leftarrow true), \\
& (att(velocity, zero, init) \leftarrow true), \\
& (att(acceleration, neg, init) \leftarrow true), \\
& (att(position, P, s(S)) \leftarrow \quad att(position, Pp, S) \wedge \\
& \qquad\qquad\qquad\qquad\quad att(velocity, Vp, S) \wedge \\
& \qquad\qquad\qquad\qquad\quad trans(Pp, Vp, P)), \\
& (att(velocity, V, s(S)) \leftarrow \quad att(velocity, Vp, S) \wedge \\
& \qquad\qquad\qquad\qquad\quad att(acceleration, Ap, S) \wedge \\
& \qquad\qquad\qquad\qquad\quad trans(Vp, Ap, V)), \\
& (att(acceleration, A, s(S)) \leftarrow \quad att(position, Vp, S) \wedge \\
& \qquad\qquad\qquad\qquad\quad qneg(Vp, A)), \\
& (trans(pos, pos, pos) \leftarrow true), \\
& (trans(pos, zero, pos) \leftarrow true), \\
& (trans(pos, neg, zero) \leftarrow true), \\
& (trans(zero, pos, pos) \leftarrow true), \\
& (trans(zero, zero, zero) \leftarrow true), \\
& (trans(zero, neg, neg) \leftarrow true), \\
& (trans(neg, pos, zero) \leftarrow true), \\
& (trans(neg, zero, neg) \leftarrow true), \\
& (trans(neg, neg, neg) \leftarrow true), \\
& (qneg(pos, neg) \leftarrow true), \\
& (qneg(pos, zero) \leftarrow true), \\
& (qneg(zero, pos) \leftarrow true), \\
& (qneg(zero, neg) \leftarrow true), \\
& (qneg(neg, zero) \leftarrow true), \\
& (qneg(neg, pos) \leftarrow true)], X)
\end{array}
\tag{12.74}
$$

The goal above will generate solutions for the predicted state (X), starting with the immediate successor state and enlarging to each future state on backtracking. The elements corresponding to a cycle through the diagram in Figure 12.3 are:

$$
\begin{array}{l}
[\ (att(position, pos, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(velocity, pos, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(acceleration, neg, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(position, zero, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(velocity, pos, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(acceleration, zero, s(s(s(s(s(s(s(init)))))))) \leftarrow true),\\
(att(position, neg, s(s(s(s(s(s(init))))))) \leftarrow true),\\
(att(velocity, pos, s(s(s(s(s(s(init))))))) \leftarrow true),\\
(att(acceleration, pos, s(s(s(s(s(s(init))))))) \leftarrow true),\\
(att(position, neg, s(s(s(s(s(init)))))) \leftarrow true),\\
(att(velocity, zero, s(s(s(s(s(init)))))) \leftarrow true),\\
(att(acceleration, pos, s(s(s(s(s(init)))))) \leftarrow true),\\
(att(position, neg, s(s(s(s(init))))) \leftarrow true),\\
(att(velocity, neg, s(s(s(s(init))))) \leftarrow true),\\
(att(acceleration, pos, s(s(s(s(init))))) \leftarrow true),\\
(att(position, zero, s(s(s(init)))) \leftarrow true),\\
(att(velocity, neg, s(s(s(init)))) \leftarrow true),\\
(att(acceleration, zero, s(s(s(init)))) \leftarrow true),\\
(att(position, pos, s(s(init))) \leftarrow true),\\
(att(velocity, neg, s(s(init))) \leftarrow true),\\
(att(acceleration, neg, s(s(init))) \leftarrow true),\\
(att(position, pos, s(init)) \leftarrow true),\\
(att(velocity, zero, s(init)) \leftarrow true),\\
(att(acceleration, neg, s(init)) \leftarrow true)]
\end{array}
$$

12.2.3 Deriving the Second Qualitative Prediction Model

Section 12.2.1 describes a simple and direct way of describing a basic qualitative prediction in HANSEL but the specification we designed has few points of contact to the KADS diagram of Figure 12.2. For example, there are no subgoals in the definition of *qp* in clauses 12.63 and 12.64 which refer explicitly to the creation of new landmark values so we cannot connect this part of the KADS diagram to the HANSEL design. This happened because we pushed the details of landmark revision into a single predicate (*envision*) which performs all the updating of state information in one sweep. This is an elegant solution (arguably, it is more elegant than the KADS diagram) but it is worth comparing it to a design which attempts to put more of the details of landmark revision at the top level of design of *qp*. We begin with the same template that we used in Section 12.2.1.

$$
\begin{array}{lcll}
qp(State, Predicted) & \leftarrow & \tau(State) \approx \tau(Predicted) & (12.75)\\
qp(State, Predicted) & \leftarrow & \tau(State) \approx \tau(NextState) \wedge & (12.76)\\
 & & qp(NextState, Predicted) &
\end{array}
$$

We now diverge from the design in Section 12.2.1 because we wish to talk about changes to the landmarks, not just changes to the state. We introduce these by two applications of rewrite 6.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.77)$$

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Landmarks) \wedge \tau(Landmarks) \approx \tau(NewLandmarks) \wedge \tau(NewLandmarks) \approx \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.78)$$

Now we can say more about chain of relations which we introduced. Using rewrite 2 we constrain the landmark values appearing at the start of the subgoal sequence in the recursive clause to be a subset of those obtainable from the current state. Using rewrite 3 we say that the next state, carried down through the recursion, contains the new landmarks. This introduces properties 12.81 and 12.82

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.79)$$

$$qp(State, Predicted) \leftarrow \tau(State) \supseteq \tau(Landmarks) \wedge \tau(Landmarks) \approx \tau(NewLandmarks) \wedge \tau(NewLandmarks) \subseteq \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.80)$$

$$\forall X. NewLandmarks \vdash X \rightarrow NextState \vdash X \quad (12.81)$$

$$\forall X. Landmarks \vdash X \rightarrow State \vdash X \quad (12.82)$$

We next commit to a relation for specialising the state to give landmark values. This introduces property 12.85.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.83)$$

$$qp(State, Predicted) \leftarrow State \overset{specialise}{\rightsquigarrow} Landmarks \wedge \tau(Landmarks) \approx \tau(NewLandmarks) \wedge \tau(NewLandmarks) \subseteq \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.84)$$

$$\exists X. State \vdash X \ \wedge \ not(Landmarks \vdash X) \quad (12.85)$$

Following this line of design a step further, we apply skeleton 9 (defining the predicate $find\_landmarks$) to select landmark information from the current state. The skeleton introduces properties 12.91 and 12.92.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.86)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge \tau(Landmarks) \approx \tau(NewLandmarks) \wedge \tau(NewLandmarks) \subseteq \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.87)$$

$$find\_landmarks(State, Landmarks) \leftarrow empty(State) \wedge empty(Landmarks) \quad (12.88)$$

$$find\_landmarks(State, Landmarks) \leftarrow \mathcal{S}_1(X, State, T) \wedge is\_landmark(X, X1) \wedge find\_landmarks(T, T1) \wedge \mathcal{A}(X1, T1, Landmarks) \quad (12.89)$$

$$find\_landmarks(State, Landmarks) \leftarrow \mathcal{S}_2(X, State, T) \wedge not(is\_landmark(X, X1)) \wedge find\_landmarks(T, Landmarks) \quad (12.90)$$

$$find\_landmarks(State, Landmarks) \rightarrow \forall X1.X1 \in State \rightarrow \left(\begin{array}{c} \exists X2.is\_landmark(X1, X2) \ \wedge \ X2 \in Landmarks \\ \vee \\ not(\exists X2.is\_landmark(X1, X2)) \end{array} \right) \quad (12.91)$$

$$find\_landmarks(State, Landmarks) \rightarrow \forall X2.X2 \in Landmarks \rightarrow (\ \exists X1.is\_landmark(X1, X2) \ \wedge \ X1 \in State \ ) \quad (12.92)$$

We now turn to the relation establishing new landmarks from the current landmark set. This cannot be refined via a set inequality, since the new landmark set may be entirely different from the old one, so we introduce a relation directly using skeleton 1. The new predicate (*new_landmarks*) maps each landmark to a new one. It introduces properties 12.97 and 12.98.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.93)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge new\_landmarks(Landmarks, NewLandmarks) \wedge \tau(NewLandmarks) \subseteq \tau(NextState) \wedge qp(NextState, Predicted) \quad (12.94)$$

$$new\_landmarks(Landmarks, NewLandmarks) \leftarrow empty(Landmarks) \wedge empty(NewLandmarks) \quad (12.95)$$

$$new\_landmarks(Landmarks, NewLandmarks) \leftarrow \mathcal{S}(X, Landmarks, T) \wedge new\_landmark(X, X1) \wedge new\_landmarks(T, T1) \wedge \mathcal{A}(X1, T1, NewLandmarks) \quad (12.96)$$

$$new\_landmarks(Landmarks, NewLandmarks) \rightarrow \forall X1.X1 \in Landmarks \rightarrow (\ \exists X2.new\_landmark(X1, X2) \ \wedge \ X2 \in NewLandmarks \ ) \quad (12.97)$$

$$new\_landmarks(Landmarks, NewLandmarks) \rightarrow \forall X2.X2 \in NewLandmarks \rightarrow (\ \exists X1.new\_landmark(X1, X2)\ \wedge\ X1 \in Landmarks\ ) \quad (12.98)$$

Our next step is to commit to a relation generalising from the new landmarks to the next state (rewrite 19). This introduces properties 12.101 and 12.102.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.99)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge new\_landmarks(Landmarks, NewLandmarks) \wedge NewLandmarks \overset{generalise}{\rightsquigarrow} NextState \wedge qp(NextState, Predicted) \quad (12.100)$$

$$\forall X1.NewLandmarks \vdash X1 \rightarrow NextState \vdash X1 \quad (12.101)$$

$$\exists X2.NextState \vdash X2\ \wedge\ not(NewLandmarks \vdash X2) \quad (12.102)$$

Continuing with this part of the design, we introduce a predicate, *replace_landmarks*, for the generalisation from landmarks to new state, using skeleton 1. This replaces all the new landmarks in the state. The skeleton introduces properties 12.107 and 12.108.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.103)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge new\_landmarks(Landmarks, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.104)$$

$$replace\_landmarks(NewLandmarks, NextState) \leftarrow empty(NewLandmarks) \wedge empty(NextState) \quad (12.105)$$

$$replace\_landmarks(NewLandmarks, NextState) \leftarrow \mathcal{S}_1(X, NewLandmarks, T) \wedge replace\_landmark(X, X1) \wedge replace\_landmarks(T, T1) \wedge \mathcal{A}_1(X1, T1, NextState) \quad (12.106)$$

$$replace\_landmarks(NewLandmarks, NextState) \rightarrow \forall X1.X1 \in NewLandmarks \rightarrow (\ \exists X2.replace\_landmark(X1, X2)\ \wedge\ X2 \in NextState\ ) \quad (12.107)$$

$$replace\_landmarks(NewLandmarks, NextState) \rightarrow \forall X2.X2 \in NextState \rightarrow (\ \exists X1.replace\_landmark(X1, X2)\ \wedge\ X1 \in NewLandmarks\ ) \quad (12.108)$$

The skeletal definition for *new_landmarks* is intended to produce new landmarks from the current ones but, form the KADS diagram of Figure 12.2 we know that it also should take into account the information defining influences

between attribute values. We therefore add an additional argument slice to *new_landmarks*, which automatically introduces additional tests to that predicate (these have already been rewritten in the specification below) and an additional test to *qp*.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.109)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge \mathcal{T}(Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.110)$$

$$new\_landmarks(Landmarks, Influences, NewLandmarks) \leftarrow empty(Landmarks) \wedge empty(NewLandmarks) \wedge influence\_set(Influences) \quad (12.111)$$

$$new\_landmarks(Landmarks, Influences, NewLandmarks) \leftarrow \mathcal{S}(X, Landmarks, T) \wedge new\_landmark(X, X1) \wedge new\_landmarks(T, Influences, T1) \wedge \mathcal{A}(X1, T1, NewLandmarks) \quad (12.112)$$

We then rewrite the test introduced into *qp* by the argument slice for *new_landmarks* as a general relation between the tested set ($Influences$) and the current state.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.113)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge \tau(State) \approx \tau(Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.114)$$

Continuing this line of design, we constrain the influences to be included in the state (rewrite 2). This introduces property 12.117.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.115)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge \tau(State) \supseteq \tau(Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.116)$$

$$\forall X. Influences \vdash X1 \rightarrow State \vdash X1 \quad (12.117)$$

We then commit to a specialisation relation between current state and influences (rewrite 12), introducing property 12.120.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.118)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge State \overset{specialise}{\rightsquigarrow} Influences \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.119)$$

$$\exists X2.State \vdash X2 \ \wedge \ not(Influences \vdash X2) \quad (12.120)$$

This specialisation relation is then defined as a filtering operation ($find\_influences$) over the current state by applying skeleton 9. This introduces properties 12.126 and 12.127.

$$qp(State, Predicted) \leftarrow \tau(State) \approx \tau(Predicted) \quad (12.121)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge find\_influences(State, Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.122)$$

$$find\_influences(State, Influences) \leftarrow empty(State) \wedge empty(Influences) \quad (12.123)$$

$$find\_influences(State, Influences) \leftarrow \mathcal{S}_1(X, State, T) \wedge is\_influence(X, X1) \wedge find\_influences(T, T1) \wedge \mathcal{A}(X1, T1, Influences) \quad (12.124)$$

$$find\_influences(State, Influences) \leftarrow \mathcal{S}_2(X, State, T) \wedge not(is\_influence(X, X1)) \wedge find\_influences(T, Influences) \quad (12.125)$$

$$find\_influences(State, Influences) \rightarrow \forall X1.X1 \in State \rightarrow \left(\begin{array}{c} \exists X2.is\_influence(X1, X2) \ \wedge \ X2 \in Influences \\ \vee \\ not(\exists X2.is\_influence(X1, X2)) \end{array} \right) \quad (12.126)$$

$$find\_influences(State, Influences) \rightarrow \forall X2.X2 \in Influences \rightarrow (\ \exists X1.is\_influence(X1, X2) \ \wedge \ X1 \in State \ ) \quad (12.127)$$

We now turn to the base case of qp, where we constrain the predicted state to be included in the current state (rewrite 1). This introduces property 12.130.

$$qp(State, Predicted) \leftarrow \tau(State) \sqsupseteq \tau(Predicted) \quad (12.128)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge find\_influences(State, Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.129)$$

$$\forall X. Predicted \vdash X1 \rightarrow State \vdash X1 \quad (12.130)$$

Following on from this, we further commit to a specialisation relation between current and predicted state (rewrite 19). This introduces property 12.133.

$$qp(State, Predicted) \leftarrow State \overset{specialise}{\rightsquigarrow} Predicted \quad (12.131)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge find\_influences(State, Influences) \wedge new\_landmarks(Landmarks, Influences, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, NextState) \wedge qp(NextState, Predicted) \quad (12.132)$$

$$\forall X. Predicted \vdash X \rightarrow State \vdash X \quad (12.133)$$

The form of specialisation we choose between current and predicted state is simply the $find\_landmarks$ predicate which we have already built - in other words the final prediction will be whatever landmarks are in the current state. The final HANSEL definition for this form of qualitative prediction is given by clauses 12.134 to 12.146 below.

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Predicted) \quad (12.134)$$

$$qp(State, Predicted) \leftarrow find\_landmarks(State, Landmarks) \wedge find\_influences(State, Influences) \wedge new\_landmarks(Landmarks, Influences, Landmarks, NewLandmarks) \wedge replace\_landmarks(NewLandmarks, State, NextState) \wedge qp(NextState, Predicted) \quad (12.135)$$

$$find\_landmarks(State, Landmarks) \leftarrow empty(State) \wedge empty(Landmarks) \quad (12.136)$$

$$find\_landmarks(State, Landmarks) \leftarrow select\_first\_element(X, State, T) \wedge is\_landmark(X, X1) \wedge find\_landmarks(T, T1) \wedge add\_element\_first(X1, T1, Landmarks) \quad (12.137)$$

$$find\_landmarks(State, Landmarks) \leftarrow select\_first\_element(X, State, T) \wedge not(is\_landmark(X, X1)) \wedge find\_landmarks(T, Landmarks) \quad (12.138)$$

$$new\_landmarks(Landmarks, Influences, AllLandmarks, NewLandmarks) \leftarrow$$
$$empty(Landmarks) \wedge$$
$$empty(NewLandmarks) \wedge \quad (12.139)$$
$$influence\_set(Influences) \wedge$$
$$landmarks\_set(AllLandmarks)$$
$$new\_landmarks(Landmarks, Influences, AllLandmarks, NewLandmarks) \leftarrow$$
$$select\_first\_element(X, Landmarks, T) \wedge$$
$$new\_landmark(X, Influences, AllLandmarks, X1) \wedge \quad (12.140)$$
$$new\_landmarks(T, Influences, AllLandmarks, T1) \wedge$$
$$add\_element\_first(X1, T1, NewLandmarks)$$

$$replace\_landmarks(NewLandmarks, NextStateNextState) \leftarrow$$
$$empty(NewLandmarks) \wedge \quad (12.141)$$
$$empty(NextState)$$
$$replace\_landmarks(NewLandmarks, State, NextState) \leftarrow$$
$$select\_first\_element(X, NewLandmarks, T) \wedge$$
$$replace\_landmark(X, X1, State, NewState) \wedge \quad (12.142)$$
$$replace\_landmarks(T, NewState, T1) \wedge$$
$$add\_element\_first(X1, T1, NextState)$$
$$replace\_landmarks(NewLandmarks, State, NextState) \leftarrow$$
$$select\_first\_element(X, NewLandmarks, T) \wedge$$
$$not(replace\_landmark(X, X1, State, NewState)) \wedge \quad (12.143)$$
$$replace\_landmarks(T, State, T1) \wedge$$
$$add\_element\_first(X, T1, NextState)$$

$$find\_influences(State, Influences) \leftarrow empty(State) \wedge \quad (12.144)$$
$$empty(Influences)$$
$$find\_influences(State, Influences) \leftarrow select\_first\_element(X, State, T) \wedge \quad (12.145)$$
$$is\_influence(X, X1) \wedge$$
$$find\_influences(T, T1) \wedge$$
$$add\_element\_first(X1, T1, Influences)$$
$$find\_influences(State, Influences) \leftarrow select\_first\_element(X, State, T) \wedge \quad (12.146)$$
$$not(is\_influence(X, X1)) \wedge$$
$$find\_influences(T, Influences)$$

12.2.4 Running the Second Qualitative Prediction Model

As in Section 12.2.2, we must supply the definitions for the predicates not built directly in HANSEL in order to run the qualitative model. The definitions of *empty*, *select_first_element* and *add_element_first* are identical to those in Section 12.1.2 (clauses 12.1.2, 12.1.2 and 12.1.2). To design the remaining predicates we first need to decide on the forms of expressions which we need in the axiom sets. These can be similar to those from our previous example (in Section 12.2.2) but can be simplified because our new qualitative simulator replaces old with new landmarks at each stage in the recursion rather than simply adding them to the set of axioms, so the landmarks don't need to have an argument denoting the state to which they belong. Thus a landmark is denoted by the expression $att(A, V)$, where A is the attribute name and V is its value. The clauses determining the next value for an attribute are also adaped to be in the

form $next\_att(A, V) \leftarrow P$ where the conclusion of the clause is the value, V, for attribute, A, in a successor state and P is the condition in the current state. All other elements of the axiom set are as given in Section 12.2.2.

We need to add predicates for identifying which axioms in our state description are landmarks and which are used in determining influences. The former is straightforward, since landmarks are all clauses of the form $att(A, V) \leftarrow true$. The latter is similarly straightforward, since the information we need is described by the *next_att*, *trans* and *qneg* predicates. The new predicates we need here are therefore as given below.

$$is\_landmark((att(A, V) \leftarrow true), (att(A, V) \leftarrow true)) \tag{12.147}$$

$$is\_influence((next\_att(A, P) \leftarrow B), (next\_att(A, P) \leftarrow B)) \tag{12.148}$$
$$is\_influence((trans(V1, V2, V3) \leftarrow true), (trans(V1, V2, V3) \leftarrow true)) \tag{12.149}$$
$$is\_influence((qneg(V, D) \leftarrow true), (qneg(V, D) \leftarrow true)) \tag{12.150}$$

Finally, we need predicates which replace old with new landmarks in the current state and which generate a new landmark from a state. The first of these requires simply that the old landmark be removed from the state to which the new one will be added. The second is similar to the *envision* predicate defined in clause 12.59 of Section 12.2.2, in that it applies the basic deduction proof rules (clause 9.32 to 9.35) to obtain the new landmark. It differs from *envision* by having to take the union of landmark and influence axioms as the basis for deduction, and by applying deduction ot a copy of that axiom set rather than to the original so as to avoid instantiatign variables in transition clauses between states.

$$\begin{aligned} &replace\_landmark((att(A, V) \leftarrow true), (att(A, V) \leftarrow true), State, NewState) \leftarrow \\ &\quad remove((att(A, V1) \leftarrow true), State, NewState) \end{aligned} \tag{12.151}$$

$$\begin{aligned} &new\_landmark((att(A, V) \leftarrow true), Influences, Landmarks, (att(A, V1) \leftarrow true)) \leftarrow \\ &\quad union(Influences, Landmarks, State) \wedge \\ &\quad copy\_term(State, State1) \wedge \\ &\quad deduce(basic, proof(State1, next\_att(A, V1))) \end{aligned} \tag{12.152}$$

We can now generate qualitative behaviours corresponding to the example in Figure 12.3 by finding solutions to the goal given in expression 12.153 below, which is similar to expression 12.74 in Section 12.2.2.

$$
\begin{array}{ll}
qp([& (att(position, pos) \leftarrow true), \\
& (att(velocity, zero) \leftarrow true), \\
& (att(acceleration, neg) \leftarrow true), \\
& (next\_att(position, P) \leftarrow \ att(position, Pp) \wedge \\
& \qquad\qquad\qquad\qquad\ att(velocity, Vp) \wedge \\
& \qquad\qquad\qquad\qquad\ trans(Pp, Vp, P)), \\
& (next\_att(velocity, V) \leftarrow \ att(velocity, Vp) \wedge \\
& \qquad\qquad\qquad\qquad\ att(acceleration, Ap) \wedge \\
& \qquad\qquad\qquad\qquad\ trans(Vp, Ap, V)), \\
& (next\_att(acceleration, A) \leftarrow \ att(position, Vp) \wedge \\
& \qquad\qquad\qquad\qquad\ qneg(Vp, A)), \\
& (trans(pos, pos, pos) \leftarrow true), \\
& (trans(pos, zero, pos) \leftarrow true), \\
& (trans(pos, neg, zero) \leftarrow true), \\
& (trans(zero, pos, pos) \leftarrow true), \\
& (trans(zero, zero, zero) \leftarrow true), \\
& (trans(zero, neg, neg) \leftarrow true), \\
& (trans(neg, pos, zero) \leftarrow true), \\
& (trans(neg, zero, neg) \leftarrow true), \\
& (trans(neg, neg, neg) \leftarrow true), \\
& (qneg(pos, zero) \leftarrow true), \\
& (qneg(zero, pos) \leftarrow true), \\
& (qneg(zero, neg) \leftarrow true), \\
& (qneg(neg, zero) \leftarrow true), \\
& (qneg(neg, pos) \leftarrow true)], X)
\end{array}
\tag{12.153}
$$

Unlike the example of Section 12.2.2, the solutions obtained are for the current set of landmarks only, not all the landmarks up to the current point. Nevertheless, we get the same simulation of behaviour, corresponding to Figure 12.3 by generating successive sets of landmarks on backtracking, as shown in sequence below.

$$
\begin{array}{c}
[(att(position, pos) \leftarrow true), (att(velocity, zero) \leftarrow true), (att(acceleration, neg) \leftarrow true)] \\
[(att(position, pos) \leftarrow true), (att(velocity, neg) \leftarrow true), (att(acceleration, neg) \leftarrow true)] \\
[(att(position, zero) \leftarrow true), (att(velocity, neg) \leftarrow true), (att(acceleration, zero) \leftarrow true)] \\
[(att(position, neg) \leftarrow true), (att(velocity, neg) \leftarrow true), (att(acceleration, pos) \leftarrow true)] \\
[(att(position, neg) \leftarrow true), (att(velocity, zero) \leftarrow true), (att(acceleration, pos) \leftarrow true)] \\
[(att(position, neg) \leftarrow true), (att(velocity, pos) \leftarrow true), (att(acceleration, pos) \leftarrow true)] \\
[(att(position, zero) \leftarrow true), (att(velocity, pos) \leftarrow true), (att(acceleration, zero) \leftarrow true)] \\
[(att(position, pos) \leftarrow true), (att(velocity, pos) \leftarrow true), (att(acceleration, neg) \leftarrow true)]
\end{array}
$$

12.3 Using the Example Properties for Testing

The three examples of this this chapter have yielded numerous properties which can be instantiated by running tests using the meta-interpreter of Section 9.7. This is capable of instantiating the properties but deliberately avoids prescribing how to prove them. In this section we give an example of how to do this using Prolog. First, each property is translated from its current predicate calculus

definition into an equivalent Prolog goal, using Lloyd-Topor transformations [Lloyd, 1985]. For example property 12.20 is translated into the Prolog goal:

$$not\left(\begin{array}{c} abstract(CaseDescription, AbstractedCase) \wedge \\ X2 \in AbstractedCase \wedge \\ not\left(\begin{array}{c} generalisation(X1, X2) \wedge \\ X1 \in CaseDescription \wedge \\ X1 \in AbstractedCase \end{array}\right) \end{array}\right)$$

The full set of properties translated in this way appears in Appendix D. All of the terms in these translated properties are either predicates defined in the specifications or correspond to the normal Prolog connectives, with the exception of two terms which we define here:

- The expression $X \in S$ is defined as the normal recursive definition of list membership in Prolog (since all of the examples use Prolog lists as the data structure to represent axiom sets.
- The expression $S \vdash X$ is defined in terms of the sets of deductive inference rules discussed in Section 9.6. We do this by connecting to clause 9.29 using the definition:

$$S \vdash X \leftarrow deduce(F, S \vdash X) \tag{12.154}$$

  where the flag used to choose a specific set of proof rules, F, is left as a variable to permit any set of deductive proof rules to be used.

We can now check for contradictions to any of the instances of the properties accumulated in the examples of this chapter by taking the negation of the corresponding Prolog goal from Appendix D and attempting to solve it in Prolog. If we succeed then we have a refutation of the property and a potential problem in our specification.

12.3.1 Results of Testing the Examples

We applied the testing procedure described above to the three examples of Sections 12.1, 12.2.1 and 12.2.3. This involved running the tests described earlier for the programs of those sections, using the meta-interpreter of Section 9.7 to accumulate test instances for appropriate properties. We then attempted to satisfy those test instances using normal Prolog, as described above. As expected, most of the tests succeeded but there where two surprises:

- Some of the tests on the qualitative simulation programs of Sections 12.2.1 and 12.2.3 were non-terminating. The cause of this was the need to satisfy instances of goals such as property12.81 which contain universally quantified goals to be proved from an axiom set via the deductive proof rules. In translating these to Prolog we converted them into negated existential goals, using the equivalence: $\forall X.P(X) \rightarrow Q(X) \leftrightarrow$

$not(\exists X.P(X) \ \wedge \ not(Q(X)))$ Solving these goals in standard Prolog requires exhaustive search for an instance of the goal, with negation as failure being used to assume its falsity in the absence of a solution. The problem is that if we attempt exhaustively to generate instances of X for a goal such as $State \vdash X$, where $State$ is an axiom set from our qualitative simulation examples, then we may be able to generate new instances indefinitely as we simulate further and further forward in time. This creates our problem of non-termination. Rather than attempt to fix the problem by making our test harness more sophisticated we simply left out the offending properties.

- In the assessment example of Section 12.1 we obtained one unexpected failure to preserve property 12.11. The instance of this property which caused the problem was:

$$\exists X. \left\{ \begin{array}{c} (middle\_aged(dave) \leftarrow true), \\ (management\_skills(dave) \leftarrow true) \end{array} \right\} \vdash X \wedge not(\left\{ \begin{array}{c} (middle\_aged(dave) \leftarrow true), \\ (management\_skills(dave) \leftarrow true) \end{array} \right\} \vdash X) \quad (12.155)$$

  This problem occurs because the property was created by the introduction of a specialisation relation between $SystemModel$ and $Norm$ in the partial program described by clause 12.10. The expectation was that the $Norm$ would always be more specific than the $SystemModel$ but it turns out that for simple system models the norm is the same. Thus we have a breached property but it is not an error.

What would happen if we really do have an error in the program? To demonstrate this we altered clause 12.31 of the final specification in Section 12.1 to:

$$\begin{array}{l} abstract(CaseDescription, AbstractedCase) \leftarrow \\ \quad select\_first\_element(X, CaseDescription, T) \wedge \\ \quad setof(X1, generalisation(X, X1), Set) \wedge \\ \quad abstract(T, T1) \wedge \\ \quad union(Set, T1, AbstractedCase) \end{array}$$

All we have done is to stop the element X, from which we performed a generalisation, being added to the *AbstractedCase*. When we now run the example and check its property instances, as before, we generate the following three property instances which cannot be satisfied. To make these simpler to read we write $\mathcal{C}$ in place of the case description instance:

$$\{(age(dave, 36) \leftarrow true), (leads\_group(dave) \leftarrow true)\}$$

and $\mathcal{A}$ in place of the abstracted case description:

$$\left\{ \begin{array}{l} (middle\_aged(dave) \leftarrow age(dave, B) \ \wedge \ B \geq 30 \ \wedge \ B < 50), \\ (old(dave) \leftarrow age(dave, C) \ \wedge \ C \geq 50), \\ (young(dave) \leftarrow age(dave, D) \ \wedge \ D < 30), \\ (experienced(dave) \leftarrow leads\_group(dave)), \\ (management\_skills(dave) \leftarrow leads\_group(dave)) \end{array} \right\}$$

Our three unsatisfied properties are then:

$$\forall X. \mathcal{C} \vdash X \rightarrow \mathcal{A} \vdash X \quad (12.156)$$

$$\exists X. \mathcal{A} \vdash X \ \wedge \ not(\mathcal{C} \vdash X) \quad (12.157)$$

$$abstract(\mathcal{C},\mathcal{A}) \rightarrow$$
$$\forall X2.X2 \in \mathcal{A} \rightarrow \left(\begin{array}{c} \exists X1. \left(\begin{array}{c} generalisation(X1,X2) \wedge \\ X1 \in \mathcal{C} \wedge \\ X1 \in \mathcal{A} \end{array} \right) \\ \vee \\ \left(\begin{array}{c} not(\exists X1.generalisation(X1,X2)) \wedge \\ X2 \in \mathcal{C} \end{array} \right) \end{array} \right) \quad (12.158)$$

Property 12.156 is an instance of clause property 12.4. It originated in an early design stage when we stipulated that the abstracted case description should be at least as large as the case description. Its failure tells us that this is not the case, creating a problem in the *assessment* predicate. Property 12.157 is an instance of clause property 12.12. It originated a little later in the description of the *assessent* predicate, when we decided to use a generalisation relation between the case description and the abstracted case. Its failure tells us that the relation we are using to actually do this isn't yielding the more general axiom set we expected. Property 12.158 is an instance of property 12.20. It was introduced along with the skeleton for *abstract*. Its failure tells us that the mapping expected to be implemented by this skeleton has not been preserved. This is, indeed the root of our problem, because our alteration to *abstract* stopped it from including in the abstracted case all the original elements of the case description along with those obtained through the *generalisation* mapping. This, in turn led to failure of the clause properties.

We have just demonstrated an example of using properties accumulated during design to find potential problems when programs are altered. This does not, however, tell us how effective such mechanisms might be in practice. Effectiveness depends on the errors we trap being the ones which are pivotal to the quality of design. Even if they are pivotal, effectiveness also depends on being able to move swiftly from detecting violated properties to finding the source of the problem. This is seldom as easy as we have made it appear in this example. Thus, there are still substantial obstacles to be overcome before this sort of method becomes of practical use.

12.4 Return to Problems of Section 7.3

The development of the HANSEL system was stimulated by problems experienced with the distributed style of design used in the LSS system. Section 7.3 summarises these problems under five subsections, which we now re-visit to consider whether HANSEL gives any improvement in these areas.

12.4.1 Choice of Style (from Section 7.3.1)

A problem experienced in LSS is that each tool requires designers to adopt a narrow style of description and normally is only able to produce a limited class of Horn clause specifications. One must know in advance of using the tool whether

its style of description suits the problem in hand. The HANSEL system avoids this problem in one sense by having a uniform style of description: there is only one choice of descriptive style. Unfortunately, this doesn't solve the problem completely because the single style of description used in HANSEL, although more general than the LSS tools still requires the designer to adopt a particular view of problem description (initially in a set-based style and refining to the use of skeletons and extensions). We know from the examples in this thesis that numerous problems can be made to fit this mould but it is not clear whether it is a convenient style of description.

Evaluations of related systems, such as techniques editors (Section 7.1), suggest that to connect to problem domains it is necessary to provide domain-specific templates and refinements which domain experts recognise readily. This could be done with HANSEL but we would then be back to the LSS situation with multiple tools and the problem of selecting them. Nevertheless, the explicit lifecycle model which HANSEL provides might make this less of a problem than in LSS because there is already a natural division into tools corresponding to the refinement tasks of Figure 10.1. If specialisation of HANSEL takes place by specialising each of these tools, while keeping the refinement framework intact, then the selection of tools could be easier because only a few would apply at each refinement stage.

12.4.2 Maintaining an Overview (from Section 7.3.2)

Perhaps the largest problem of distributed design systems, and of formal knowledge sharing in general, is keeping track of the "big picture" of a large design when working on a small portion of it. The LSS system attempts to do this simply through sharing specifications between tools in a standard formal language, but this language only describes what each tool has done. It does not describe the overall design into which each part of the specification must fit. The overview tool in LSS does allow the structure of predicates in a large design to be viewed but only after the design has been completed, so overall coordination in LSS is retrospective.

The refinement framework in HANSEL takes the opposite view. The entire design is coordinated from the beginning and new definitions can only be added if gaps exist in the developing specification. Furthermore, the properties which are associated with refinements allow the appropriateness of design choices to be tested during verification of the design. Earlier in this this chapter we gave examples of properties being accumulated during HANSEL design and in Section 12.3we gave examples of their use in testing. The incorrect *subprocess*1 definitions in clauses 7.9, 7.10 and 7.11 of Section 7.3.2 could be detected by requiring the following properties to hold for variables S_i and S_f of clause 7.1 in section 7.1:

- If we require that S_f is a set then the type error introduced in clause 7.9 can be detected.

- If we require that $\forall X.X \in S_f \rightarrow X \in S_i$ then we detect the error introduced by clause 7.10.
- If we additionally require that $\exists X.X \in S_i \ \wedge \ not(X \in S_f)$ then we force *subprocess*1 to make the argument driving recursion in *process* smaller, hence protecting against the possibility of non-termination introduced by clause 7.11.

The price we pay for his sort of coordination in HANSEL is that we must adopt its style of coordination. There is also an interface limitation since the way this is presented is by having a central specification which is viewed by designers and from which all the refinements are coordinated. There is a human limit on the size which this specification can reach before it becomes impractical to view it, in the same way that it is impractical to view the whole of a large conventional program in a single file.

12.4.3 Saying Less (from Section 7.3.3)

All of the LSS tools describe specifications with Horn clauses which may contain open test and update subgoals, as described in Section 4.1. This gives a narrow repertoire of concepts for representing stages in precision of definitions. The HANSEL system extends this repertoire by including notions of set inequality, allowing it to express intermediate stages in design like the ones given in clauses 7.18 and 7.19 of Section 7.3.3.

The ability to perform refinement in this way comes at a price: the specifications which it is appropriate for HANSEL to construct are those for which it is possible to think initially about the problem in terms of relations between sets. It would, therefore, be inappropriate to use HANSEL, for example, to define the meta-interpreter of Section 9.7 because this problem does not invite us to think about it in a set-based style.

12.4.4 Transformations Over Many Predicates (from Section 7.3.4)

With the exception of its retrospective overview tools, which are not concerned with designing new specifications, the LSS tools are fixated on individual predicates. This makes it vulnerable to adaptations of the specification where changes in one predicate should be reflected in adaptation of others. The example given in Section 7.3.4 involved the addition of an argument slice to one predicate, which requires that argument to appear in all subgoals which call that predicate. HANSEL copes with this type of problem because it has a central specification (so the appropriate change can be rippled through the specification) and a formal notation which allows the newly introduced variables to be flagged by tests which, through the refinement mechanism, allow them to be connected to other variables in the clauses where they were added.

Of course, HANSEL cannot adapt to all interactions between predicate definitions because some of them may not be as direct as the example. For instance,

we might make a change in the definition of a predicate which had the effect of restricting the kind of axiom set it could deal with. In some circumstances it might be useful to test that the axiom set concerned is of the appropriate form before calling the predicate but it is difficult to predict whether this is necessary and, if it is, what the check should be. However, the use of properties associated with refinements allows at least some retrospective checking that properties required by one part of the design are preserved by others.

12.4.5 Maintaining Properties During Use and Revision (from Section 7.3.5)

LSS gives no mechanism for describing properties of predicates so it cannot test whether these hold of the design. HANSEL allows properties to be associated with clauses and predicates so it provides an expressive way of recording this type of information along with design steps. It also gives a framework for accumulating properties at no additional cost to the designer as a specification is built, and Section 9.7 demonstrates how simple meta-interpretation methods can use these definitions to generate tests relevant to particular executions of the specification. This is clearly an improvement over LSS but problems remain. The mechanism by which the testing information supplied by HANSEL is fed back into re-design when requirements have been violated is not yet understood, although it seems likely that routes exist (see Section 12.5.1).

A further problem, first raised in Section 12.3.1, concerns properties (like the one in the example of Section 7.3.5) which relate to the termination of predicates. There is no problem in recording appropriate properties in HANSEL but there could be a problem in generating test instances of those properties by execution of the specification using a meta-interpreter like the one in Section 9.7, since we are most interested in the properties related to non-termination of predicates exactly when the predicate does not terminate. In this case our simple meta-interpreter will not yield test instances. One solution to this problem is to use more sophisticated meta-interpreters which ensure termination (*e.g.* by imposing a depth limit on proofs).

12.5 New Problems Raised by HANSEL

The HANSEL framework solves some of the problems observed in LSS but it is not all good news. To conclude our evaluation we discuss two problems which are raised by the move to a more tightly integrated form of lifecycle.

12.5.1 Managing Requirements Expressed as Properties

It is easy in HANSEL to accumulate numerous properties for clauses and predicates when building designs. One of its advantages is that designers need not even be aware that these are being accumulated - they come for free as an adjunct to the design method. Nevertheless, to be useful they must at some stage become

visible to human engineers. In the current system this is intended to happen in testing (see Section 9.7) where examples are used to execute the specifications, and as predicates and clauses are used in the execution the corresponding instances of properties are collected. This raises two problems:

- A large number of instances of properties can be generated. For example, the specification of Section 12.2.3 contains 17 properties and in each test we perform on the specification these are likely to be instantiated several times (perhaps tens of times for recursive clauses where the recursion is deep). Therefore, it is possible to generate hundreds of instances of properties for moderately sized tests. Checking all of these is time consuming, even when done automatically.

- Even if all the properties can be checked there is the problem of deciding what to do about those which are refuted by the specification. The means of obtaining a refutation of a property in the current system is simply to prove the negation of the property (see Section 9.7) but the instance obtained for this goal is not guaranteed to give sufficient information to pinpoint the source of the problem, since a problem detected in one part of the specification may be a knock-on effect of an error made elsewhere. Even if the source of the error can be determined, this is unlikely on its own to determine how the specification might be revised to correct it.

In short, HANSEL is good at generating properties of specifications but gives no support in interpreting the results of testing them. It can be used to find potential problems but does not tell us for sure where the errors leading to those problems originate, nor does it offer corrective advice.

12.5.2 Adapting the Refinement Framework to Domains of Application

The HANSEL system of refinement has been made as general and task-independent as possible so that we could concentrate on the abstract aspects of this kind of formal lifecycle. It is rare, however, that such abstract frameworks become popular with requirements or software engineers because they demand an understanding of formal concepts outside of the normal experience of these groups of people. Consequently, it is unrealistic to expect HANSEL as it currently exists to be used in application domains. One way of adapting it so that it might be used would be to make its refinement rules and skeletons domain-specific while retaining the overall framework. This is technically possible, since the libraries of refinements and skeletons can easily be adapted without altering the other parts of the system. We do not know, however, whether the resulting system would then be close enough to the understanding of some significantly sized group of designers or whether we would need further adaptations to the framework itself, and the interface in particular, to bring it close enough to styles of design understood in that domain.

This sort of experiment in restricting an abstract framework to a domain has not been attempted in HANSEL but an estimate of the potential for success can be obtained from experience of building similar systems. The lessons from these are:

- Adaptation of abstract design methods to domains of application takes significant amounts of time, with much of the effort going into traditional knowledge engineering tasks of identifying the forms of domain knowledge which must be represented in the method and describing it in ways which suit both method and human designers. In the ECO project [Robertson et al., 1991] the abstract method of design was based on parameterisable components which were used within an interactive synthesiser to construct a class of population dynamics models. Much of the three man-years spent on that part of the project was used in understanding how to describe the domain in a way suited to this form of representation. The TeMS system summarised in Section 7.1.2 tackled a similar problem using techniques editing and required a similar investment of effort in knowledge acquisition and representation.

- It is possible in theory to make the refinement libraries of an automated synthesiser domain specific without altering the interface used to apply them, but in practice the interface is often changed significantly to conform to the expectations of designers in the target domain. The TeMS system (Section 7.1.2) could have used a generic techniques editor to construct much of the specification which it synthesises but, instead, it has a domain-specific user interface which interacts with the generic forms of techniques application used inside the system. This effect, of a generic synthesis mechanism sandwiched between a domain-specific user interface and domain-specific refinement libraries, occurs because domain experts want the design method to be familiar as well as the design knowledge.

- A more positive consequence of the item above is that, if appropriate user interfaces and refinement libraries can be devised then basic methods of formal synthesis often do transfer between domains, so the core technical ideas persist. Examples of this appear in Chapter 5 of [Robertson and Agusti, 1999]. Therefore, for HANSEL we expect that the core framework of refinement and skeleton application might transfer into target domains but that the user interface and refinement/skeleton libraries would need to be adapted.

Part IV

Conclusions

Chapter 13

Contributions of this Research

The overall aim of the work reported in this thesis is to explore the relationship between pragmatic styles of design and a formal specification language. This is a boundless topic so we confined our attention to a specific formal language: Horn clauses made executable through Prolog. We also chose two contrasting styles of design: the first viewing design as a distributed activity in which different tools communicate via a shared language; the second viewing design as a centrally controlled process in which choice of tool is determined by previous design steps and information useful in verifying designs is derived as an integral part of the process. We now summarise the main technical contributions of the thesis in Section 13.1 and, in Section 13.2, summarise how the research relates to the general questions raised at the end of Chapter 1.

13.1 Main Technical Contributions

The technical innovations of this thesis emerged as part of the exploration of pragmatics in early design and were therefore driven by practical need rather than the urge to explore a particular aspect of theory. We summarise the main results below, following the chronology of the thesis.

- Novel forms of design tool were developed in the LSS system. The means of coordination between tools (via a shared language) is not new but, to the best of our knowledge, the program described in Section 7.2 is the largest yet built with the assistance of this kind of system, thus setting a practical target for other implementations. The contributions from individual tools are:
  - One of the earliest implementations of a techniques editor which is usable for significantly sized design tasks (Section 5.1).

  - A tool allowing a diagrammatic representation of sequences to be translated into Definite Clause Grammar clauses, allowing this form of logic program to be built visually by direct manipulation (Section 5.2).
  - A tool allowing graphical representation of recursive logic programs, with an automatic translation to partial predicate definitions suitable for completion by a techniques editor (Section 5.3).

- The HANSEL system is the first to combine set-based refinement, for high level design, with techniques editing, for lower level design (Chapter 10). In doing this, it was necessary to invent: a set of generic rewrite rules for the high-level refinements (Section 9.4); a library of task-specific skeletons (Section 9.5); and a means of introducing more sophisticated systems of proof beyond those described in the skeleton library (Section 9.6).

- The set-based representation used in the early stages of HANSEL design uses Horn clauses defined over axiom sets which themselves contain Horn clauses (Section 9.2). This gives a flexible, abstract and compact form of specification which is related to other systems of set refinement but is used here in a novel way. The evaluation examples of Chapter 12 are, to our knowledge, the first instances of this form of set based refinement being used to describe early knowledge engineering models.

- The use of meta-interpretation to generate test examples (Section 9.7) is not new but combining this with a refinement system which associates properties with clauses (Section 9.3) is novel and provides an additional link between early specification and later testing.

13.2 Return to the Questions of Chapter 1

Having described the technical contributions of this thesis, we now return to the broader questions raised in Chapter 1.

13.2.1 Sharing Design Knowledge Using Techniques-Based Specifications

Chapter 3 gives an overview of the LSS system which is an example of a system containing numerous different tools, each targeted at a particular style of design. A particular style of representing partial specifications (originating in techniques editing) was used uniformly to give a language of interchange between design tools. This made a number of interactions between tools possible, when they would have been impossible using normal logic programs. The window editor and diagrammatic recursion editor, in particular, are only able to produce partial programs which must be refined using the techniques editor.

In the evaluation of LSS (Chapter 7) it was shown that, although substantial specifications can be built using this system, it contains a number of weaknesses.

It is not always easy to see which tool suits which specification problem. It can be difficult to get a sense of the overall specification when working with an individual tool on a small part of it. The notion of partiality which we obtain using the techniques-based notation does not describe some of the shades of meaning which might be useful. The assumption that tools can focus on individual predicates ignores commonly occurring interactions between predicates during design. There is no mechanism for sharing, along with the partial descriptions of the specifications, information about the properties we expect our completed specifications to possess.

In summary, our contribution here is to have shown that a distributed design system based on a conventional form of techniques editing can build substantial specifications, expressed as logic programs, but it does not provide many of the facilities which we would expect from a well integrated software engineering process.

13.2.2 A Refinement System for Coordinating a Class of Designs

Many systems of formal refinement exist but these are normally far removed from conventional logic programming. Our aim was to supply some of the coordination which was lacking in LSS while staying close to forms of representation and computation familiar from logic programming. In the HANSEL system we invented a way of doing this for problems that can be described using predicates which manipulate sets of Horn clauses (without negation). This gives a simple style of refinement where early designs are composed from set inequalities and are refined into normal logic programs via a generic library of rewrite rules and a task-specific library of skeletons. The evaluation of Chapter 12 shows how this form of coordinated design can be applied to tasks inspired by a conventional high-level methodology for designing knowledge-based systems, although we do not claim that it is a replacement for these. The basic method of description and refinement is given in Chapter 9. We claim that this style of formal refinement gives a novel combination of existing methods of set-based specification, techniques editing and association of properties with predicates.

The framework for refinement in HANSEL is generic but the library of skeletons is task-specific. It is necessary to have a methodical way of constructing such libraries. In Section 9.5.2 we demonstrate how part of the current library was defined in a methodical way, using the properties associated with skeletons to help map out the space of possibilities for a precisely defined task. This sort of methodical population of task-specific libraries is a subject which is often ignored in modelling methods but is important in ensuring adequate coverage of a class of problems.

13.2.3 Accumulating Test Goals During Refinement

In conventional refinement systems the closest relatives to HANSEL properties are algebraic specifications normally used to supply the first step in synthesis,

with predicate definitions being synthesised in response to properties described by the designer. The HANSEL system inverts this view. It takes Horn clauses (expressing early designs) as the starting point and gives a means of accumulating properties as an integral (but hidden) part of design. This means that the synthesis is more like simple structural editing of a logic program, with the accumulated information about properties being used for testing rather than to direct synthesis. The basic mechanisms for this are described in Chapter 9 and shown in action in Chapter 12. We claim that this has not been done before in this sort of refinement framework. It opens an interesting area of research into the interaction between design and verification in logic-based systems.

Having given a means of accumulating properties of specifications it is also necessary to have a means of testing them. There are many possibilities for doing this and difficult problems which remain to be addressed (see Section 12.5.1) but we have taken advantage of our close connection to logic programming to enable automated testing using a simple form of meta-interpreter and conversion of properties to Prolog goals (Sections 9.3, 9.7 and 12.3).

13.2.4 Tools Supporting Distributed and Coordinated Design of Logic Programs

As vehicles for experimentation we built two design systems: LSS (described in chapters 3 to 7) and HANSEL(described in chapters 8 to 12). These are novel in themselves: LSS being the only system of which we are aware which attempts the design of logic programs using a collection of specialised but interacting tools; HANSEL being the only system in existence with its particular mixture of set-based refinement and techniques editing. We make no claims that these systems will be easily used by designers - on the contrary, we believe that the only route to wider use of such tools is through producing more domain-specific versions of them. Nevertheless, we were able to demonstrate that specifications which are substantial (by logic programming standards) could be built using these tools (see Chapters 7 and 12).

In building the interfaces for LSS and HANSEL we chose two contrasting routes. In LSS we attempted where possible to use graphical notations focussed on a particular specification task. These could be diverse because we assumed no interaction between tools other than through the interchange language. In HANSEL we used a uniform textual style of description which is close to the underlying formal language, since the task of developing a unifying textual/graphical language seemed beyond our resources. Nevertheless, some integration of textual and graphical notations already occurs in HANSEL because the menus used in choosing skeletons use graphical summaries of the skeletons, like the ones in Section 9.5.2, to summarise the behaviour expected from these components.

This concludes our summary of contributions of research. In the next, and final, chapter we look forward to further work.

Chapter 14

Future Work

In earlier chapters we flagged a variety of areas where there were unsolved problems or where there were unexplored possibilities for further research. In this concluding chapter we touch upon those areas which we believe would merit further study.

14.1 Appropriate Choice of Properties

The HANSEL design system is not predicated on the existence of complete sets of properties for each structural feature introduced into the specification. This allows it to operate somewhere in between structure editors such as techniques editors, where properties are seldom used explicitly in the design process, and constructive synthesis systems (see Section 2.1), where design requires all properties to be given. Where HANSEL sits between these two extremes depends on how comprehensively properties are related to structural refinements. This flexibility is a feature of this style of design, allowing greater or lesser emphasis to be given to properties according to taste. There is, however, benefit in developing more rigorous methods for choosing properties, other than simply allowing designers of refinements to invent them as they see fit.

Perhaps the most obvious area where there is room for improvement is in being parsimonious with the assignment of properties to refinements, given the properties assigned to earlier refinements. In the current system, some properties of early refinement steps are replicated in later refinements. For example, Rewrite 17 of Section 9.4.2 introduces the property $\forall X2.S2 \vdash X2 \rightarrow S1 \vdash X2$ but this rewrite applies to a specialisation of the form $\tau(S1) \supseteq \tau(S2)$. To have reached this point we must have applied one of the other rewrites which introduced the specialisation (one of rewrites 1, 2, 8, 11, 15 or 16). Any of these would already have introduced the same property. It might be helpful to have methods of narrowing the set of properties associated with each refinement to just those specific to it, and not subsumed by earlier properties. This might be done by checking of refinement libraries prior to design or by checking during

design.

14.2 Making Better Use of Properties

Sections 9.3, 9.7 and 12.3 describe how the properties accumulated during design in HANSEL may be used to generate tests for logic programs, with the implementation of these tests being done using a normal Prolog interpreter. This has the advantage of simplicity but is only one of many possible uses for these sorts of properties. We could make better use of them in both testing and in guiding the design.

The current system of testing is primitive because it provides no help in selecting appropriate tests, determining whether difficult tests are satisfied or interpreting the results of tests. The problems in each of these areas are:

- The test instances generated by the meta-interpreter of Section 9.7 depend on the right test queries being given to the logic program. Selecting the appropriate test cases is a notoriously difficult problem because the cases should ideally trap all errors at minimal cost in analysis, but we don't know exactly what the errors are and we are seldom certain of the cost of analysis.

- We currently use a standard Prolog interpreter to check the test instances with respect to the logic program. In Section 12.3 we noted that there are problems in doing this because we have no guarantee that there will be a terminating proof for every test. We cannot solve this problem in general, unless we limit the range of properties we consider, but we could use more robust and more powerful proof methods. For some kinds of application it might even be possible to connect to recent work in model checking (see [Heitmeyer et al., 1996] for an example of model checking of system requirements).

- The properties currently used in HANSEL are sometimes difficult to read, especially when translated into standard Prolog queries. This, and the fact that failures of tests don't necessarily reveal the sources of errors, means that even if the tests suggested by HANSEL find anomalies we may not be able to figure out from these failures where the problem lies. Even if we find the problem, we may not choose the appropriate re-design to get around it. This problem, in its totality, is extremely difficult but the fact that we know a great deal about the structures which contribute properties (and hence give rise to failed tests) might give some insight into strategies for re-design.

Our decision not to use properties to guide design but only to use them for retrospective testing was made for reasons of expediency. There was not time to explore both issues and testing seemed more in need of attention than design. Nevertheless, there is great scope for making use of the properties attached to rewrites and skeletons in order to guide design. In particular, it would

be possible to check the compatability of properties when giving menus of options for refinements. For example, a property associated with Rewrite 3, which introduces a generalisation from $S1$ to $S2$, is: $\forall X.S1 \vdash X \rightarrow S2 \vdash X$. If we satisfy this relation between $S1$ and $S2$ using Skeleton 15 (in Appendix C), which forms $S2$ from an appropriate element of $S1$, then we add the property: $\exists X1.X1 \in S1 \wedge R(X1) \wedge \mathit{singleton}(X1, S2)$ We would then have two properties which, if $S1$ contains more than one element, would be inconsistent. It might be more effective to detect these potential inconsistencies during design rather than catching them in later testing, since designers might have fewer choices to make and the complexity of testing might be lowered. This takes us back to our earlier remarks about the tension between design constained by properties versus design audited by testing. Some balance should be possible but the current version of HANSELis at the latter extreme.

14.3 Making More Use of Specialised Inference Methods

In Section 9.6 we described how sets of inference rules specialised to particular forms of inference could be connected to the refinement system in the same way as skeleton definitions. The idea is to describe "families" of inference methods using sets of proof rules and then supply a single interpreter for each family of proof rules. This allows variants of inference methods to be produced by altering the proof rules, while the proof strategy (as defined by the interpreter) remains the same. We showed how basic forms of deduction and abduction can be described in this way but this is only a tiny proportion of what is possible. These families are very large in themselves and, beyond them, other major families of inference methods are noticeable by their absence - in particular induction methods for generalising axiom sets are not included. There is much scope for extending this part of the system.

Before investing this effort, however, it would be necessary to understand in greater depth how designers could control the use of this sort of refinement. For the other skeletons, such as those for traversal or search, there is an obvious style of explanation for each family of skeletons (we explained this for a group of traversal skeletons in Section 9.5.2). The more flexible method of defining skeletons using proof rules allows diverse variants to be described through small changes in the proof rules. Organising all of this in a way which can be explained methodically to designers is a difficult task.

14.4 Restricting HANSEL to Domains and Tasks

The most effective uses of logic based methods in early design seem to be those which are specialised to particular tasks or domains. The MECO, ECO and Amphion systems of Section 2.4.3 and the TeMS system of Section 7.1.2 are specialised to domains. Most of the techniques editors of section 2.4.1 are focused

on particular tasks or target groups of designers. The LSS system acknowledges this by having task-specific tools (such as the process editor of Section 5.2). The HANSEL system is, within the limits of its early specification language, domain and task independent in the early stages of design and, even in the later stages of skeleton application, is specific to particular forms of logic program design rather than to concepts which are directly meaningful in a target domain. This makes it unlikely that HANSEL could be applied in domains of application without some form of tuning to those domains, as has been the case for the other systems mentioned above.

The ideal way of adapting HANSEL to a domain would be by adapting the libraries of initial templates, rewrite rules and skeletons, while leaving the overall framework for applying these intact. Technically, this is straightforward because the libraries are modular components of the system and the rest of the framework is general for libraries of these sorts. Instead of the generic initial templates currently used in HANSEL we might provide more specific ones, in which more design decisions had been made (in effect, starting further down the refinement chain). Instead of general rewite rules we might provide more specific ones. Instead of the abstract forms of skeleton in the current system we might supply a library of domain-specific skeletons. The main threat to this ideal is whether the general framework (which governs the overall method of design) would really transfer without change to a new domain if given appropriately adapted libraries. We do not know if this would happen, and evidence from other systems is inconclusive. Optimistic news comes from systems such as Amphion which have been used in different domains but in Amphion the development of a problem description is done in a separate phase from the construction of a program for that description. In HANSEL the designer interacts all the way through the construction of a specification, since the design steps are not sufficiently constrained by domain information to be performed automatically. The TeMS system was an attempt to build a domain-specific synthesiser based on techniques editing. It brings more pessimistic news, since a considerale part of the effort in developing TeMS was in building user interface and problem description mechanisms specific to its target domain. Without building experimental systems it is difficult to predict whether a domain-specific version of HANSELwould be more like Amphion or TeMS.

Bibliography

[Agusti et al., 1998] Agusti, J., Puigsegur, J., and Robertson, D. (1998). A visual syntax for logic and logic programming. *Journal of Visual Languages and Computing*, 9.

[Angle et al., 1998] Angle, J., Fensel, D., Landes, D., and Studer, R. (1998). Developing knowledge-based systems with MIKE. *Journal of Automated Software Engineering*, 5(3):389–418.

[Barstow et al., 1982] Barstow, D., Duffy, R., Smoliar, S., and Vestal, S. (1982). An overview of phinix. In *National Conference on Artificial Intelligence*, Pittsburgh, Pennsylvania. AAAI.

[Bowles and Brna, 1993] Bowles, A. and Brna, P. (1993). Programming plans and programming techniques. In *Artificial Intelligence in Education, 1993: Proceedings of AI-ED 93*, pages 378–385, Edinburgh, Scotland. Virginia: AACE.

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

[Bundy et al., 1979] Bundy, A., Byrd, L., Luger, G., Mellish, C., Milne, R., and Palmer, M. (1979). Solving mechanics problems using meta-level inference. In Buchanan, B., editor, *Proceedings of IJCAI-79*, pages 1017–1027. International Joint Conference on Artificial Intelligence. Reprinted in 'Expert Systems in the microelectronic age' ed. Michie, D., Edinburgh University Press, 1979. Also available from Edinburgh as DAI Research Paper No. 112.

[Bundy et al., 1991] Bundy, A., Grosse, G., and Brna, P. (1991). A recursive techniques editor for Prolog. *Instructional Science*, 20:135–172.

[Castro, 1999] Castro, A. (1999). *A Techniques-based Framework for Domain-specific Synthesis of Simulation Models*. PhD thesis, University of Edinburgh.

[Cleland and MacKenzie, 1995] Cleland, G. and MacKenzie, D. (1995). Inhibiting factors, market structure and the industrial uptake of formal methods.

In *Workshop on Industrial-Strength Formal Specification Techniques*, pages 46–60, Boca Raton, Florida.

[Conklin and Begeman, 1988] Conklin, J. and Begeman, M. (1988). gIBIS: A hypertext tool for exploratory policy discussion. *ACM Transactions on Office Information Systems*, 6(4):303–331.

[Deville and Lau, 1994] Deville, Y. and Lau, K. (1994). Logic program synthesis. *Journal of Logic Programming*, 19(20).

[Eisenstadt and Brayshaw, 1988] Eisenstadt, M. and Brayshaw, M. (1988). The transparent prolog machine (tpm): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342.

[Fuchs and Fromherz, 1994] Fuchs, N. and Fromherz, M. (1994). Transformational development of logic programs from executable specifications - schema-based visual and textual composition of logic programs. In Beckstein, C. and Geske, K., editors, *Workshop on Development Test and Maintenance of Declarative AI Programs, GMD Studien Nr. 238, Gesellschaft fur Informatik und Datenverarbeitung,*, pages 13–28.

[Fuchs and Robertson, 1996] Fuchs, N. and Robertson, D. (1996). Declarative specification. *Knowledge Engineering Review (special issue on Logic Engineering)*, 11(4):317–331. ISSN 1361-0244.

[Gegg-Harrison, 1989] Gegg-Harrison, T. (1989). Basic Prolog schemata. Technical Report CS-1989-20, Department of Computer Science, Duke University.

[Gegg-Harrison, 1991] Gegg-Harrison, T. (1991). Learning Prolog in a schema-based environment. *Instructional Science*, 20(2/3):173–192.

[Goguen, 1989] Goguen, J. (1989). Principles of parameterised programming. In Biggerstaff, A. and Perlis, A., editors, *Software Reusability Volume 1: Concepts and Models*, pages 159–225. Addison Wesley.

[Gorlick et al., 1990] Gorlick, M., Kesselman, C., Marotta, D., and Parker, D. (1990). Mockingbird: A logical methodology for testing. *Journal of Logic Programming*, 8:95–119.

[Heitmeyer et al., 1996] Heitmeyer, C., Djeffords, R., and Labow, B. (1996). Automated consistency checking for requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261.

[Kalfoglou and Robertson, 1999] Kalfoglou, Y. and Robertson, D. (1999). Use of formal ontologies to support error checking in specifications. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management (EKAW-99), Germany*, pages 207–221. Springer Verlag (Lecture Notes in Computer Science 1621).

[Kirschenbaum et al., 1989] Kirschenbaum, M., Lakhotia, A., and Sterling, L. (1989). Skeletons and techniques for Prolog programming. Tr 89-170, Case Western Reserve University.

[Le Charlier et al., 1999] Le Charlier, B., Leclere, C., Rossi, S., and Cortesi, A. (1999). Automated verification of prolog programs. *Journal of Logic Programming*, 39:3–342.

[Levy, 1994] Levy, J. (1994). *The Calculus of Refinements, a Formal Specification Model Based on Inclusions.* PhD thesis, Departament de Llenguatges i Sistemes Informatics, Universitat Politecnica de Catalunya.

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

[Lloyd, 1985] Lloyd, J. (1985). *Foundations of Logic Programming.* Springer Verlag. ISBN 0-387-13299-6.

[Lowry et al., 1994] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I. (1994). A formal approach to domain-oriented software design environments. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference, Monterey, California*, pages 48–57.

[Lowry and Van Baalen, 1997] Lowry, M. and Van Baalen, J. (1997). Meta-amphion: Synthesis of efficient domain-specific program synthesis systems. *Automated Software Engineering*, 4:199–241.

[Naish and Sterling, 1997] Naish, L. and Sterling, L. (1997). A higher order reconstruction of stepwise enhancement. Technical Report 97/8, Department of Computer Science, University of Melbourne.

[Ormerod and Ball, 1996] Ormerod, T. and Ball, L. (1996). An empirical evaluation of TEd, a techniques editor for Prolog programming. In Gray, W. and Boehm-Davis, D., editors, *Empirical Studies of Programmers 6.* Ablex Publishing Corporation.

[Perry, 1995] Perry, W. (1995). *Effective Methods for Software Testing.* John Wiley & Sons. ISBN 0-471-06097-6.

[Plummer, 1990] Plummer, D. (1990). Cliche programming in Prolog. In *Proceedings of the META-90 workshop*, Leuven, Belgium. META-90.

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

[Robertson, 1991] Robertson, D. (1991). A simple Prolog techniques editor for novice users. In Wiggins, G., Mellish, C., and Duncan, T., editors, *Proceedings of 3rd Annual Conference on Logic Programming*, pages 190–205, Edinburgh. Springer-Verlag Workshops in Computing Series.

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

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

[Robertson, 1996b] Robertson, D. (1996b). Domain specific problem description. In *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering, Nevada, USA*. Knowledge Systems Institute, Illinois. ISBN 0-9641699-3-2.

[Robertson, 1998a] Robertson, D. (1998a). An empirical study of the LSS specification toolkit in use. *Journal of Systems and Software*, 42:115–123. one of the selected papers from SEKE-96.

[Robertson, 1998b] Robertson, D. (1998b). Pitfalls of formality in early system design. In *Proceedings of the ARO/NSF Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development*, Monterey, California. to appear in a special issue of The Science of Computer Programming.

[Robertson and Agusti, 1998] Robertson, D. and Agusti, J. (1998). Pragmatics in the synthesis of logic programs. In Flener, P., editor, *Logic-Based Program Synthesis and Transformation: 8th International Workshop, Manchester, UK (Selected papers)*, pages 41–60. Springer-Verlag, Lecture Notes in Computer Science 1559. ISBN 3-540-65765-7.

[Robertson and Agusti, 1999] Robertson, D. and Agusti, J. (1999). *Software Blueprints: Lightweight Uses of Logic in Conceptual Modelling*. Addison Wesley/ACM Press. ISBN 0201398192.

[Robertson et al., 1994] Robertson, D., Agusti, J., Hesketh, J., and Levy, J. (1994). Expressing program requirements using refinement lattices. *Fundamenta Informaticae*, 21(3):163–183. Longer version of one of five papers selected from proceedings of ISMIS-93.

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

[Sannella and Tarlecki, ming] Sannella, D. and Tarlecki, A. (forthcoming). Algebraic methods for specification and formal development of programs. *ACM Computing Surveys*.

[Schreiber et al., 1993] Schreiber, G., Wielinga, B., and Breuker, J. (1993). *KADS: A Principled Approach to Knowledge-Based System Development*. Academic Press. ISBN 0-12-629040-7.

[Stasko et al., 1998] Stasko, J., Domingue, J., Brown, M., and Price, B. (1998). *Software Visualisation: Programming as a Multimedia Experience*. MIT Press. ISBN 0-262-19395-7.

[Sterling and Kirschenbaum, 1993] Sterling, L. and Kirschenbaum, M. (1993). Applying techniques to skeletons. In Jacquet, J., editor, *Constructing Logic Programs*, pages 127–140. Wiley.

[Sterling and Shapiro, 1986] Sterling, L. and Shapiro, E. (1986). *The Art of Prolog: Advanced Programming Techniques*. MIT Press. ISBN 0-262-119250-0.

[Tansley and Hayball, 1993] Tansley, D. and Hayball, C. (1993). *A KADS Developer's Handbook*. Prentice Hall. ISBN 0-13-515479-0.

[Uschold, 1990] Uschold, M. (PhD Thesis: Submitted July 1990). *The Use of Types Lambda Calculus for Comprehension and Construction of Simulation Models in the Domain of Ecology*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

[Uschold and Gruninger, 1996] Uschold, M. and Gruninger, M. (1996). Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 11(2):93–136.

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

[Whittle, 1998] Whittle, J. (1998). *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh. PhD Thesis.

[Wielinga et al., 1992a] Wielinga, B., Schreiber, A., and Breuker, J. (1992a). Kads: A modelling approach to knowledge engineering. *Knowledge Acquisition*, 4(1):5–53. Reprinted in Buchanan, B and Wilkins, D. (eds) 1992, Readings in Knowledge Acquisition and Learning, pp 92-116, Morgan Kaufmann.

[Wielinga et al., 1992b] Wielinga, B. J., Schreiber, A. T., and Breuker, J. A. (1992b). KADS: A modelling approach to knowledge engineering. *Knowledge Acquistion Journal*, 4(1):5–53. Special issue 'The KADS approach to

knowledge engineering'. Reprinted in: Buchanan, B. and Wilkins, D. editors (1992), *Readings in Knowledge Acquisition and Learning*, San Mateo, California, Morgan Kaufmann, pp. 92-116.

Appendix A

HANSEL Syntax

The syntax of HANSEL specifications is given below. Essentially, it is standard Horn clause syntax without disjunction (for simplicity) and allowing only variable names in argument positions (to avoid committment to data structures for axiom sets). A number of special forms of subgoal are introduced as refinement terms.

- A HANSEL specification is a set of Horn clauses as defined below.
- A Horn clause is either a single unit term or is of the form $H \leftarrow B$, where H is a unit term and B is either a unit term; a refinement term; or a conjunction of unit and refinement terms.
- A conjunction is written $C_1 \wedge \ldots \wedge C_n$, where each C is a unit term.
- A unit term is written $P(A_1, \ldots, A_n)$, where P is an atom and each A is a variable name.
- Variable names are words beginning with an upper case character.
- Atoms are words beginning with a lower case character.
- A refinement term is one of the following, where X, $S1$ and $S2$ are variable names:
  - A general relation, written $S1 \approx S2$.
  - A specialisation relation, written $S1 \supseteq S2$.
  - A generalisation relation, written $S1 \subseteq S2$.
  - A relation goal, written $\overset{S1}{\rightsquigarrow} S2$.
  - A selection relation, written $\mathcal{S}(X, S1, ,)S2$.
  - An addition relation, written $\mathcal{A}(X, S1, ,)S2$.
  - A test, written $\mathcal{T}(S1)$.

All variables in HANSEL specifications, with the exception of those appearing in the first argument of selection and addition relations, refer to axiom sets. The data structure used to contain an axiom set is not prescribed by HANSEL allowing different data structures to be used according to taste, although the most likely is a Prolog list. HANSEL does prescribe the syntax of axioms within axiom sets. This is given below. Essentially, it is standard Horn clause syntax without disjunction (for simplicity) and without meta-logical predicates and negation (to preserve monotonicity when refining axiom sets).

- An axiom within a HANSEL axiom set must be of the form $H \leftarrow B$, where H is a unit goal and B is either a unit goal or a conjunction of unit goals.
- A conjunction is written $C_1 \wedge \ldots \wedge C_n$, where each C is a unit goal.
- A unit goal is one of the following:
  - The atom *true*.
  - A term of the form $P(A_1, \ldots, A_n)$, where P is an atom and each A is an argument term.
- An argument term is one of the following:
  - An atom.
  - A variable name.
  - A term of the form $P(A_1, \ldots, A_n)$, where P is an atom and each A is an argument term.

Appendix B

Current Set of LSS Skeletons

This chapter contains the library of skeletons currently used in the LSS techniques editor. When selected in the editing tool the variable P in the definitions below is instantiated to the given predicate name. The functor name F, which appears in expressions such as $F(Left, Right)$ in skeleton 4 is instantiated in the editor. In particular, it can be instantiated to the '.' operator used in Prolog as the functor for internal representation of lists, giving $.(Left, Right)$ which is equivalent in Prolog to $[Left|Right]$. This means that many of the skeletons in Section B.2 subsume those of Section B.3.

B.1 Basic Skeletons

LSS Skeleton 1 *Basic fact.*

$$P(X) \tag{B.1}$$

LSS Skeleton 2 *Basic rule.*

$$P(X) \leftarrow \mathcal{T}(X) \tag{B.2}$$

B.2 General Forms of Recursion

LSS Skeleton 3 *Recursion via tests and update.*

$$P(X) \leftarrow \mathcal{T}(X) \tag{B.3}$$
$$P(X) \leftarrow \mathcal{U}(X, Y) \wedge P(Y) \tag{B.4}$$

LSS Skeleton 4 *Recursion on the right of a binary term.*

$$P(X) \leftarrow \mathcal{T}_1(X) \tag{B.5}$$
$$P(F(Left, Right)) \leftarrow \mathcal{T}_2(Left) \wedge P(Right) \tag{B.6}$$

LSS Skeleton 5 *Recursion on the left of a binary term.*

$$P(X) \leftarrow \mathcal{T}_1(X) \quad \text{(B.7)}$$
$$P(F(Left, Right)) \leftarrow \mathcal{T}_2(Right) \wedge P(Left) \quad \text{(B.8)}$$

LSS Skeleton 6 *Double recursion on a binary term.*

$$P(X) \leftarrow \mathcal{T}(X) \quad \text{(B.9)}$$
$$P(F(Left, Right)) \leftarrow P(Left) \wedge P(Right) \quad \text{(B.10)}$$

LSS Skeleton 7 *Selective recursion on a pair of arguments.*

$$P(X, Y) \leftarrow \mathcal{T}_1(X) \wedge \mathcal{T}(Y) \quad \text{(B.11)}$$
$$P(F(A, B), Term) \leftarrow \mathcal{T}_2(A) \wedge P(B, Term) \quad \text{(B.12)}$$
$$P(Term, F(A, B)) \leftarrow \mathcal{T}_3(A) \wedge P(Term, B) \quad \text{(B.13)}$$

LSS Skeleton 8 *And-or decomposition of term.*

$$P(X) \leftarrow \mathcal{T}(X) \quad \text{(B.14)}$$
$$P(F(Left, Right)) \leftarrow P(Left) \quad \text{(B.15)}$$
$$P(F(Left, Right)) \leftarrow P(Right) \quad \text{(B.16)}$$
$$P(F(Left, Right)) \leftarrow P(Left) \wedge P(Right) \quad \text{(B.17)}$$

LSS Skeleton 9 *A vanilla meta-interpreter.*

$$P(X) \leftarrow \mathcal{T}(X) \quad \text{(B.18)}$$
$$P(or(Left, Right)) \leftarrow P(Left) \quad \text{(B.19)}$$
$$P(or(Left, Right)) \leftarrow P(Right) \quad \text{(B.20)}$$
$$P(and(Left, Right)) \leftarrow P(Left) \wedge P(Right) \quad \text{(B.21)}$$
$$P(Goal) \leftarrow \mathcal{U}(Goal, SubGoal) \wedge P(SubGoal) \quad \text{(B.22)}$$

B.3 Recursion on Lists

LSS Skeleton 10 *List search.*

$$P(X) \leftarrow \mathcal{T}_1(X) \quad \text{(B.23)}$$
$$P([H|T]) \leftarrow \mathcal{T}_2(H) \wedge P(T) \quad \text{(B.24)}$$

LSS Skeleton 11 *List traversal.*

$$P([]) \quad \text{(B.25)}$$
$$P([H|T]) \leftarrow \mathcal{T}(H) \wedge P(T) \quad \text{(B.26)}$$

LSS Skeleton 12 *List search and traversal.*

$$P([]) \tag{B.27}$$
$$P(X) \leftarrow \mathcal{T}_1(X) \tag{B.28}$$
$$P([H|T]) \leftarrow \mathcal{T}_2(H) \wedge P(T) \tag{B.29}$$

LSS Skeleton 13 *List double recursion.*

$$P(X) \leftarrow \mathcal{T}(X) \tag{B.30}$$
$$P([H|T]) \leftarrow P(H) \wedge P(T) \tag{B.31}$$

LSS Skeleton 14 *Selective recursion on a pair of lists.*

$$P(X,Y) \leftarrow \mathcal{T}_1(X) \wedge \mathcal{T}(Y) \tag{B.32}$$
$$P([H|T],Term) \leftarrow \mathcal{T}_2(H) \wedge P(T,Term) \tag{B.33}$$
$$P(Term,[H|T]) \leftarrow \mathcal{T}_3(H) \wedge P(Term,T) \tag{B.34}$$

LSS Skeleton 15 *And-or decomposition of a list.*

$$P(X) \leftarrow \mathcal{T}(X) \tag{B.35}$$
$$P([H|T]) \leftarrow P(H) \tag{B.36}$$
$$P([H|T]) \leftarrow P(T) \tag{B.37}$$
$$P([H|T]) \leftarrow P(H) \wedge P(T) \tag{B.38}$$

LSS Skeleton 16 *Mapping all elements of a list.*

$$P([],[]) \tag{B.39}$$
$$P([Ha|Ta],[Hb|Tb]) \leftarrow \mathcal{U}(Ha,Hb) \wedge P(Ta,Tb) \tag{B.40}$$

LSS Skeleton 17 *Mapping some elements of a list.*

$$P([],[]) \tag{B.41}$$
$$P([Ha|Ta],[Hb|Tb]) \leftarrow \mathcal{U}(Ha,Hb) \wedge P(Ta,Tb) \tag{B.42}$$
$$P([Ha|Ta],List) \leftarrow \mathcal{T}(Ha) \wedge P(Ta,List) \tag{B.43}$$

LSS Skeleton 18 *List binary split.*

$$P([]) \tag{B.44}$$
$$P([H|T]) \leftarrow \mathcal{T}_1(H) \wedge P(T) \tag{B.45}$$
$$P([H|T]) \leftarrow \mathcal{T}_2(H) \wedge P(T) \tag{B.46}$$

B.4 Counters

LSS Skeleton 19 *Generic counter.*

$$P(X) \leftarrow \mathcal{T}(X) \tag{B.47}$$
$$P(X) \leftarrow Y \text{ is } F(X,N) \wedge P(Y) \tag{B.48}$$

LSS Skeleton 20 *Decrementing counter.*

$$P(0) \tag{B.49}$$
$$P(X) \leftarrow Y \; is \; X - 1 \wedge P(Y) \tag{B.50}$$

LSS Skeleton 21 *Incrementing counter.*

$$P(X, X) \tag{B.51}$$
$$P(X, Y) \leftarrow X < Y \wedge Z \; is \; X + 1 \wedge P(Z, Y) \tag{B.52}$$

Appendix C

Current Set of HANSEL Skeletons

C.1 Traversal Skeletons

Skeleton 1

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}(X1,S1,T1) \wedge \\
 & R(X1,X2) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}(X2,T2,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\exists X2.R(X1,X2) \wedge X2 \in S2 \right) \\
X2 \in S2 \rightarrow \left(\exists X1.R(X1,X2) \wedge X1 \in S1 \right)
\end{array} \right\}
\end{array}
$$

Skeleton 2

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1, X2) \wedge \\
 & P(T1, T2) \wedge \\
 & \mathcal{A}_1(X2, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1, X2)) \wedge \\
 & P(T1, T2) \wedge \\
 & \mathcal{A}_2(X1, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2 \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \ \wedge \ X1 \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 3

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1, X2), \mathit{Set}), P(T1, T2) \wedge \\
 & \mathit{union}(\mathit{Set}, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2 \\ \wedge \\ \forall X2.R(X1,X2) \rightarrow X2 \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 4

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1,X2), \mathit{Set}) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathit{union}(\mathit{Set}, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}(X1, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2) \\ \wedge \\ \forall X2'.R(X1,X2') \rightarrow X2' \in S2 \end{array} \right) \\ \vee \\ \mathit{not}(\exists X2''.R(X1,X2'')) \ \wedge \ X1 \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 5

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}(X1, S1, T1) \wedge \\
 & R(X1,X2) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}_1(X1, T2, S3) \wedge \\
 & \mathcal{A}_2(X2, S3, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\ \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \ \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 6

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1, X2) \wedge \\
 & P(T1, T2) \wedge \\
 & \mathcal{A}_1(X1, T2, S3) \wedge \\
 & \mathcal{A}_2(X2, S3, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1, X2)) \wedge \\
 & P(T1, T2) \wedge \\
 & \mathcal{A}_3(X1, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \ \wedge \ X1 \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 7

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1, X2), \mathit{Set}) \wedge \\
 & P(T1, T2) \wedge \\
 & \mathcal{A}(X1, \mathit{Set}, S3) \wedge \\
 & \mathit{union}(S3, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \\ \wedge \\ \forall X2'.R(X1,X2') \rightarrow X2' \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 8

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1,X2), \mathit{Set}), P(T1,T2) \wedge \\
 & \mathcal{A}_1(X1, \mathit{Set}, S3) \wedge \\
 & \mathit{union}(S3, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}_2(X1, T2, S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \\ \wedge \\ \forall X2'.R(X1,X2') \rightarrow X2' \in S2 \end{array} \right) \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \ \wedge \ X1 \in S2 \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 9

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1,X2) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}(X2, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2 \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \end{array} \right) \\
X2 \in S2 \rightarrow \left(\ \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \right)
\end{array} \right\}
\end{array}
$$

Skeleton 10

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1,X2), \mathit{Set}) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathit{union}(\mathit{Set}, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X2 \in S2 \\ \wedge \\ \forall X2'.R(X1,X2') \rightarrow X2' \in S2 \end{array} \right) \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \end{array} \right) \\
X2 \in S2 \rightarrow \left(\ \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \right)
\end{array} \right\}
\end{array}
$$

Skeleton 11

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1,X2) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}_1(X1, T2, S3) \wedge \\
 & \mathcal{A}_2(X2, S3, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 12

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & \mathit{setof}(X2, R(X1,X2), \mathit{Set}) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}(X1, \mathit{Set}, S3) \wedge \\
 & \mathit{union}(S3, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1,X2)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} \left(\begin{array}{c} \exists X2.R(X1,X2) \ \wedge \ X1 \in S2 \ \wedge \ X2 \in S2 \\ \wedge \\ \forall X2'.R(X1,X2') \rightarrow X2' \in S2 \end{array} \right) \\ \vee \\ \mathit{not}(\exists X2.R(X1,X2)) \end{array} \right) \\
X2 \in S2 \rightarrow \left(\begin{array}{c} \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \ \wedge \ X1 \in S2 \\ \vee \\ \mathit{not}(\exists X1.R(X1,X2)) \ \wedge \ X2 \in S1) \end{array} \right)
\end{array} \right\}
\end{array}
$$

Skeleton 13

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1, S1, T1) \wedge \\
 & R(X1) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathcal{A}(X1, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1, S1, T1) \wedge \\
 & \mathit{not}(R(X1)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\ (R(X1) \ \wedge \ X1 \in S2) \ \vee \mathit{not}(R(X1)) \ \right) \\
X2 \in S2 \rightarrow \left(\ R(X1) \ \wedge \ X1 \in S1 \ \right)
\end{array} \right\}
\end{array}
$$

Skeleton 14

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathit{empty}(S1) \wedge \\
 & \mathit{empty}(S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\
 & \mathit{setof}(X1, R(X1), \mathit{Set}) \wedge \\
 & P(T1,T2) \wedge \\
 & \mathit{union}(\mathit{Set}, T2, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\
 & \mathit{not}(R(X1)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
X1 \in S1 \rightarrow \left(\begin{array}{c} (R(X1) \ \wedge \ X1 \in S2) \ \wedge \forall X.(X = X1 \ \wedge \ R(X)) \rightarrow X \in S2 \\ \vee \\ \mathit{not}(R(X1) \end{array} \right) \\
X2 \in S2 \rightarrow \left(\ YYY \ \right)
\end{array} \right\}
\end{array}
$$

C.2 Search Skeletons

Skeleton 15

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\
 & R(X1) \wedge \\
 & \mathit{singleton}(X1, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\
 & \mathit{not}(R(X1)) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
\exists X1.X1 \in S1 \ \wedge \ R(X1) \ \wedge \ \mathit{singleton}(X1, S2) \\
\forall X2.X2 \in S2 \rightarrow R(X2) \ \wedge \ X2 \in S1
\end{array} \right\}
\end{array}
$$

Skeleton 16

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll}
P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\
 & \mathit{not}(R(X1)) \wedge \\
 & \mathit{singleton}(X1, S2), \\
P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\
 & R(X1) \wedge \\
 & P(T1,S2)
\end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l}
\exists X1.X1 \in S1 \ \wedge \ \mathit{not}(R(X1)) \ \wedge \ \mathit{singleton}(X1, S2) \\
\forall X2.X2 \in S2 \rightarrow \mathit{not}(R(X2)) \ \wedge \ X2 \in S1
\end{array} \right\}
\end{array}
$$

Skeleton 17

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & R(X1,X2) \wedge \\ & \mathit{singleton}(X2,S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & \mathit{not}(R(X1,X2)) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1, X2.X1 \in S1 \ \wedge \ R(X1,X2) \ \wedge \ \mathit{singleton}(X2,S2) \\ \forall X2.X2 \in S2 \rightarrow \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 18

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & \mathit{setof}(X1, R(X1), S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & \mathit{not}(R(X1)) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1.X1 \in S1 \ \wedge \ \forall X1'.((X1' = X1 \ \wedge \ R(X1')) \rightarrow X1' \in S2) \\ \forall X2.X2 \in S2 \rightarrow R(X2) \ \wedge \ X2 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 19

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & \mathit{setof}(X2, R(X1,X2), S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & \mathit{not}(R(X1,X2)) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1.X1 \in S1 \ \wedge \ \forall X1'.((X1' = X1 \ \wedge \ \exists X2.R(X1',X2)) \rightarrow X2 \in S2) \\ \forall X2.X2 \in S2 \rightarrow \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 20

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R1, R2]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & R1(X1) \wedge \\ & \mathit{singleton}(X1,S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & R2(X1) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1.X1 \in S1 \ \wedge \ R(X1) \ \wedge \ \mathit{singleton}(X1,S2) \\ \forall X2.X2 \in S2 \rightarrow R(X2) \ \wedge \ X2 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 21

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R1, R2]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & R(X1,X2) \wedge \\ & \mathit{singleton}(X2,S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & R2(X1) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1, X2.X1 \in S1 \ \wedge \ R(X1,X2) \ \wedge \ \mathit{singleton}(X2,S2) \\ \forall X2.X2 \in S2 \rightarrow \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 22

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R1, R2]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & \mathit{setof}(X1, R(X1), S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & R2(X1) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1.X1 \in S1 \ \wedge \ \forall X1'.((X1' = X1 \ \wedge \ R(X1')) \rightarrow X1' \in S2) \\ \forall X2.X2 \in S2 \rightarrow R(X2) \ \wedge \ X2 \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 23

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R1, R2]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & \mathcal{S}_1(X1,S1,T1) \wedge \\ & \mathit{setof}(X2, R(X1,X2), S2), \\ P(S1,S2) \leftarrow & \mathcal{S}_2(X1,S1,T1) \wedge \\ & R2(X1) \wedge \\ & P(T1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X1.X1 \in S1 \ \wedge \ \forall X1'.((X1' = X1 \ \wedge \ \exists X2.R(X1',X2)) \rightarrow X2 \in S2) \\ \forall X2.X2 \in S2 \rightarrow \exists X1.R(X1,X2) \ \wedge \ X1 \in S1 \end{array} \right\}
\end{array}
$$

C.3 Deduction Skeletons

Skeleton 24

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, D]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,G) \wedge \\ & \mathit{deduce}(D, S1 \vdash G) \wedge \\ & \mathit{singleton}(G,S3) \wedge \\ & \mathit{union}(S3,S1,S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \forall X1.X1 \in S1 \rightarrow X1 \in S2 \\ \exists G.R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G) \\ \forall G.G \in S2 \rightarrow G \in S1 \vee (R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G)) \end{array} \right\}
\end{array}
$$

Skeleton 25

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, D]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,G) \wedge \\ & \mathit{setof}(G, \mathit{deduce}(D, S1 \vdash G), S3) \wedge \\ & \mathit{union}(S3, S1, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \forall X1.X1 \in S1 \rightarrow X1 \in S2 \\ \exists G.R(S1,G) \ \wedge \ \forall X.(X = G \ \wedge \ \mathit{deduce}(D, S1 \vdash G) \rightarrow X \in S2) \\ \forall G.G \in S2 \rightarrow G \in S1 \vee (R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G)) \end{array} \right\}
\end{array}
$$

Skeleton 26

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, D]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,G) \wedge \\ & \mathit{deduce}(D, S1 \vdash G) \wedge \\ & \mathit{singleton}(G, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists G.R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G) \ \wedge \ \mathit{singleton}(G, S2) \\ \forall G.G \in S2 \rightarrow (R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G)) \end{array} \right\}
\end{array}
$$

Skeleton 27

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, D]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,G) \wedge \\ & \mathit{setof}(G, \mathit{deduce}(D, S1 \vdash G), S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists G.R(S1,G) \ \wedge \ \forall X.((X = G \ \wedge \ \mathit{deduce}(D, S1 \vdash G)) \rightarrow X \in S2) \\ \forall G.G \in S2 \rightarrow (R(S1,G) \ \wedge \ \mathit{deduce}(D, S1 \vdash G)) \end{array} \right\}
\end{array}
$$

C.4 Abduction Skeletons

Skeleton 28

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, A]) \\
\mathit{code} & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,G) \wedge \\ & \mathit{abduce}(A, S1 \vdash G, Ae) \wedge \\ & \mathit{union}(Ae, S1, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \forall X1.X1 \in S1 \rightarrow X1 \in S2 \\ \exists G.R(S1,G) \ \wedge \ \mathit{abduce}(A, S1 \vdash G, Ae) \ \wedge \ \forall X.(X \in Ae \rightarrow X \in S2) \\ \forall X2.X2 \in S2 \rightarrow \left(\begin{array}{l} X2 \in S1 \vee \\ \exists G, Ae.(R(S1,G) \ \wedge \ \mathit{abduce}(D, S1 \vdash G, Ae) \ \wedge X2 \in Ae) \end{array} \right) \end{array} \right\}
\end{array}
$$

Skeleton 29

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, A]) \\
\mathit{code} & : & \left\{ P(S1,S2) \leftarrow \begin{array}{l} R(S1,G) \wedge \\ \mathit{setof}(G, \mathit{abduce}(A, S1 \vdash G, Ae), S3) \wedge \\ \mathit{union}(S3, S1, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \forall X1.X1 \in S1 \rightarrow X1 \in S2 \\ \exists G.R(S1,G) \ \wedge \ \forall X.(X = G \ \wedge \ \mathit{abduce}(D, S1 \vdash G, Ae) \rightarrow (X' \in Ae \rightarrow X' \in S2)) \\ \forall X2.X2 \in S2 \rightarrow \left(\begin{array}{l} X2 \in S1 \vee \\ \exists G, Ae.(R(S1,G) \ \wedge \ \mathit{abduce}(D, S1 \vdash G, Ae) \ \wedge X2 \in Ae) \end{array} \right) \end{array} \right\}
\end{array}
$$

Skeleton 30

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, A]) \\
\mathit{code} & : & \left\{ P(S1,S2) \leftarrow \begin{array}{l} R(S1,G) \wedge \\ \mathit{abduce}(A, S1 \vdash G, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists G.R(S1,G) \ \wedge \ \mathit{abduce}(D, S1 \vdash G, S2) \\ \forall X2.X2 \in S2 \rightarrow \exists G.(R(S1,G) \ \wedge \ \mathit{abduce}(D, S1 \vdash G, Ae) \ \wedge X2 \in Ae) \end{array} \right\}
\end{array}
$$

Skeleton 31

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R, A]) \\
\mathit{code} & : & \left\{ P(S1,S2) \leftarrow \begin{array}{l} R(S1,G) \wedge \\ \mathit{setof}(G, \mathit{abduce}(A, S1 \vdash G, Ae), S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists G.R(S1,G) \ \wedge \ \forall X, Xe. \left(\begin{array}{c} X = G \wedge \\ \mathit{abduce}(D, S1 \vdash G, Ae) \wedge \\ Xe \in Ae \end{array} \right) \rightarrow Xe \in S2 \\ \forall X.X \in S2 \rightarrow \exists G.(R(S1,G) \ \wedge \ \mathit{abduce}(D, S1 \vdash G, Ae) \ \wedge \ X \in Ae) \end{array} \right\}
\end{array}
$$

C.5 General Relation Skeletons

Skeleton 32

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \left\{ P(S1,S2) \leftarrow \begin{array}{l} R(S1,X) \wedge \\ \mathit{singleton}(X, S2) \end{array} \right\} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X.R(S1,X) \ \wedge \ \mathit{singleton}(X, S2) \\ \forall X.X \in S2 \rightarrow R(S1,X) \end{array} \right\}
\end{array}
$$

Skeleton 33

$$
\begin{array}{rcl}
\mathit{parameters} & : & (P, S1, S2, [R]) \\
\mathit{code} & : & \{ P(S1,S2) \leftarrow \mathit{setof}(X, R(S1,X), S2) \} \\
\mathit{properties} & : & \left\{ \begin{array}{l} \exists X.R(S1,X) \ \wedge \ X \in S2 \\ \forall X.R(S1,X) \rightarrow X \in S2 \\ \forall X.X \in S2 \rightarrow R(S1,X) \end{array} \right\}
\end{array}
$$

Skeleton 34

$$
\begin{array}{rcl}
parameters & : & (P, S1, S2, [R]) \\
code & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & R(S1,X) \wedge \\ & singleton(X,S3) \wedge \\ & union(S3,S1,S2) \end{array} \right\} \\
properties & : & \left\{ \begin{array}{l} \exists X.R(S1,X) \ \wedge \ X \in S2 \\ \forall X.X \in S1 \rightarrow X \in S2 \\ \forall X.X \in S2 \rightarrow R(S1,X) \ \vee \ X \in S1 \end{array} \right\}
\end{array}
$$

Skeleton 35

$$
\begin{array}{rcl}
parameters & : & (P, S1, S2, [R]) \\
code & : & \left\{ \begin{array}{ll} P(S1,S2) \leftarrow & setof(X, R(S1,X), S3) \wedge \\ & union(S3,S1,S2) \end{array} \right\} \\
properties & : & \left\{ \begin{array}{l} \exists X.R(S1,X) \ \wedge \ X \in S2 \\ \forall X.R(S1,X) \rightarrow X \in S2 \\ \forall X.X \in S1 \rightarrow X \in S2 \\ \forall X.X \in S2 \rightarrow R(S1,X) \ \vee \ X \in S1 \end{array} \right\}
\end{array}
$$

Appendix D

Prolog Goals for Example Properties

Corresponding to property 12.4:

$$not(CaseDescription \vdash X \ \wedge \ not(AbstractedCase \vdash X)) \tag{D.1}$$

Corresponding to property 12.9:

$$not(Norm \vdash X \ \wedge \ not(SystemModel \vdash X)) \tag{D.2}$$

Corresponding to property 12.11:

$$SystemModel \vdash X \ \wedge \ not(Norm \vdash X) \tag{D.3}$$

Corresponding to property 12.12:

$$AbstractedCase \vdash X \ \wedge \ not(CaseDescription \vdash X) \tag{D.4}$$

Corresponding to property 12.19:

$$not\left(\begin{array}{l} \\ not\left(\begin{array}{c} abstract(CaseDescription, AbstractedCase) \wedge \\ X1 \in CaseDescription \wedge \\ \left(\begin{array}{c} generalisation(X1, X2) \wedge \\ X1 \in AbstractedCase \wedge \\ X2 \in AbstractedCase \end{array}\right) \\ \wedge \\ not(generalisation(X1, X2') \ \wedge \ not(X2' \in AbstractedCase)) \end{array}\right) \\ \\ \end{array}\right) \tag{D.5}$$

Corresponding to property 12.20:

$$not\left(\begin{array}{c} abstract(CaseDescription, AbstractedCase) \wedge \\ X2 \in AbstractedCase \wedge \\ not\left(\begin{array}{c} \left(\begin{array}{c} generalisation(X1, X2) \wedge \\ X1 \in CaseDescription \wedge \\ X1 \in AbstractedCase \end{array}\right) \\ \vee \\ \left(\begin{array}{c} not(generalisation(X1, X2)) \wedge \\ X2 \in CaseDescription \end{array}\right) \end{array}\right) \end{array}\right) \tag{D.6}$$

Corresponding to property 12.21:

$$not\left(\begin{array}{c} select\_norms(SystemModel, Norm) \wedge \\ X1 \in SystemModel \wedge \\ not\left(\begin{array}{c} relevant(X1, X2) \ \wedge \ X2 \in Norm \\ \vee \\ not(relevant(X1, X2))) \end{array}\right) \end{array}\right) \tag{D.7}$$

Corresponding to property 12.22:

$$not\left(\begin{array}{c} select\_norms(SystemModel, Norm) \wedge \\ X2 \in Norm \wedge \\ not\left(\ relevant(X1, X2) \ \wedge \ X1 \in SystemModel \ \right) \end{array}\right) \tag{D.8}$$

Corresponding to property 12.51:

$$not(State \vdash X \ \wedge \ not(NextState \vdash X)) \tag{D.9}$$

Corresponding to property 12.52:

$$not(Predicted \vdash X \ \wedge \ not(State \vdash X)) \tag{D.10}$$

Corresponding to property 12.55:

$$State \vdash X \ \wedge \ not(Predicted \vdash X) \tag{D.11}$$

Corresponding to property 12.56:

$$X \in NextState \ \wedge \ not(X \in State) \tag{D.12}$$

Corresponding to property 12.60:

$$not\left(\begin{array}{c} envision(State, NextState) \wedge \\ X1 \in State \ \wedge \ not(X1 \in NextState) \end{array}\right) \tag{D.13}$$

Corresponding to property 12.61:

$$not\left(\begin{array}{c} envision(State, NextState) \wedge \\ not(attribute\_goal(State, G) \ \wedge \ not((X = G \ \wedge \ deduce(D, State \vdash G)) \ \wedge \ not(X \in NextState))) \end{array}\right) \tag{D.14}$$

Corresponding to property 12.62:

$$not\left(\begin{array}{c} envision(State, NextState) \wedge \\ not(G \in NextState \ \wedge \ not(G \in State \vee (attribute\_goal(State, G) \ \wedge \ deduce(D, State \vdash G)))) \end{array}\right) \tag{D.15}$$

Corresponding to property 12.69:

$$not\left(\begin{array}{c} final\_attributes(State, Predicted) \wedge \\ X1 \in State \ \wedge not\left(\begin{array}{c} is\_attribute(X1, X2) \ \wedge \ X2 \in Predicted \\ \vee \\ not(is\_attribute(X1, X2)) \end{array}\right) \end{array}\right) \tag{D.16}$$

Corresponding to property 12.70:

$$not\left(\begin{array}{c} final\_attributes(State, Predicted) \wedge \\ X2 \in Predicted \ \wedge \ not(\ is\_attribute(X1, X2) \ \wedge \ X1 \in State \ ) \end{array}\right) \tag{D.17}$$

Corresponding to property 12.81:

$$not(NewLandmarks \vdash X \ \wedge \ not(NextState \vdash X)) \tag{D.18}$$

Corresponding to property 12.82:

$$not(Landmarks \vdash X \ \wedge \ not(State \vdash X)) \tag{D.19}$$

Corresponding to property 12.85:

$$State \vdash X \ \wedge \ not(Landmarks \vdash X) \tag{D.20}$$

Corresponding to property 12.91:

$$not\left(\begin{array}{c} find\_landmarks(State, Landmarks) \wedge \\ X1 \in State \wedge \\ not\left(\begin{array}{c} is\_landmark(X1, X2) \ \wedge \ X2 \in Landmarks \\ \vee \\ not(is\_landmark(X1, X2)) \end{array}\right) \end{array}\right) \tag{D.21}$$

Corresponding to property 12.92:

$$not\left(\begin{array}{c} find\_landmarks(State, Landmarks) \wedge \\ X2 \in Landmarks \wedge \\ not(\ is\_landmark(X1, X2) \ \wedge \ X1 \in State \ ) \end{array}\right) \tag{D.22}$$

Corresponding to property 12.97:

$$not\left(\begin{array}{c} new\_landmarks(Landmarks, NewLandmarks) \wedge \\ X1 \in Landmarks \wedge \\ not(\ new\_landmark(X1, X2) \ \wedge \ X2 \in NewLandmarks \ ) \end{array}\right) \tag{D.23}$$

Corresponding to property 12.98:

$$not\left(\begin{array}{c} new\_landmarks(Landmarks, NewLandmarks) \wedge \\ X2 \in NewLandmarks \wedge \\ not(\ new\_landmark(X1, X2) \ \wedge \ X1 \in Landmarks \ ) \end{array}\right) \tag{D.24}$$

Corresponding to property 12.101:

$$not(NewLandmarks \vdash X1 \ \wedge \ not(NextState \vdash X1)) \tag{D.25}$$

Corresponding to property 12.102:

$$NextState \vdash X2 \ \wedge \ not(NewLandmarks \vdash X2) \tag{D.26}$$

Corresponding to property 12.107:

$$not\left(\begin{array}{c} replace\_landmarks(NewLandmarks, NextState) \wedge \\ X1 \in NewLandmarks \wedge \\ not\left(\ replace\_landmark(X1, X2) \ \wedge \ X2 \in NextState \ \right) \end{array}\right) \tag{D.27}$$

Corresponding to property 12.108:

$$not\left(\begin{array}{c} replace\_landmarks(NewLandmarks, NextState) \wedge \\ X2 \in NextState \wedge \\ not\left(\ replace\_landmark(X1, X2) \ \wedge \ X1 \in NewLandmarks \ \right) \end{array}\right) \tag{D.28}$$

Corresponding to property 12.117:

$$not(Influences \vdash X1 \ \wedge \ not(State \vdash X1)) \tag{D.29}$$

Corresponding to property 12.120:

$$State \vdash X2 \ \wedge \ not(Influences \vdash X2) \tag{D.30}$$

Corresponding to property 12.126:

$$not\left(\begin{array}{c} find\_influences(State, Influences) \wedge \\ X1 \in State \wedge \\ not\left(\begin{array}{c} is\_influence(X1, X2) \ \wedge \ X2 \in Influences \\ \vee \\ not(is\_influence(X1, X2)) \end{array}\right) \end{array}\right) \tag{D.31}$$

Corresponding to property 12.127:

$$not\left(\begin{array}{c} find\_influences(State, Influences) \wedge \\ X2 \in Influences \wedge \\ not\left(\ is\_influence(X1, X2) \ \wedge \ X1 \in State \ \right) \end{array}\right) \tag{D.32}$$

Corresponding to property 12.130:

$$not(Predicted \vdash X1 \ \wedge \ not(State \vdash X1)) \tag{D.33}$$

Corresponding to property 12.133:

$$not(Predicted \vdash X \ \wedge \ not(State \vdash X)) \tag{D.34}$$