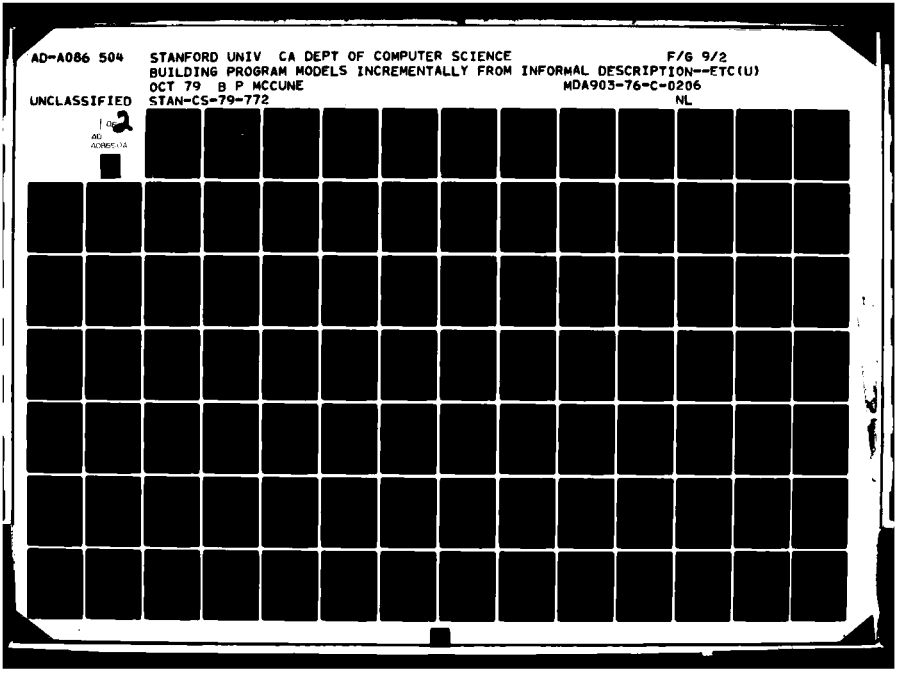

AD-A086 504
STANFORD UNIV CA DEPT OF COMPUTER SCIENCE
F/G 9/2
BUILDING PROGRAM MODELS INCREMENTALLY FROM INFORMAL DESCRIPTION--ETC(U)
OCT 79 B P MCCUNE
MDA903-76-C-0206
UNCLASSIFIED
STAN-CS-79-772
NL
2
AD
A086504

Stanford Artificial Intelligence Laboratory
Memo AIM-333

October 1979

Computer Science Department
Report No. STAN-CS-79-772

Systems Control, Inc.
Technical Report SCI.ICS.U.79.2

Building Program Models Incrementally from Informal Descriptions

by

Brian P. McCune

Research sponsored by

Defense Advanced Research Projects Agency

COMPUTER SCIENCE DEPARTMENT
Stanford University

and

Computer Science Department
Systems Control, Inc.
Palo Alto, California

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(14) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: AIM-333 (STAN-CS-79-772, AIM-333) 2. GOVT ACCESSION NO.: AD-A086504	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): (6) Building Program Models Incrementally from Informal Descriptions.	5. TYPE OF REPORT & PERIOD COVERED: technical, October 1979
	6. PERFORMING ORG. REPORT NUMBER: AIM-333 (STAN-CS-79-772)
7. AUTHOR(s): (10) Brian P. McCune	8. CONTRACT OR GRANT NUMBER(s): (15) N00014-79-C-0127, MDA903-76-C-0206
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, Stanford University, Stanford, California 94305 USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: (12) 144
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, Information Processing Techniques Office, 1400 Wilson Avenue, Arlington, Virginia 22209 (11)	12. REPORT DATE: Oct [redacted]79 — 13. NO. OF PAGES: 140
	15. SECURITY CLASS. (of this report): Unclassified
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office): Mr. Philip Surra, Resident Representative, Office of Naval Research, Durand 165, Stanford University	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

(9) Technical rep.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see other side)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

094 120

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

Program acquisition is the transformation of a program specification into an executable, but not necessarily efficient, program that meets the given specification. This thesis presents a solution to one aspect of the program acquisition problem: the incremental construction of program models from informal descriptions. The key to the solution is a framework for incremental program acquisition that includes (1) a formal language for expressing program fragments that contain informalities, (2) a control structure for the incremental recognition and assimilation of such fragments, and (3) a knowledge base of rules for acquiring programs specified with informalities.

The thesis describes a LISP based computer system called the *Program Model Builder* (abbreviated PMB), which receives informal program fragments incrementally and assembles them into a very high level program model that is complete, semantically consistent, unambiguous, and executable. The program specification comes in the form of partial program fragments that arrive in any order and may exhibit such informalities as inconsistencies and ambiguous references. Possible sources of fragments are a natural language parser or a parser for a surface form of the fragments. PMB produces a program model that is a complete and executable computer program. The *program fragment language* used for specifications is a superset of the language in which program models are built. This *program modelling language* is a very high level programming language for symbolic processing that deals with such information structures as sets and mappings.

The recognition paradigm used by PMB is a form of subgoaling that allows the parts of the program to be specified in an order chosen by the user, rather than dictated by the system. Knowledge is represented as a set of data driven antecedent rules of two types, *response rules* and *demons*, which are triggered respectively by either the input of new fragments or changes in the partial program model. In processing a fragment, a response rule may update the partial program model and create new subgoals with associated response rules. To process subgoals that are completely internal to PMB (e.g., model consistency checks), demon rules are created that delay execution until their prerequisite information in the program model has been filled in by response rules or perhaps other demons.

PMB has been tested both as a module of the PSI program synthesis system and independently. Models built as part of PSI have been acquired via natural language dialogs and execution traces and have been automatically coded into LISP by other PSI modules. PMB has successfully built a number of moderately complex programs for symbolic computation.

Stanford Artificial Intelligence Laboratory
Memo AIM-333

October 1979

Computer Science Department
Report No. STAN-CS-79-772

Systems Control, Inc.
Technical Report SCLICS.U.79.2

Building Program Models Incrementally from Informal Descriptions

by

Brian P. McCune

ABSTRACT

Program acquisition is the transformation of a program specification into an executable, but not necessarily efficient, program that meets the given specification. This thesis presents a solution to one aspect of the program acquisition problem: the incremental construction of program models from informal descriptions. The key to the solution is a framework for incremental program acquisition that includes (1) a formal language for expressing program fragments that contain informalities, (2) a control structure for the incremental recognition and assimilation of such fragments, and (3) a knowledge base of rules for acquiring programs specified with informalities.

The thesis describes a LISP based computer system called the *Program Model Builder* (abbreviated "PMB"), which receives informal program fragments incrementally and assembles them into a very high level program model that is complete, semantically consistent, unambiguous, and executable. The program specification comes in the form of partial program fragments that arrive in any order and may exhibit such informalities as inconsistencies and ambiguous references. Possible sources of fragments are a natural language parser or a parser for a surface form of the fragments. PMB produces a program model that is a complete and executable computer program. The *program fragment language* used for specifications is a superset of the language in which program models are built. This *program modelling language* is a very high level programming language for symbolic processing that deals with such information structures as sets and mappings.

The recognition paradigm used by PMB is a form of subgoaling that allows the parts of the program to be specified in an order chosen by the user, rather than dictated by the system. Knowledge is represented as a set of data driven antecedent rules of two types, *response rules* and *demons*, which are triggered respectively by either the input of new fragments or changes in the

partial program model. In processing a fragment, a response rule may update the partial program model and create new subgoals with associated response rules. To process subgoals that are completely internal to PMB (e.g., model consistency checks), demon rules are created that delay execution until their prerequisite information in the program model has been filled in by response rules or perhaps other demons.

PMB has been tested both as a module of the PSI program synthesis system and independently. Models built as part of PSI have been acquired via natural language dialogs and execution traces and have been automatically coded into LISP by other PSI modules. PMB has successfully built a number of moderately complex programs for symbolic computation.

This dissertation was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Available from University Microfilms International, P. O. Box 1346, Ann Arbor, Michigan 48106

Acknowledgments

Professor Cordell Green provided his time, encouragement, and many good ideas throughout the period he was my thesis adviser and leader of the PSI project. The other members of my thesis reading committee, Professors Bruce G. Buchanan and Terry Winograd, spent many hours reading my thesis and discussing it with me in order to crystallize the ideas it contains, relate them to other work, and help me present them better. The fourth member of my oral examination committee, Professor Gio Wiederhold, helped teach me what a Ph.D. thesis is and how to present it to a wider audience than just the artificial intelligence community.

Many other people helped me in various ways during the writing of this thesis. Discussions with the knowledge based programming group at Systems Control, Inc., helped clarify the problems addressed by the Program Model Builder and how it relates to the PSI program synthesis system. Elaine Kant, Thomas T. Pressburger, Stephen J. Westfold, and Michael J. Clancy provided helpful criticisms of drafts of the thesis.

This work would not have been possible without the ideas, hacking, support, and friendship of the entire PSI group at the Stanford Artificial Intelligence Laboratory. Jerrold M. Ginsparg, Jorge V. Phillips, Louis I. Steinberg, Stephen J. Westfold, and Ronny van den Heuvel collaborated on the definition of the program fragment language. David R. Barstow created the initial specifications of the program modelling language. David R. Barstow, Elaine Kant, Bruce Nelson, and Juan J. Ludlow helped "debug" the modelling language. Bruce Nelson wrote the program model interpreter, and Richard E. Pattis enhanced it to parse arbitrary inputs. Thomas T. Pressburger wrote the program that generates readable program models. Avra Cohn and Ronny van den Heuvel outlined the types of programming knowledge a program acquisition system should have. Steve T. Tappel wrote and helped specify the rule expander.

Finally, thanks to Lester D. Earnest and the rest of the hacker/volleyball community at the Artificial Intelligence Laboratory for making it a fun and productive environment in which to work. And thanks to my housemates, friends, parents, and relatives for having faith in me and for putting up with much work and little play for too long.

This thesis describes research done in the Computer Science Department, Systems Control, Inc., and the Artificial Intelligence Laboratory, Stanford University. The research was supported in part by the Defense Advanced Research Projects Agency under DARPA Order 3687, Contract N00014-79-C-0127, which is monitored by the Office of Naval Research, and DARPA Order 2494, Contract MDA903-76-C-0206. Additional support was provided by the National Science Foundation through an NSF Graduate Fellowship, the International Business Machines Corporation through an IBM Graduate Fellowship, and the Josephine de Karman Fellowship Trust through a Josephine de Karman Fellowship. I am extremely grateful to all of these organizations for their financial support.

The views and conclusions contained in this thesis are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of any of the organizations mentioned above.

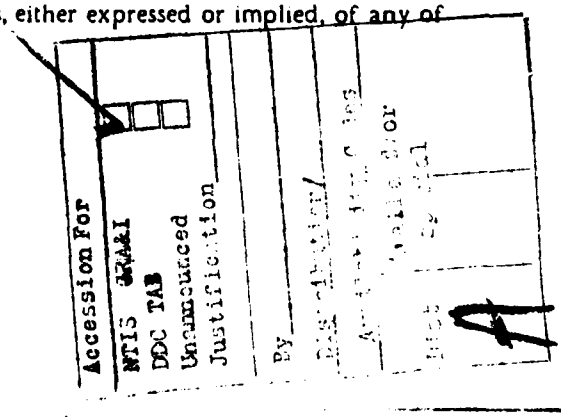


Table of Contents

Table of Contents

Table of Contents

Chapter 1. Introduction

Most of the research that has been done under the rubric of "automatic programming" during the past decade has been of a formal nature. The two areas that have seen the greatest effort are synthesis systems based on theorem proving and low level programming aids such as syntax oriented program editors. Both of these areas have made important contributions to the field. But both of these areas require the human user to provide a formal input of some sort, be it predicate calculus statements or a computer program. For many purposes, such formal specifications are appropriate.

There is another approach that also shows promise of making the programmer's task easier. This approach emphasizes *informal* program specification. Some examples of this type of specification are natural language dialog, example pairs of inputs and corresponding outputs, execution traces of important process states, and graphical examples. Most researchers on informal program acquisition have proceeded by choosing one of these specification techniques and then writing a system that derives programs for a class of specifications using that technique.

The approach developed in this thesis looks at informality in programming terms, rather than in terms of the external specification technique. The goal is to find a common set of informalities that are useful for programming, independent of any particular specification technique. One offshoot of this approach is the creation of a formal programming language for the expression of informalities that itself might be the basis of useful program specification by people. In addition to its relevance to artificial intelligence ("AI"), this work may be of interest to such software systems research areas as semiautomatic programming aids, intelligent program editors, and incremental compilers.

1.1 Summary of Research

Program acquisition is the transformation of a program specification into an executable, but not necessarily efficient, program that meets the given specification. This thesis presents a solution to one aspect of the program acquisition problem: the incremental construction of program models from informal descriptions. The key to the solution is a framework for incremental program acquisition that includes (1) a formal language for expressing program fragments that contain informalities, (2) a control structure for the incremental recognition and assimilation of such fragments, and (3) a knowledge base of rules for acquiring programs specified with informalities.

The thesis describes a LISP based computer system called the *Program Model Builder* (hereafter abbreviated "PMB"), which receives informal program fragments incrementally and assembles them into a very high level program model that is complete, semantically consistent, unambiguous, and executable. The program specification comes in the form of partial program fragments that arrive in any order and may exhibit such informalities as inconsistencies and ambiguous references. Possible sources of fragments are a natural language parser or a parser for a surface form of the fragments. PMB produces a program model that is a complete and executable computer program. The *program fragment language* used for specifications is a superset of the language in which program models are built. This *program modelling language*

is a very high level (hereafter abbreviated "VHL") programming language for symbolic processing that deals with such information structures as sets and mappings.

PMB has expertise in the general area of simple symbolic computations, but PMB is designed to be independent of more specific programming domains and particular program specification techniques at the user level. However, the specifications given to PMB must still be algorithmic in nature. Because of the VHL nature of the program model produced, PMB also operates independently from implementation details such as the target computer and low level language. PMB has been tested both as a module of the PSI program synthesis system [Green-76A, Green et al.-79] and independently. Models built as part of PSI have been acquired via natural language dialogs and execution traces and have been automatically coded into LISP by other PSI modules. PMB has successfully built a number of moderately complex programs for symbolic computation.

By design the user is allowed to have control of the specification process. Therefore PMB must handle program fragments interactively and incrementally. Interesting problems arise because these informal fragments may arrive in an arbitrary order, may convey an arbitrarily small amount of new information, and may be incomplete, semantically inconsistent, and ambiguous. To allow the current point of focus to change, a *program reference language* has been designed for expressing patterns that specify what part of the model a fragment refers to. Various combinations of syntactic and semantic reference patterns in the model may be specified.

The recognition paradigm used by PMB is a form of subgoaling that allows the parts of the program to be specified in an order chosen by the user, rather than dictated by the system. Knowledge is represented as a set of data driven antecedent rules of two types, *response rules* and *demons*, which are triggered respectively by either the input of new fragments or changes in the partial program model. In processing a fragment, a response rule may update the partial program model and create new subgoals with associated response rules. To process subgoals that are completely internal to PMB, demon rules are created that delay execution until their prerequisite information in the program model has been filled in by response rules or perhaps other demons.

PMB has a knowledge base of rules for handling modelling language constructs, processing informalities in fragments, monitoring the consistency of the model, and transforming the program to canonical form. Response rules and simple demons are procedural. *Compound demons* have more complex antecedents that test more than one object in the program model. Compound demons use declarative antecedent patterns that are expanded automatically into procedural form.

1.2 The Problem

The two key problems faced by PMB come from processing fragments that specify programs *incrementally* and *informally*.

1.2.1 Incremental Specification

The notion of incremental program specification means that the fragments specifying a program may be received in an arbitrary order and may contain an arbitrarily small amount of new information. Fragments are accepted in any order to allow the user to provide new knowledge about any part of the program at any time. The amount of new information conveyed by each fragment is allowed to be small in order to provide the greatest flexibility for interactive, incremental specification. For example, a single fragment conveying a small number of pieces of information is the statement "*A* is a collection."[1]. This identifies an information structure called *A* and defines it as a **collection** of objects. However, the fragment says nothing about what sorts of objects comprise the **collection**, whether *A* is a set or a list, how many elements are in *A*, where and how *A* is used in the algorithm, etc. All of these details are provided in other program fragments that may occur either *before* or *after* this one.

With respect to the feature of accepting fragments in arbitrary order, PMB is analogous to an intelligent program editor. Whereas nearly all interactive editors are text or, at most, syntactic editors, PMB incorporates knowledge of the semantics of a particular programming language so that higher level feedback can be given to the user incrementally and so that only legal programs will be admitted in the end.

1.2.2 Informal Specification

The use of informality means that fragments may be incomplete, semantically inconsistent, or ambiguous; may use generic operators; and may provide more than one equivalent way of expressing a program part.

The description of one part of a program model may be incomplete at any point during model building. It may then be completed either by use of a default value, by inference by PMB, or from later fragments from the user.

Program model consistency is monitored at all times. PMB tries to resolve inconsistencies first; otherwise, it reports them to the user. For example, the fragment

$$x \in A$$

(a Boolean operation that checks if object x is an element of collection A) requires that either A have elements of the same type as x (whenever the types of A and x finally become known) or the type of one of them be inferred to be the same as the other.

Because a fragment may possess ambiguities, it may be interpreted in a number of ways, depending upon the program model context. For example, PMB specializes a generic operator into the appropriate primitive operation, based upon the information structure used. For example,

---

[1] In this thesis the names of program constructs or entities are set in special type faces. Control and information structure types appear in **boldface**. The names of other primitive operations, as well as information structures and procedures defined by the user, appear in *italics*. Be aware that italics is sometimes also used in standard ways, e.g., for emphasis and to denote special terminology that is being defined.

part_of(x,A)

(a Boolean operation that checks if information structure x is somehow contained within information structure A) becomes

$$x \in A$$

if A is a collection with elements of the same type as x. However, if A were a plex (record structure) instead, an *is_component* operation might be used to test whether x is the value of one of the components of A.

Within the modelling language there may be a number of formally equivalent ways to encode the same expression or information structure. An important task for PMB is *canonization*, the transformation of information and procedural structures into concise, high level, canonical forms. The intent is to map equivalent expressions into one canonical form whenever they are detected. This allows subsequent automatic coding the greatest freedom in choosing implementations. Interesting patterns are detected by specific rules set up to watch for them. For example, expressions that are quantified over elements of a set are canonized to the corresponding expression in set notation. Here is an expression that uses a notation similar to that of the predicate calculus to represent a predicate over all elements of some universe:

$$(\forall\, x)\; x \in A \supset x \in B$$

This Boolean expression determines whether every element x in collection A is also in collection B. The process of canonizing transforms this expression into

$$A \subseteq B$$

(is A a subset of B).

1.3 Control Structure

The model building problem is one of acquiring knowledge from the external environment. This knowledge takes the form of a program model. But many other domains of knowledge based understanding (e.g., natural language, speech, vision) have analogous problems and knowledge bases. The general paradigm that many of these systems follow is called "recognition" [Minsky-75, Bobrow & Winograd-77]. In this paradigm, the system watches for new information, recognizes the information based upon the system's knowledge of the domain and the current situation, and then integrates the new knowledge into its knowledge base.

The control structure of PMB is based upon the recognition paradigm, and has one key feature: PMB subgoals may be dealt with in an order chosen by the user, rather than dictated by the system. Subgoals are satisfied either externally or internally to PMB. The two cases are handled by the two kinds of data driven antecedent rules, *response rules* and *demons*, which are triggered respectively by either the input of new fragments or changes in the partial program model. When new information arrives in fragments, appropriate response rules are triggered to process the information, update the model being built, and perhaps create more subgoals and associated response rules. Each time a subgoal is generated, an associated "question" asking for

new fragments containing a solution to the subgoal is sent out by PMB to its external environment. This process continues until no further information is required to complete the model. To process subgoals that are completely internal to PMB, demon rules are created that delay execution until their prerequisite information in the program model has been filled in by response rules or perhaps other demons. Information is added to the program model monotonically. Therefore, if an inconsistency caused by the most recent fragment can't be resolved automatically by PMB, the last fragment must be changed by the user.

The incremental approach of PMB may be contrasted with the approach used in SAFE, the only comparable program acquisition system [Balzer et al.-78]. A SAFE program specification, in the form of a preparsed English paragraph, is passed through three noninteractive phases. The first acquires domain knowledge in the form of relations by recognizing what relations exist in the sentences; the second infers ordering constraints on the parts of the program; and the last partially symbolically evaluates the program to fill in missing operands and other information. The output is a program in an AI language that includes demons as a standard control structure and relations as the only information structure. The last stage of SAFE handles completeness and consistency issues, but not incrementally–except in the sense that it finds problems in the order of program execution. If there are unresolvable errors in the program, the specification must be changed and the system restarted.

1.4 Knowledge Base

PMB has a knowledge base of rules for handling constructs of the program modelling language, processing informalities in fragments, monitoring consistency of the model, and doing limited forms of program canonization. These rules about the modelling language include facts about five different information structures, six control structures, and approximately twenty primitive operations. The control structures are ones that are common to most high level languages. The language's real power comes from its very high level operators for information structures such as sets, lists, mappings, records, and alternatives of these.[2]

Below are English paraphrases of three rules that exemplify the major types of rules used in PMB. The first rule is a response rule for processing a new loop. The second is a demon that checks that the arguments of an *is_subset* operation are consistent. The third is a canonization demon that transforms a case into a test when appropriate.

[2] A complete list of the constructs in the program modelling language is given in Chapter 7.

Examples of Three Types of Model Building Rules

A **loop** consists of an optional initialization, required body, and required pairs of exit tests and exit blocks. Each exit test must be a Boolean expression occurring within the body.

Require that the two arguments of an *is_subset* operation both return collections of the same prototypic element.

if
(1) the statement is a case,
(2) the case has two condition/action pairs, and
(3) the first condition is the negation of the second condition,

then
change the case into a test.

Knowledge is represented as a set of data driven antecedent rules that are triggered by either the input of new fragments or changes in the partial program model. The rules are separated into two categories, response rules and demons, based upon these two ways of being triggered. Both response rules and simple demons are procedural. Compound demons (i.e., those whose antecedents test more than one object in the program model) use declarative antecedent patterns that are expanded automatically into procedural form.

As an example, consider the above response rule for loops. When a loop is required, this rule creates a unique template in the program model for the loop. Then the rule sets up subgoals for the loop's initialization, body, and exit blocks, along with appropriate response rules for each. These rules will process the parts of the loop as they arrive in fragments, and then store the results in the loop template in the program model. Questions soliciting information about these loop components are sent outside PMB to the user or other knowledge sources. In addition, for each exit test a demon is created that will wait until the location of that test becomes known and then check that the location is within the body of the loop.

1.5 Example of PMB in Operation

The example from Chapter 4 of PMB building one entire program model is excerpted below in order to give the reader the flavor of the two types of processing going on: (1) growth of the program model tree in a fashion that is generally top down, but data driven, and (2) completion and monitoring of parts of the model by demons. Note that this excerpt does justice neither to the concept of arbitrary order of fragments nor the types of programming knowledge in PMB.

The trace includes four of the program fragments that were generated by the PSI parser/interpreter from an English dialog. Before each fragment, a hypothetical English sentence that might result in such a fragment is given. Each fragment is followed by a description of how it was processed by PMB, a snapshot of the partial program model at that point, and a list of the outstanding demons. The processing of the first fragment presented is traced in greater detail than the rest, in order to show PMB focusing on individual slots of a

fragment, creating model templates, and creating subgoals. The discussion of the last two fragments emphasizes the creation and triggering of demons.

Comments interspersed within the trace are indented. Fragments and models are printed in a PASCAL-like notation, although they are maintained internally as property lists. Names preceding colons are unique template names (analogous to statement labels in ALGOL) that allow fragments to refer to different parts of the model. Mnemonic names have been assigned wherever possible to avoid using the computer symbols generated by PSI. Missing parts of the partial program model that are still to be filled in by later fragments are denoted by "???". Lines that are new or have changed from the immediately preceding model or demon list are denoted by the character "|" at the right margin.

Excerpt of Model Building Trace

The excerpt starts after the first fragment has already caused the partial program model shown below to be created. It only contains the names of the model, CLASSIFY, and the main algorithm, "algorithm_body". No demons have been created yet.

```
Current program model:

program classify;

   algorithm_body: ???

Current demons active:
    None
```

The second fragment describes the top level algorithm as a control structure having type composite and two steps called "input_concept" and "classify_loop". This fragment might have arisen from a sentence from the user such as "The algorithm first inputs the concept and then classifies it.".

```
Inputting fragment:

algorithm_body:
   begin
      input_concept;
      classify_loop
   end
```

A composite is a compound statement with a partial ordering on the execution of its subparts. The partial ordering is optional and defaults to sequential. The response rule that processes the composite creates the following two subgoals (or questions), along with response rules to handle the answers (not shown):

```
Processing ALGORITHM-BODY.TYPE = COMPOSITE
    Creating subgoal:
        ALGORITHM-BODY.SUBPARTS = ???
    Creating subgoal:
        ALGORITHM-BODY.ORDERINGS = ???
Done processing ALGORITHM-BODY.TYPE = COMPOSITE
```

Within the same fragment the two subparts are defined as operational units with unique names, but of unknown types. An *operational unit* can be any control structure, primitive operation, or procedure call. Two new templates are created and their types are requested.

```
Processing ALGORITHM-BODY.SUBPARTS = (INPUT-CONCEPT CLASSIFY-LOOP)
    Creating template INPUT-CONCEPT with value
        INPUT-CONCEPT.CLASS = OPERATIONAL-UNIT
    Creating subgoal:
        INPUT-CONCEPT.TYPE = ???
    Creating template CLASSIFY-LOOP with value
        CLASSIFY-LOOP.CLASS = OPERATIONAL-UNIT
    Creating subgoal:
        CLASSIFY-LOOP.TYPE = ???
Done processing ALGORITHM-BODY.SUBPARTS = (INPUT-CONCEPT CLASSIFY-LOOP)
```

At this point, the program model is missing the definitions of the two parts of the composite.

```
Current program model:
```

program *classify*;

begin
  input_concept: ???;
  classify_loop: ???
end

```
Current demons active:
    None
```

The next fragment defines an *input* primitive operation that reads from the user an information structure of type *concept_prototype*. The three arguments of an *input* operation are the prototype of the information structure to be input, the source of the input, and a prompt string to be output just prior to input. This fragment would be generated from the same sentence as the last fragment: "The algorithm first inputs the concept and then classifies it.".

```
Inputting fragment:
```

input_concept: *input*(*concept_prototype*, user, *concept_prompt*)

PMB creates information structure prototypes called *concept_prototype* and *concept_prompt*. PMB infers that the object of type *concept_prototype* that is input should be saved in an instance (or variable) of that type. So PMB creates one called *concept* and puts the *input* inside a *remember* operation (denoted by "←" below). Prototypes are listed after the keyword type below, and instances of prototypes are listed after var.

```
Current program model:

program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        classify_loop: ???
    end

Current demons active:
     None
```

Now the fragment that defines the second step of the composite is processed. This fragment might have been produced from the following sentence: "The classification step is a loop with a single exit condition.".

```
Inputting fragment:

classify_loop:
   until exit (exit_condition)
      repeat loop_body
      finally exit:
   endloop
```

This fragment defines a loop that repeats "loop_body" (as yet undefined) until a Boolean expression called "exit_condition" is true. At such time, the loop is exited to the empty exit block, called "exit", which is associated with "exit_condition".

At this point PMB doesn't know for sure where in the algorithm the test of "exit_condition" will be located, so it is shown separately from the main algorithm below. The response rule that processes the loop needs to guarantee that "exit_condition" is contained within the body of the loop. Since this can't be determined until the location of "exit_condition" is defined in a fragment, the response rule creates a demon to wait until this event. So Demon 1 (the number is assigned only for identification) is created and attached to the template for "exit_condition". Here Demon 1 will await the definition of the control structure that contains "exit_condition".

Similarly, Demon 2 is created to await the location of "exit_condition" and then put it inside a test with an *assert_exit_condition* as its true branch. This will cause the loop to be exited when the exit condition becomes true.

```
Current program model:

program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                loop_body: ???
            finally
                exit:
        endloop
    end

    exit_condition: ???

Current demons active:
    Demon 1:  awaiting control structure containing "exit_condition"
    Demon 2:  awaiting control structure containing "exit_condition"
```

The final fragment of this excerpt defines the body of the loop, thus triggering the two demons set up previously. One English specification that could be the source of this fragment is "The loop first inputs a scene, tests whether the datum that was input is really the signal to exit the loop, classifies the scene, and then outputs this classification to the user.".

```
Inputting fragment:

loop_body:
    begin
        loop_input;
        exit_condition;
        classification;
        output_classification
    end
```

So "loop_body" is a composite with four named steps. Even though none of the four is fully defined yet, PMB now knows where "exit_condition" occurs and that it must return a Boolean value.

Demon 1 is awakened to find that "exit_condition" is located inside the composite "loop_body". Since this isn't a loop, Demon 1 continues up the tree of nested

control constructs. It immediately finds that the parent of "loop_body" is the desired loop. So Demon 1 succeeds and is destroyed, as are most demons after they succeed.

Demon 2 is also awakened. Since it now knows "exit_condition" and its parent, Demon 2 can create a new template between them. The demon creates a test with "exit_condition" as its predicate and an *assert_exit_condition* that will leave the loop as its true action.

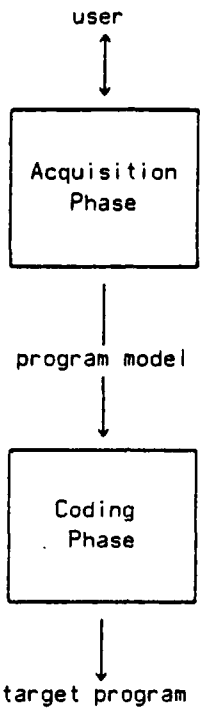
```
Current program model:

program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    loop_input: ???;
                    if exit_condition: ??? then assert_exit_condition(exit);
                    classification: ???;
                    output_classification: ???
                end
            finally
                exit:
        endloop
    end

Current demons active:
    None
```

At the end of the excerpt, five fragments have been processed, and 27 more must be before the program model is complete.

1.6 Role of PMB in a Program Synthesis System

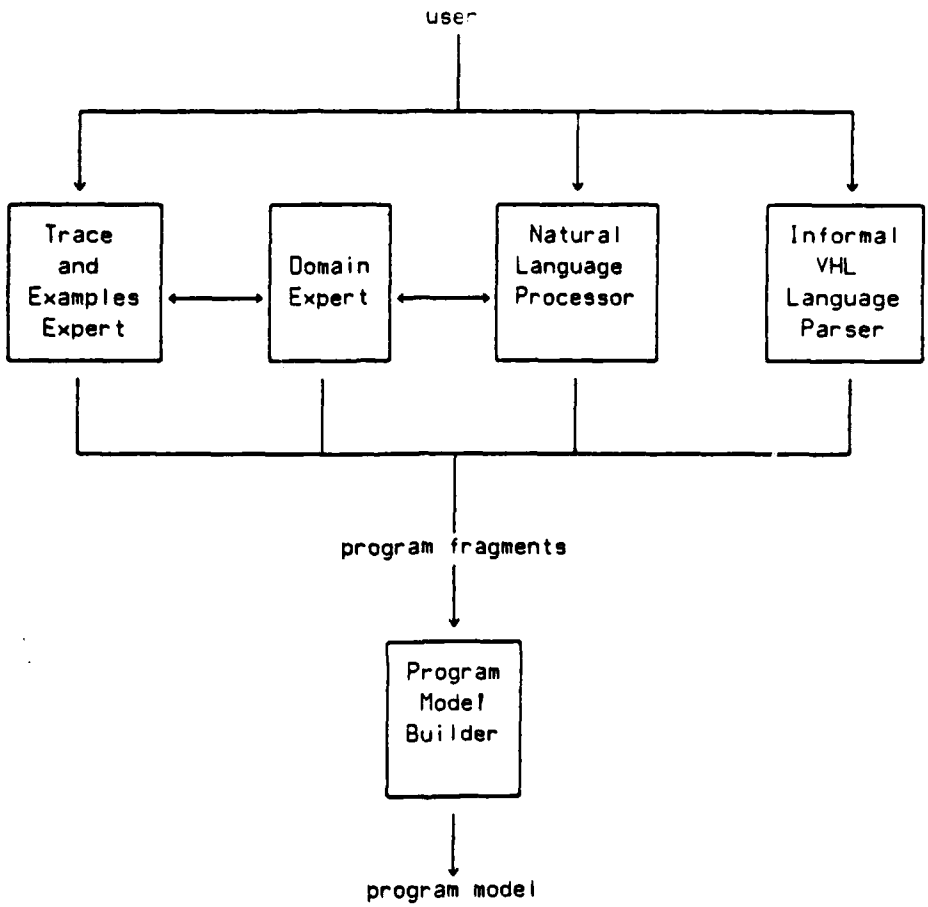
PMB was designed to operate as part of a more complete program synthesis system with two distinct phases: acquisition and automatic coding. In such a system the program model would serve as the interface between the two phases. Automatic coding is the process of transforming a program model into an efficient program, without human intervention. Program models are acquired during the acquisition phase; coding of the model is only done when it is complete and consistent. A system organization based on this paradigm is shown in the diagram below.

Two-Phase Program Synthesis Paradigm



PMB was designed so that it may operate within a robust acquisition environment. In such an environment, program fragments may come from many other knowledge sources, such as those expert in traces and examples, natural language, and specific programming domains, as depicted in the diagram below. However, the operation of PMB is not predicated on the existence of other modules, each of which is a complex AI program in its own right. For example, all fragments to PMB could be produced by a straightforward deterministic parser for an informal VHL surface language, such as the one used to express program fragments.

Typical Acquisition Environment for Which PMB Was Designed



In addition to providing a focus and testbed for developing a framework of program acquisition, PMB plays a central role as one of the expert modules or knowledge sources of the PSI program synthesis system [Green-76A, Green et al.-79]. PSI is divided into acquisition and coding phases, with PMB as part of the acquisition phase. In this first phase a VHL model of the desired program is acquired from the user. In the second phase an efficient program is coded from this model. The user's external program specification (e.g., English) is first transformed into fragments by other acquisition modules that are experts on particular specification techniques or programming domains. Then PMB builds a complete and consistent program model based upon the user's program specification. When complete, this program model is refined by the PSI coding phase into an efficient target program [Barstow-79A, Kant-79A, Kant-79B]. In actual operation, PMB has built models from fragments produced by PSI's English parser/interpreter [Ginsparg-78] and PSI's expert on program inference from traces and examples [Phillips-77].

1.7 Outline of Thesis

Subsequent chapters discuss the problem area in detail, examine all aspects of the solution, and end with a discussion of the results.

The program acquisition problem is defined in some detail in Chapter 2, because PMB was designed to fit within a particular framework for automatic (or, more accurately, semiautomatic) programming. Chapter 2 is my personal view of the program synthesis research arena. The chapter is a discussion of the important problems and a global system design strategy for solving them. The discussion starts with a general definition of program synthesis and progresses to a detailed definition of the program model building problem in terms of the differences between PMB's input and output. Of course, much of the discussion also reflects conceptions—and biases—resulting from my interactions with the PSI group over a number of years.

Chapter 3 provides a brief survey of related research areas. It is designed only to provide pointers to the literature and a basic framework for understanding how PMB relates to the rest of program synthesis, artificial intelligence, and software systems. Detailed comparisons of approaches and techniques are contained in the relevant sections elsewhere.

A detailed example of the operation of PMB is presented in Chapter 4. This provides the details of building an entire program model for a fairly complicated program, from start to finish. In addition to an annotated trace of model building, the English dialog that was the source of program fragments is given. A sample execution of the resultant program model is listed too.

Chapter 5 discusses the format of fragments and the aspects of informality that make the program fragment language different from other programming languages. The last section of the chapter introduces the program reference language, which could be used to describe where in the program model a fragment is to be incorporated. The discussion will interest those concerned with intelligent program editors and other programming aids. A more complete description of the reference language is presented in Appendix A.

Chapter 6 describes the control structure of PMB, which consists of user directed subgoaling with asynchronous demon invocation. This structure is compared to other problem solving techniques used in both AI (notably, natural language understanding) and software systems, and the generality of the techniques is related to structured programming. Chapter 6 is probably of greatest interest to people in artificial intelligence.

The program modelling language is briefly defined in Chapter 7. That material will be of most interest to designers of programming languages, especially very high level languages.

Chapter 8 defines the structure of the rule base of program acquisition knowledge and the types of rules in it. It also provides some example rules and a discussion of the various types of knowledge in PMB. Many more rules are listed in Appendix B. Chapter 8 will probably be of greatest interest to those doing research in knowledge based systems or program synthesis.

The conclusion, Chapter 9, discusses the program models that have been built; the contributions of the thesis to the areas of program synthesis, artificial intelligence, and software systems; and directions for extensions of the research to remove some of its limitations.

Chapter 2. The Problem

The problem addressed by this thesis is termed the "model building" problem. The discussion of the problem is comprehensive because the area is relatively new and experience has shown that the problem is not a particularly intuitive one. The key notions involved in building program models are not directly analogous to any one body of prior research.

The goals of research on PMB are quite similar to the broader goals for program synthesis systems such as PSI. In both cases, people outside of the field often have difficulty understanding these goals. Hence, it is important to clearly state the case for systems of this type.

We present a top down development of the problem definition. First the context within the larger, more general program synthesis problem is presented. Then major assumptions, design goals, and design decisions of the current work are discussed.

2.1 The Program Synthesis Context

The model building problem that is addressed is at the bottom of a hierarchy of problems.

The top level problem is the general *program synthesis problem*[1]: Transform a program specification expressed in some formal or informal specification language into an executable and efficient program that meets the given specification. This is a very broad problem, including not only artificial intelligence approaches, but standard compiler technology as well. However, the most difficult (and perhaps most interesting) case arises when the program specification is not directly executable in its initial form.

The subproblem of interest within the program synthesis problem is the *program acquisition problem:* Transform a program specification into an executable, but not necessarily efficient, program that meets the given specification. This subproblem merely eliminates the requirement that the program resulting from the transformation be efficient.[2]

Finally we reach the specific problem dealt with by this thesis, a part of the program acquisition problem called the *model building problem:* Receive informal program fragments incrementally and assemble them into a very high level program model that is complete, semantically consistent, unambiguous, and executable. Here the program specification comes in the form of partial program fragments that arrive in any order and may exhibit inconsistencies and ambiguous references. The program produced is a program model that is a complete and executable computer program.

---

[1] Sometimes also called the "automatic programming problem"

[2] There is another subproblem, called *problem acquisition*, which additionally removes the requirement that the result of the transformation be an executable program. An example is a set of statements in the predicate calculus that, when processed by a theorem prover, yields a result, possibly in the form of a computer program [Green-69]. The set of statements is not itself a program, rather merely a problem statement that, when solved, will result in a computer program. Arbitrary problem acquisition is not the focus of this thesis.

2.2 Limiting Assumptions

First we note two fundamental limitations that we have imposed in order to make the model building problem tractable.

2.2.1 User Group: Programmers

A fundamental assumption of our approach is that the user of the system is assumed to be familiar with programming. In limiting users to programmers, we are implying that specifications will still reflect a particular algorithm and information structures, not something more abstract. The system is not expected to be especially creative with respect to designing its own algorithms.

Many people feel that it is time to forget programmers and try to help nonprogrammers directly. We feel that there are many difficult and important problems left to be solved in the quest of making programming easier for programmers before tackling the much harder problem of extending to nonprogrammers the capabilities now afforded only to programmers. We think that helping programmers with some of their less creative, more mundane tasks (e.g., bookkeeping) is an appropriate first step, and is on the critical path to achieving automatic programming for nonprogrammers as well.

Programming is an intellectual task that is highly labor intensive. As computers get cheaper and qualified programmers get relatively more expensive, automating as much of the programming process as possible becomes more and more important.

To accommodate users who are not programmers would introduce an entirely new level of complexity to our problem. One solution would be to merely add a front end or integrate into the original system a capability for the computer assisted teaching of programming. This solution avoids the problem of doing program acquisition for nonprogrammers by forcing the user to become a programmer in the standard sense.

A better solution would involve changing the task from *program* acquisition to *problem* acquisition. The problem would be stated in a nonprocedural language that is highly tailored to a specific task domain. Since the user is presumably an expert in this domain, the communication problem would therefore be reduced, but the system would then have the additional difficulty of converting the problem specification into a program. Solutions might involve techniques for algorithm creation [Tappel-79] and discovery[3] and certainly would require much domain specific knowledge. These are interesting research areas, but ones that are not covered here.

2.2.2 Programming Domain: Symbolic Computation

There really is no such thing as domain independent programming. Hence domain independent automatic programming, as purported in [Balzer et al.-74], doesn't exist either.

[3] For a possible approach to the discovery of algorithms, consider the work on automated mathematics discovery [Lenat-76, Lenat-77].

Instead there are hierarchies of programming domains, from very general to very specific.[4] For example, symbolic computation is a very broad domain; so is the domain of numerical algorithms. Algorithms for the solution of differential equations form a narrower domain, and the class of all programs based on the Runge-Kutta method is a very specific programming domain.

Some programming domain has to be chosen just to make our problem feasible. Hopefully the domain selected will have widespread importance. An even more important consideration is that the domain chosen not limit the generality of the program acquisition framework that is developed.

The programming domain chosen for this work is that of simple symbolic computations[5]—as opposed to, say, numerical or realtime control algorithms. For our purpose, the universe of possible programs is limited to those for symbolic processing, and specific programming domains are subsets of this universe. Example domains include set manipulation, list processing, searching, sorting, simple data storage and retrieval, pattern matching, and symbolic learning. Although, say, numerical algorithms could have been chosen, the area of symbolic programming is fundamental to much of computer science (even to some numerical algorithms). Knuth has said about just the subdomain of sorting and searching [Knuth-73B, page v]:

> Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting or searching!

2.3 Design Goals

Now we list some useful properties of a model building system that form the basic assumptions on which the thesis rests. In general, the goals discussed in this section stem from a desire to provide the user with more capabilities and flexibility in program specification, thereby forcing the program acquisition system, and PMB in particular, to provide more intelligent assistance to the user.

2.3.1 Very High Level Specification

Perhaps the most significant assumption is that much higher level (in terms of information and control structures) languages than are now prevalent will be used in our interactions with computers. Programming is a difficult intellectual exercise. Part of the difficulty is the sheer complexity of the details required to successfully compose a nontrivial program in any of the commonly used languages. Program specification and subsequent modification will be done at a higher conceptual level than at present, to reduce the magnitude of the programming details involved and the complexity of communicating with the user. This implies that the program model is updated during interactive acquisition, rather than the target program being manipulated incrementally. For example, PMB builds programs in a very high level (VHL) language, and automatic coding systems such as the one in PSI take care of the details of

[4] Of course, domains may overlap as well.

[5] Of course, at the lowest level, *all* computation is symbolic in nature.

translating this language to lower level ones.[6]

Limited English is one VHL language suitable for certain applications. This does not imply that an algorithmic language such as ALGOL has no place, but merely that it can often be successfully replaced. When specifying a program in English to PSI, the user should not have to understand the target language program and make changes at that level.

Present day high level languages and their compilers provide a good analogy. Users need to write code at the level produced by the compiler only for special purposes; nearly all programming is carried out at the source language level.

Now add the assumption that the target language used for the synthesized program is significantly below the level of the specification language. Potential target languages today are conventional high level languages, assembly language, microcode, and possibly mixtures of these. There is a large gap between a specification language such as English and a target language such as assembly language, in terms of brevity, ambiguity, consistency, and completeness. It is clear that direct automatic conversion between the two, especially incrementally, would be very difficult. It is reasonable to interpose an intermediate level specification language into which the user's specification is first transformed. This is the role of the program modelling language used by PMB.

The level of this intermediate, acquisition level language, in terms of its control and information structure primitives, is an important design decision. Choosing too high a level (e.g., unrestricted predicate calculus) may result in an intractable coding problem. Choosing too low a level (e.g., assembly language) may unduly restrict the coding options available and make the acquisition process too difficult. Standard high level languages have often been proposed for this intermediate language. But they all have information structures that are designed for direct, efficient implementation on most computers. So high level languages are too low level for our needs because they restrict coding and make acquisition more difficult.

2.3.2 Interactive Specification

The system should be interactive so that it can provide immediate feedback to its user, either in response to questions or specifications that aren't fully understood. One rationale for interaction arises from the following analogy: Interactive program specification is to batch specification as interactive program development using high level languages and text editors is to batch programming using decks of cards. But an even a stronger analogy can be made, since most interactive program development today is really a form of online batch in which the text editor takes the place of the card punch and calling the compiler on the edited file corresponds to reading a card deck into a card reader.

This second analogy defines interactive program specification as a combined incremental

[6] Programming in standard high level languages and even assembly language is still done today because appropriate VHL substitutes are not widely available and, more important, the problem of coding efficient target programs from them has not been solved. But the trend toward higher and higher level languages will persist because of (1) the continuous (albeit slow) advances in these two areas and (2) the trend of software to cost increasingly more than hardware.

editor/compiler in which changes to statements or expressions are checked for consistency after each change. Such compilers today only allow simple editing (e.g., total replacement of a line of text) because of the difficulty of maintaining enough context for truly incremental compilation to take place. The approach taken by PMB is to maintain the program as a VHL model after checking for syntactic and semantic consistency, but not to code incrementally.

2.3.3 Incremental User Control

The perfect programming paradigm has yet to be devised. In particular, different programmers may desire and different applications may require different orders of defining the parts of a program. This is especially true for large systems (e.g., compilers, operating systems) in which the total complexity is so great that a programmer cannot plan out in advance all the possibilities that the program must consider. In such cases, the most general solution for an automatic programming system or semiautomatic programming aid is to allow an arbitrary order for defining, refining, and modifying the parts of the program.

We subscribe to the philosophy that the user should have the option of controlling the interaction by asking questions or changing the topic under discussion at any time. A user should be able to converse with an acquisition system as freely and easily as with another human programmer. Lacking this flexibility, an acquisition system will restrict the user and thus not be as habitable as is desirable.

This philosophy doesn't necessarily contradict the aims of the school of structured programming. Our goal is not to force the user to do low level programming and debugging on a program that was not well thought out. Our goal is to provide computer aids that will allow the high level thinking and planning phases of programming to take place in a more automated and less error prone environment.

There are two levels of incremental specification capability. A fully incremental system allows the user to go back and make changes to what has been specified before. A system with "monotonically" incremental specification doesn't allow a part to be modified, once it has been specified. The term "incremental", as used in this thesis, refers to the monotonic case.

There are two aspects of incremental programming: (1) the order that parts of the program are specified and (2) how much is specified at each step.

Since the user controls the interaction, the user may determine the order in which information flows into PMB. Information is received piecemeal, as chunks or fragments of program description. These fragments may arrive in arbitrary order, as long as they carry enough program reference information to determine their context in the program model being assembled.

In addition to deciding what part of the program to work on and when, the user may decide how much of the program part to specify at one time. This means that a program model may be incomplete at any level, from an entire procedure being missing to a single parameter of a single operation being left unspecified. This is the reason that PMB is capable of delaying its operations until the required arguments are completely specified.

Even if the user isn't exercising the option to take control of the interaction, information for

PMB may still arrive incrementally and arbitrarily ordered, since the generation of questions and program fragments may be mediated by the activities of other expert programs. For example, there might be experts on moderating dialogs, modelling the state of the user, or providing domain support. Any of these experts can intervene between PMB and the user so that questions are asked of the user in a much different order than PMB's default order. In addition, other experts may produce fragments as a result of their own reasoning. These may be sent to PMB independently of what is transpiring with the user. Thus, even if the user yields control of the dialog to the system, there is no guarantee that questions will be answered in PMB's default order.

2.3.4 Informal Specification

Another important feature of VHL model building is informality of specification. Most VHL language designs have concentrated on the issue of defining new and useful VHL programming primitives. This is certainly necessary, and indeed has been one part of this research. However, we believe that informality of specification is going to be an equally important design issue in the future. Any system capable of dealing with informality will have to possess a large amount of general programming knowledge so that many details can be handled automatically; so that ambiguities, omissions, and inconsistencies can be caught; and so that different ways of specifying the same thing can be handled.

An incremental, informal program specification does not constitute an executable algorithm. Before assembling the individual pieces of program and removing any informalities, the specification will lack the quality of "definiteness" that is required for any procedure to be considered an algorithm. This property requires that each step of the procedure be known and be precisely, rigorously, and unambiguously defined [Knuth-73A, page 5].

An executable algorithm isn't very useful if it isn't also correct. We use a less formal sense of program correctness than that used in the program verification field [Luckham-77, London-77]. Here a "correct" program model is one that the user is satisfied meets specifications (e.g., by interpreting it on test data or reading a listing of it) and that the system is satisfied is complete and consistent. Thus PMB is only concerned with guaranteeing that the final model is a legal program syntactically and semantically. Verifying that the program is correct in a pragmatic sense is left to the user or perhaps other knowledge sources. To help out in this endeavor, assertions could be attached to the program model. These assertions would then be evaluated during model interpretation to make sure they are always true.

Incompleteness

Compared with their communication with computers, people communicate with one another fairly efficiently. This is partly because many assumptions are in force and need not be restated during each conversation. Informality is the analog in program acquisition. VHL primitives allow programs to be expressed succinctly; informality encourages succinctness by having PMB *infer* details that aren't stated explicitly. To do this requires rules about default values for pieces of programs and rules for computing the appropriate value from elsewhere in the program.

Another issue is temporary (or local) incompleteness. In this case information is missing and not inferable. PMB asks for the information and only continues processing that is based on it after it is provided.

Semantic Inconsistency

Statements by people are not consistent, especially during such complex intellectual exercises as programming. Informality allows for semantic inconsistencies among fragments; these are noticed by PMB and either resolved automatically or brought to the attention of the user. For example, a reference to an information structure definition when its value is required instead can be corrected automatically.

Ambiguous Operands

Ambiguity is also a powerful tool for making succinct program specifications. These have to be appropriately interpreted from context by the acquisition system. For example, there are many ways to refer to a particular information structure, in this case a particular set named *A*:

the last set I talked about
the set *A*
the set containing elements of type x
the smallest set

A program reference language has been designed for PMB, but not implemented. If the examples above were specified in this language, they would be ambiguous. If the last statement from the user referred to "set *B*, which is a subset of set *C*", then which of those sets is the last one talked about? There might be two sets called *A*, declared in two different subroutines. There might be two sets containing elements of type *x*. And the smallest set (presumably out of all known sets) may vary during program execution. If by "smallest" is meant the set with the smallest minimum size, then there still could be ambiguity. In most cases, which set is meant is easily determined by what part of the program is currently being built or which set meeting the specifications was referred to most recently.

Generic Operators

Another useful type of informality is the generic or "polymorphic" operator. Just as ambiguous references to operands naturally arise, a generic capability makes it possible for operators to express ambiguity too. For example, *part_of* returns a subpart of some other information structure. Since *part_of* doesn't exist in the program modelling language, it is transformed into the appropriate primitive operation based upon the information structure it operates on. For a collection (e.g., a set), *part_of* might become an *is_element*; for a plex (record structure), an *is_component*.

Multiple Equivalent Specifications

Another aspect of informality is the desire to allow alternate ways of specifying the same underlying concept or action, even within the same basic specification method (e.g., English). All of these equivalent specifications are mapped into the same final form in the program model: the canonical form, the form that is the most concise expression of the concept and that will allow the greatest freedom of choice when it is coded. This requires program equivalence transformations that recognize opportunities for and then carry out program canonization.

2.3.5 Program Modification

Studies of the programming life cycle have shown that most programs that are used over any length of time are written once from scratch and then subsequently modified numerous times. So programming consists of much more modification, reprogramming, or maintenance than initial programming. Modification may be required because of the discovery of programming errors, changes in design goals, or changes in the program's execution environment (e.g., changes in the format of the program's input). Note that a partial program being written can undergo modification, as well as a previously completed program. Although a modification capability is a goal of this line of research, it is not available in PMB.

Just as with initial program acquisition and coding, desired modifications should be expressed and carried out at the highest level feasible, not at the target language level. The program model should be modified, and a new and efficient target program produced from the new model.[7] Compilers provide an analogy. Users need to look at target code only for special purposes; most interaction is carried out at the source code level.

Modifications can be complicated to handle. A minor change in one part of a program can have major effects throughout. Thus, conceptually simple changes may be difficult because of all of the places in the model that must be updated. Of course, this is the sort of bookkeeping that computers should be used for.

2.3.6 Target Program Goals

Now consider goals for the final programs produced by the entire synthesis system.

Efficiency

The target programs produced should be reasonably efficient in both space and time.

Flexibility

The system should be, in principle, capable of generating programs in more than one language, for more than one computer, and for varying assumptions about the inputs (e.g., size, distribution).

Independence of Specification and Coding

The program specification process should be independent of the implementation details. The user should not have to change to accommodate a change in the level of the target language (e.g., standard high level language, assembly language, or even microcode), the particular target language (e.g., LISP or PASCAL), or the target machine.

---

[7] Whether or not any of the old target program should be used or modified is a question for the coding phase of how efficient this would be versus coding an entirely new program from scratch.

2.4 Program Synthesis Paradigm: Separate Acquisition and Coding Phases

Our solution to some of the design goals discussed in the preceding section is a program synthesis paradigm that separates the problem into two distinct phases: acquisition and coding. The *program acquisition* problem is to receive a specification of the desired program, expressed in one or more formal or informal languages, and to construct an effective procedure for realizing the specified program. The *automatic coding* problem is to turn such an effective procedure into an efficient program. The interface between the two phases is a *program model* expressed in a very high level program modelling language for symbolic processing. The program produced by PMB is termed a "model" because of the desire to model the corresponding program in the user's head and because it is an abstract, implementation independent program specification that may actually lead to many different concrete implementations. Program models are constructed during the acquisition phase; coding from the model is only done when it is complete and consistent. A system organization based on this paradigm was shown in the diagram labeled "Two-Phase Program Synthesis Paradigm" in Section 1.6.

There are many reasons for dividing the problem into acquisition and coding parts, given the assumptions stated earlier. Both program acquisition and automatic coding are known to be difficult problems, quite likely more successfully dealt with as separate subproblems.

Acquisition and subsequent modification of a program should be done at a high conceptual level, to reduce the magnitude of the programming details involved and the complexity of communicating with the user. This implies that updating is done to the program model during interactive acquisition and that incremental manipulation of the target program is not done.

Coding can be done most effectively if the complete program is available from the start and is known to be correct. Coding can proceed with little or no interaction with the user. In contrast, incremental program specification by definition builds up a complete program from scratch. Thus, acquisition relies primarily upon forward inferencing from incrementally acquired information, while coding may use backward chaining to make deductions from the completed model.

Dividing program synthesis into two phases separated by the program model allows programs to be optimized by taking different runtime environments into account. The program can be acquired once and a program model built. Then different programs can be produced by specifying different execution estimates of the model (e.g., set sizes and branching probabilities), a different target language (which affects what primitives are available and how much they cost), and even a different target machine (i.e., instruction set). The programs will of course have the same input/output behavior, but the code will be designed and optimized for the particular environment.

Further details of approaches to the automatic coding problem are found in [Barstow-79A], [Kant-79A], and [Kant-79B]. Coding considerations are for the most part ignored throughout the remainder of the present work.

An alternative design for a program synthesis system is a monolithic system with only one phase or "pass" [Phillips-79]. This approach obviates the VHL program modelling language and perhaps avoids having the same knowledge in more than one place in the system. In addition to the possibility of simply emulating the two phase approach, a one phase system has

the flexibility to explore coding possibilities for parts of a program before the rest has been fully specified. This system uses a single language to express programming concepts from the most informal specification to the most concrete target language detail. The forms of knowledge (e.g., rules) may be unified and a single unified global data structure kept.

2.5 Additional Design Goals for Acquisition

Now, concentrating on the acquisition phase, we add two more design goals.

2.5.1 Multiple Specification Techniques

A number of different program specification techniques should be allowed, separately or intermingled, reflecting the belief that different techniques are useful for different programs and different users [Green et al.-74]. Examples of candidate techniques include predicate calculus, informal VHL languages, noninteractive natural languages (e.g., a limited subset of English), natural language dialog, speech, execution traces of important process states, example pairs of inputs and corresponding outputs, and graphical examples.

A model for program specification using multiple techniques may be taken from Knuth's series of programming texts [Knuth-73A, Knuth-69, Knuth-73B]. In addition to the actual programs coded in the MIX assembly language, a number of higher level techniques are used to convey the intent of an algorithm to the reader. Knuth normally uses an informal programming notation embedded in explanatory English. He often supplements this description with graphical examples and traces of the program's operation.

A problem arises when more than one specification technique is allowed. The specifications must somehow be integrated into a single model of the desired program. This problem is exacerbated when each technique can only be understood by a complicated AI program. In addition to the programming knowledge specific to one specification technique, each of these programs would also require a large, redundant body of knowledge, both about the target programming language and about programming in general. A separate programming expert such as PMB would eliminate this redundancy.

2.5.2 Understanding Specific Programming Subdomains

Once a general programming domain (e.g., symbolic computation) has been chosen, the system should be capable of acquiring programs in very specific subdomains (e.g., concept learning, text editing, sorting, searching). This capability requires an understanding of the specific subdomain in terms of its "programming vocabulary" (i.e., standard information structures and algorithms).

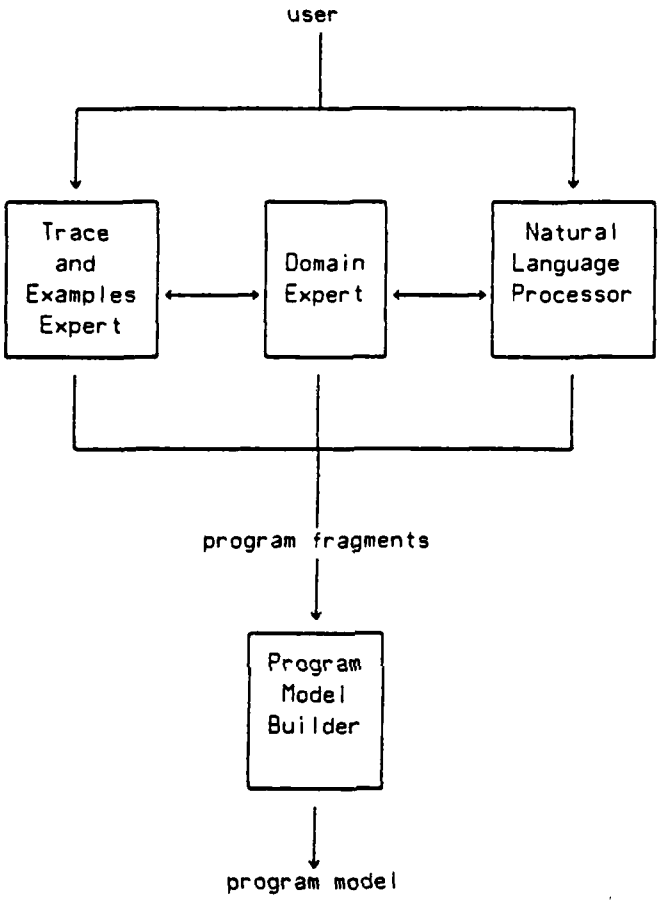
However, the system should also be able to get by without any knowledge of specific programming subdomains. Of course, without such knowledge the system will behave less intelligently and therefore rely more upon the user to provide the domain expertise (e.g., substitute programming terminology for domain specific terminology). But the system should still be functional.

2.6 Program Acquisition Organization: Independent Programming Expert

A tremendous amount of knowledge, both about the particular programming language and about programming in general, is required to write a correct program within the large programming universe of symbolic processing. Much of this knowledge is independent of the human user's choice of specification technique, and much of this knowledge is independent of any specific subdomain of symbolic processing. Therefore, it is possible and desirable to identify and codify this knowledge independently. This suggests an organization for the acquisition phase that includes an independent module embodying general programming knowledge.[8] Such an organization is shown below. This is essentially the organization of the acquisition phase of the PSI program synthesis system, in which PMB plays the role of programming expert.

[8] Another possible solution would be to simply have one enormous system that handles all aspects of acquisition. With our current understanding of the program acquisition problem and under current constraints on the size of programs, this is impractical. On a more fundamental level, as knowledge bases grow larger and larger, it is important to recognize and take advantage of any modularity inherent in the domain. A compromise would be to have separate modules that use standardized knowledge bases. Then knowledge could be shared when necessary.

Program Acquisition Organization with Independent Programming Expert



PMB accepts fragments from one or more specification modules and one or more domain modules. It is responsible for building the program model independently of any particular specification techniques and programming domains. This solution leads to a new, but more tractable problem: designing an interface language for describing the types of informal program fragments that are produced from external specifications.

2.7 Detailed Problem Definition: Differences between Fragments and Model

The following table and discussion summarize the differences between program fragments and the program model, thus providing a concise definition of the model building problem. These features all correspond to particular design goals for informal program acquisition discussed earlier in this chapter. The remainder of the thesis defines fragments and models in more detail, provides many examples of each, and presents the processing and underlying knowledge base necessary to transform one into the other.

Differences between Program Fragments and Program Model

Fragments (input)	Model (output)
Small chunks of program ...tion	Complete program description
Written using superset of ...m modelling language ...	Written in VHL program modelling language
Many independent sources	Produced only by PMB
Arbitrarily ordered	Always ordered in same way: information structures, procedures, algorithm
Incomplete	Complete, cross-referenced
Nonexecutable	Executable
Semantically inconsistent	Semantically consistent
Ambiguous	Unambiguous
Generic operators	Specialized operators for each information structure type
Many ways to say something	Concise, high level canonical form

Fragments are explicitly limited to the description of programs (as opposed to arbitrary problems) because of the assumption that the system will only deal with algorithms specified by programmers.

The amount of new information conveyed by each fragment is allowed to be small in order to provide the greatest flexibility for interactive, incremental specification. For example, a set *A* might be described by separate fragments conveying the following facts: "*A* is a collection", "*A* is unordered", "*A* has no repeated elements", "each instance of *A* has at least five elements".

Fragments are expressed using a superset of the primitives in the program modelling language so that (1) straightforward programs may be written in the modelling language using the same PMB mechanisms and (2) fragments will have at least the same power as the modelling language for expressing symbolic computation programs at a very high level.

Fragments are handled independently of their source, so that more than one specification or domain module may be active at once.

Fragments may be arbitrarily ordered to provide freedom to the user in ordering the specification process. For example, once *A* has been defined to be a collection above, the other three fragments may arrive in any order, with other fragments not referring to *A* occurring in between them.

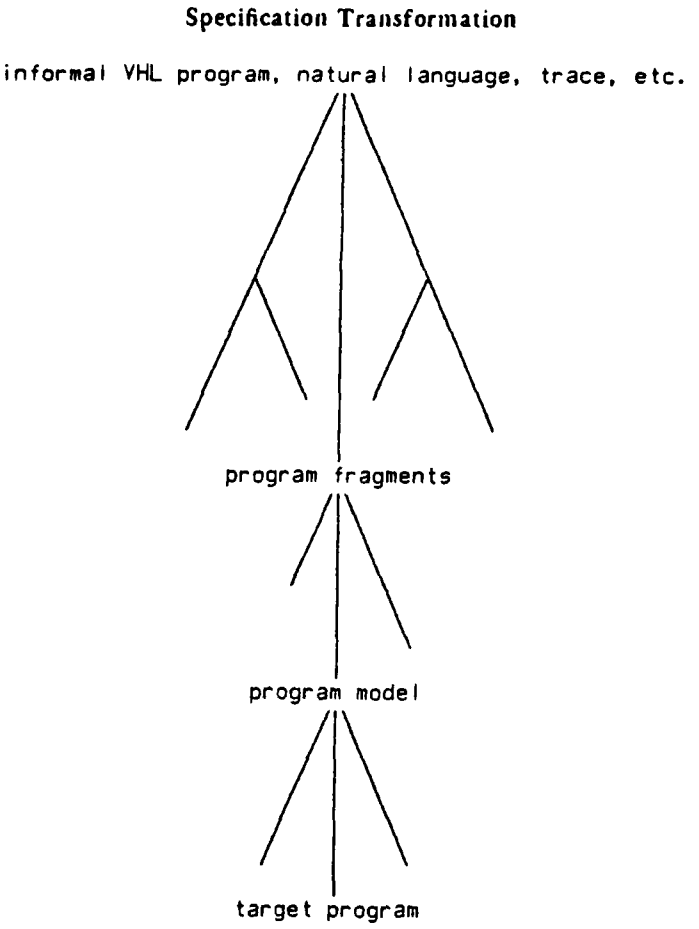
The fragments may be incomplete, semantically inconsistent, or ambiguous; may use generic operators; and may provide more than one equivalent way of expressing a program part. Until the fragments are built into a single model and such informalities removed, they obviously don't form an executable program.

2.8 Program Synthesis As Specification Transformation

Now we ascend to reconsider the general program synthesis problem. We have seen that this problem is to transform a human program specification into an efficient program that meets the specification. Each program specification that is informal or at a very high level can often be met by hundreds of reasonable target programs. The program synthesis system produces only one of these programs: (hopefully) the one that meets the program specification most efficiently under the known constraints.

This transformation process can be broken down into a small number of relatively sequential and independent phases that carry out part of the overall transformation process. Each of these phases concentrates on bringing one type of knowledge to bear on the problem. Each phase narrows the space of programs that are still under consideration. Adjacent phases communicate only via an appropriate interface language.

The diagram below shows the space of all possible programs being constrained further and further by each successive knowledge level, until finally exactly one target program is produced. The program specification is first transformed into program fragments by the application of knowledge of particular specification techniques and the particular programming domain. Fragments are transformed into a complete and consistent program model by the application of general programming knowledge. Then the model is transformed into a target program by the application of coding and target language knowledge. The middle step is the topic of this thesis.


Specification Transformation
informal VHL program, natural language, trace, etc.
program fragments
program model
target program

Chapter 3. Survey of Related Work

A number of distinct research areas are related to the present work. Rather than giving a detailed, but out-of-context discussion of them all in a single chapter early in the thesis, I only provide a brief mention of the more important pieces of related research here. Detailed discussions occur as appropriate in the chapters that present the relevant parts of PMB.

PMB is a program acquisition system. Artificial intelligence approaches to the program acquisition problem, as well as related problems of program synthesis and "automatic programming", are surveyed in [Heidorn-76], [Green-76B], [Biermann-76], and [Elschlager & Phillips-79].

The other areas covered in this chapter are the incremental acquisition of informal programs, programmer aid systems, recognition problem solving using demons, programming methodologies, very high level languages, knowledge representation by rules and frames, and standard compiler technology.

3.1 Incremental Acquisition of Informal Programs

No other research appears to have attacked the problem of acquiring programs that are specified both incrementally and informally.

The SAFE program acquisition system [Balzer et al.-78] is the closest in goals and scope to PMB. SAFE translates from a preparsed form of natural language into a VHL language featuring relations as its only data type. SAFE deals with some of the same issues of program incompleteness and inconsistency, but isn't incremental or interactive.

The NLPQ system acquires simulation programs from natural language [Heidorn-72, Heidorn-74, Heidorn-75]. This was the first successful natural language program acquisition system. It is not an incremental system, but does allow for questions at the end of specification, from the user for verification of the acquired program and from the system to fill in any gaps left in the program. Most inconsistencies are not discovered until the simulation program is run, and the class of programs handled is small.

The XREP system [Wilczynski-75] deals with the problems inherent in referencing variables by English noun phrases. XREP assumes that such a specification is parsed into a production rule programming language, along with a set of intentions (or plan). Then the program is executed to see if it matches the intentions.

Hobbs catalogs a number of completeness and consistency inferences and canonizing program transformations, many of which PMB can do [Hobbs-77A, Hobbs-77B]. But he is more interested in dealing with them at an earlier, more linguistic level, and a system encompassing his ideas hasn't been implemented yet.

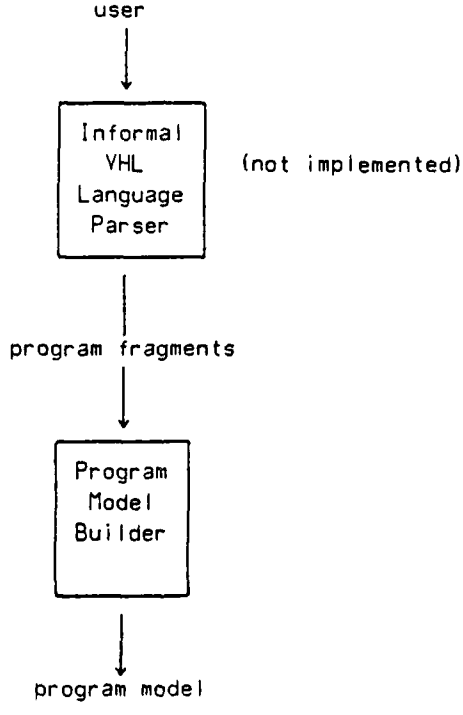
The SID verification system [Moriconi-77, Moriconi-79] is incremental, but deals with formal programs (in a PASCAL-like language) annotated with assertions. These are created and changed using a standard text editor.

3.2 Programmer Aid Systems

A number of attempts have been (and are being) made to provide interactive tools to support the programming process. These are variously called programmer aids, programmer assistants, intelligent program editors, and programming environments, but they all require the user to program in a standard high level or very high level (VHL) language. They provide one or more programming aids that are implicitly or explicitly invoked to take care of programming details, make checks for errors, make suggestions to the user, etc.

In this vein, PMB might be a useful programming aid if it were provided with a straightforward front-end parser for the surface form of the program fragment language. This parser would generate fragments for PMB to process for completeness and semantic consistency. The structure of such a system is depicted below.

Role of PMB in Intelligent Program Editor



An intelligent program editor along these lines is being developed by Stephen J. Westfold as part of the CHI system [Phillips-79]. This program synthesis system is the successor to the PSI system and incorporates many of the new ideas that arose during the development of PSI and PMB. CHI also has a first implementation of the program reference language discussed in Chapter 5.

The INTERLISP system [Teitelman-78] was the first to provide a good set of low level tools for programmer assistance. These are an editor enforcing LISP syntax; a "prettyprinter" specifically

for LISP constructs; a symbolic debugger; the "Do What I Mean" (DWIM) facility [Teitelman-72A], featuring a spelling corrector, among other niceties; the Programmer's Assistant [Teitelman-72B, Teitelman-77], including a history list of past user activities and commands to manipulate them; and the program analysis package called MASTERSCOPE. SCOPE is an extension of this package [Masinter-79].

The PCM system [Yonke-75] extends many of the INTERLISP notions to block-structured languages such as ALGOL and PASCAL. PCM does incremental parsing and thus quickly discovers syntactic and simple semantic errors (i.e., those determinable by looking up variable properties in the symbol table). There is also a syntax oriented editor for PASCAL [Donzeau-Gouge et al.-75] and one for BDL ("Business Definition Language") that uses a fancy color display [Hammer et al.-74]. A set of routines for manipulating program parse trees within such systems as syntax oriented editors is described in [Robinson & Parnas-73].

Work on programmer apprentices [Winograd-74, Hewitt & Smith-75, Rich & Shrobe-78, Rich et al.-79, Waters-78, Waters-79, Shrobe-79A, Shrobe-79B, Rich-79] is aimed at assisting a programmer with the details and with understanding a program that is being written primarily by the programmer, rather than the system.

3.3 Recognition Problem Solving Using Demons

Demons were invented—or at least named—as a tool for pattern recognition [Selfridge-59]. They have become a common feature of artificial intelligence languages and were popularized as the key mechanism in a story understander [Charniak-72]. PMB's use of demons attached to templates is most similar to the control structure of the Genial Understander System (GUS) [Bobrow et al.-77]. The reasoner portion of GUS is at the back end of a natural language understanding system. The reasoner builds up smaller trees of frames than PMB and never manipulates them once they are completed. Our use of demons also fits nicely into Rieger's general theory of spontaneous computation [Rieger-77].

The successive refinement paradigm is used by the automatic coding phase of PSI, which consists of the PECOS coder [Barstow-79A] and LIBRA efficiency expert [Kant-79A, Kant-79B]. This paradigm appears to have much in common with the basic top down, goal oriented completion of a program model in PMB. But the coding process proceeds from a complete program model and has total control over what subgoal to work on next, whereas acquisition of the model only proceeds in an orderly top down fashion by default. The program acquisition process allows informalities and is usually data driven. Typically the user will jump around from subgoal to subgoal and even create new, unanticipated subgoals (e.g., define previously unreferenced procedures and information structures).

3.4 Programming Methodologies

PMB allows the user to determine the order in which program parts are defined. However, this unrestrictive methodology for program development could be restricted if desirable, e.g., in support of structured programming. Many programming methodologies advocate writing programs in a top down, structured fashion [Dahl et al.-72, Wirth-73]. PMB could support

such orders of program development in two ways. The first way, which is already available, is to trust the user to impose a particular ordering on the fulfillment of subgoals (e.g., all information structures must be defined before they are referenced). The other way would be to modify PMB so that it imposes this ordering.

Most of what PMB does can be viewed as the piecewise transformation of programs. This differs from standard program transformation work [Burstall & Darlington-77, Loveman-77, Kibler et al.-77, Kibler-78, Balzer et al.-76] in that the program being transformed is only a partial program and many of the transformations are designed to remove informalities from the input form. Most program transformation research deals with source-to-source transformations, i.e., equivalence preserving transformations done all within the same language and on complete programs. PMB's rules of canonization are transformations of this sort.

3.5 Very High Level Languages

PMB produces a final program in the program modelling language, which is a very high level language. Its very high level nature stems from its information structures, rather than any fancy control structures. It is most akin to the "set oriented" languages such as SETL [Schwartz-75] and VERS2 [Earley-74]. However, systems dealing with languages such as these concentrate on automatic coding. These systems don't do program acquisition. Therefore, the program must be handwritten by the user.

3.6 Knowledge Representation by Rules and Frames

PMB represents knowledge as both templates and rules.

Static knowledge of model building is stored as a rule base of antecedent/consequent rules [Davis et al.-77, Barstow-79A]. However, PMB's rules are data driven (i.e, triggered by the antecedents being or becoming true), such as in ARS [Stallman & Sussman-77].

Dynamic knowledge (i.e., the partial program model) is stored as a tree of templates, which are similar to structured property lists or frames [Minsky-75, Bobrow & Winograd-77].

3.7 Compiler Technology

A number of ideas in PMB have been borrowed from standard compiler theory and practice. The notion of incrementally processing a program is borrowed from incremental compilers, e.g., PL/ACME [Breitbard & Wiederhold-69]. However, all of these appear to be limited to a grain of incremental processing that is one line of text.

Many compilers handle some of the informalities that PMB does. A common example is type coercion. But compilers do such operations at either compile time or runtime. The philosophy of PMB is that it is better to move such processing from runtime or coding (compile) time to "acquisition time", before the user is out of the picture.

Chapter 4. An Example

This chapter presents in some detail a single example of PMB in action. The example is one of the programs that have been acquired and coded by the PSI program synthesis system.

We start with an informal description of the desired program, called CLASSIFY. Then the inputs and outputs of PSI are shown, to provide context. We present the actual dialog that a human user carried out with the PSI parser/interpreter to specify CLASSIFY. A few program fragments are described; they are the form of program specification that PMB receives from the parser/interpreter. To demonstrate the output of PMB, the completed program model is listed, along with a sample execution by the model interpreter. Finally, the most important (and longest) section has an annotated trace of the program fragments as they are input by PMB from the parser/interpreter and the resulting partial program model as it is built by PMB.

4.1 The CLASSIFY Program

CLASSIFY is a simple program for classifying symbolic patterns. At the heart of the algorithm, CLASSIFY simply tests to see whether one set is a subset of another set. To provide motivation for such a program, an application is described in parentheses along with the abstract algorithm below. The application involves a set of qualifications that are required for a job. The qualifications of one or more job applicants are tested against this set to see if any applicant fits the job requirements.

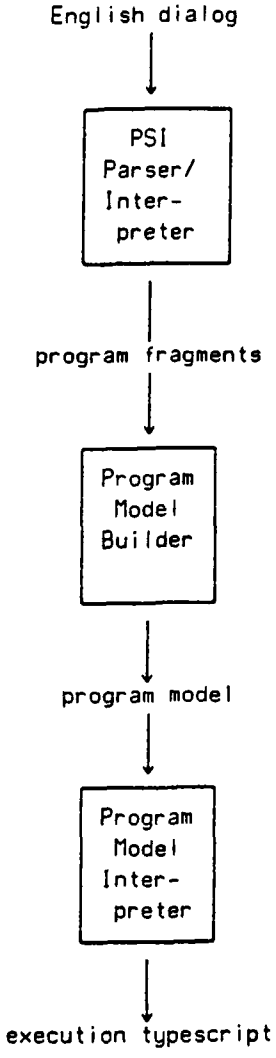
CLASSIFY first inputs a sample set of items (e.g., a list of required job qualifications), where each item is a relation (or n-tuple). Then the program repeatedly inputs a trial set (e.g., the qualifications of an applicant), tests whether the sample set (required qualifications) is a subset of the trial set (applicant's qualifications), and prints "Fit" or "Didn't fit" accordingly. The program is exited if the word "quit" is typed instead of a trial set.

CLASSIFY is typical of the size and complexity of programs that can be built by PMB. CLASSIFY was chosen for this example because it makes use of a wide variety of modelling language features, including all the control structures and all information structures but the correspondence (or mapping). Because of this diversity, CLASSIFY requires the use of much of PMB's knowledge of program models and many of its special techniques for handling completeness, consistency, type coercion, canonization, etc.

4.2 Example Inputs and Outputs

The following diagram shows the relationship between the four inputs and outputs that are presented in this section. The fragments and resultant model are germane to the thesis, but the English dialog with the user and execution trace of the model are also given to provide additional context.

Relationship of Inputs and Outputs Used in Example



4.2.1 English Specification Dialog with PSI

The following dialog was held with PSI to specify CLASSIFY. Sentences from the user were processed by the PSI parser/interpreter [Ginsparg-78]. Questions from PSI were generated by the PSI explainer [Green et al.-79]. The choice of what question to ask next was made by the PSI dialog moderator [Steinberg-79].

Specification Dialog for CLASSIFY

PSI: What is the name of the program you want to write?

User: CLASSIFY.

PSI: Does CLASSIFY have a special program type?

User: Yes, classification.

PSI: Please describe CLASSIFY.

User: CLASSIFY first inputs a concept. Then it repeatedly accepts an input scene from the user, classifies it based on whether or not it fits the concept, and outputs this classification to the user.

PSI: What is the structure of the concept?

User: The concept is a set of relations, each of which is a property name followed by the names of the objects over which it holds.

PSI: What is the structure of the scene?

User: The same as the concept.

PSI: Please define what it means for a scene to fit the concept.

User: A scene fits the concept if all of the relations in the concept are in the scene.

PSI: What is the exit test of the loop?

User: Stop when the user types "quit" instead of an input scene.

PSI: What is the format of the classification output?

User: Output "Fit" if the scene fit the concept, "Didn't fit" if it did not.

PSI: Are we finished?

User: Yes.

4.2.2 Program Fragments Input to PMB

Since all of the 32 fragments generated from the preceding dialog will be given in the detailed trace of model building later in this chapter, only an example of each basic type (control structure, primitive operation, and information structure) is given here.

The first example fragment arises from the following sentences in the preceding dialog:

> CLASSIFY first inputs a concept. Then it repeatedly accepts an input scene from the user, classifies it based on whether or not it fits the concept, and outputs this classification to the user.

Actually the fragment below is only the first of a number of fragments that are gleaned from these two sentences. A paraphrase of the fragment in terms of the above sentences is "CLASSIFY *first* [does something]. Then it [does something else].".

```
algorithm_body:
   begin
      input_concept;
      classify_loop
   end
```

The fragment above defines a control structure called by the unique name "algorithm_body"[1]. It is a **composite** (similar to a compound statement in ALGOL) with two parts, called "input_concept" and "classify_loop". These are arbitrary names representing the two things that CLASSIFY is supposed to do. Notice that this fragment neither defines where "algorithm_body" is to be invoked in the algorithm of CLASSIFY, nor defines what "input_concept" and "classify_loop" are, nor specifies whether the two parts are to be executed sequentially or in parallel.

The second fragment arises from the part of this sentence that precedes the word "instead": "*Stop when the user types 'quit' instead of an input scene.*".

exit_condition: *input_data_prototype = quit_prototype*

This fragment is a primitive operation called "exit_condition", which tests for equality between two information structures, *input_data_prototype* and *quit_prototype*. Once again, other fragments specify where this test is to be made. The operator is a generic equality condition that is specialized based upon the types of its two arguments.

The final example comes from the part of this sentence before the comma: "The concept is a set of relations, each of which is a property name followed by the names of the objects over which it holds.":

type *concept_prototype:* **set of** *relation_prototype*

This fragment declares an information structure called *concept_prototype*. *Concept_prototype* is an unordered **collection** without repetition of elements. The prototypic element of the **collection** is a *relation_prototype*. The size of the **collection**, where it is referenced, and the definition of *relation_prototype* are not provided in this fragment.

---

[1] Mnemonic names have been assigned wherever possible to avoid using the computer symbols generated by PSI.

4.2.3 Program Model Output by PMB

Below is the program model of CLASSIFY that PMB produced. It was printed using a PASCAL-like syntax by the readable program model generator [Pressburger-78]. Information structures have both prototypes (defined after the keyword type below) and instances of the prototypes (defined after the keyword var). All cross-references, assertions, and other nonessential annotations have been omitted for clarity. The program modelling language, from which listings such as this are derived, is defined in Chapter 7.

One interesting feature of this model is *input_data_prototype*, which is an information structure prototype that is an alternative of two other prototypes. This is a mutually exclusive selector, so that an instance of *input_data_prototype* has to be of one type or the other, either a *scene_prototype* or the string "quit". This construct is useful in recognizing nonstandard data that mark the end of processing.

CLASSIFY Program Model

```
program classify;

    type
        input_data_prototype: alternative of {scene_prototype, quit_prototype},
        scene_prototype: set of relation_prototype,
        concept_prototype: set of relation_prototype,
        relation_prototype: plex of <relation_name: string, arguments: list of string>,
        quit_prototype: string = "quit";

    var
        input_data: input_data_prototype,
        scene, fit_scene: scene_prototype,
        concept, fit_concept: concept_prototype,
        fit_result: Boolean;

    procedure fit(fit_scene, fit_concept): Boolean;
        fit_concept ⊆ fit_scene;

    begin
        concept ← input(concept_prototype, user, "Ready for concept");
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, "Ready");
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    scene ← input_data;
                    fit_result ← fit(scene, concept);
                    case
                        fit_result: inform_user("Fit");
                        ¬fit_result: inform_user("Didn't fit")
                    endcase
                end
            finally
                exit:
        endloop
    end
```

Program Model Structure

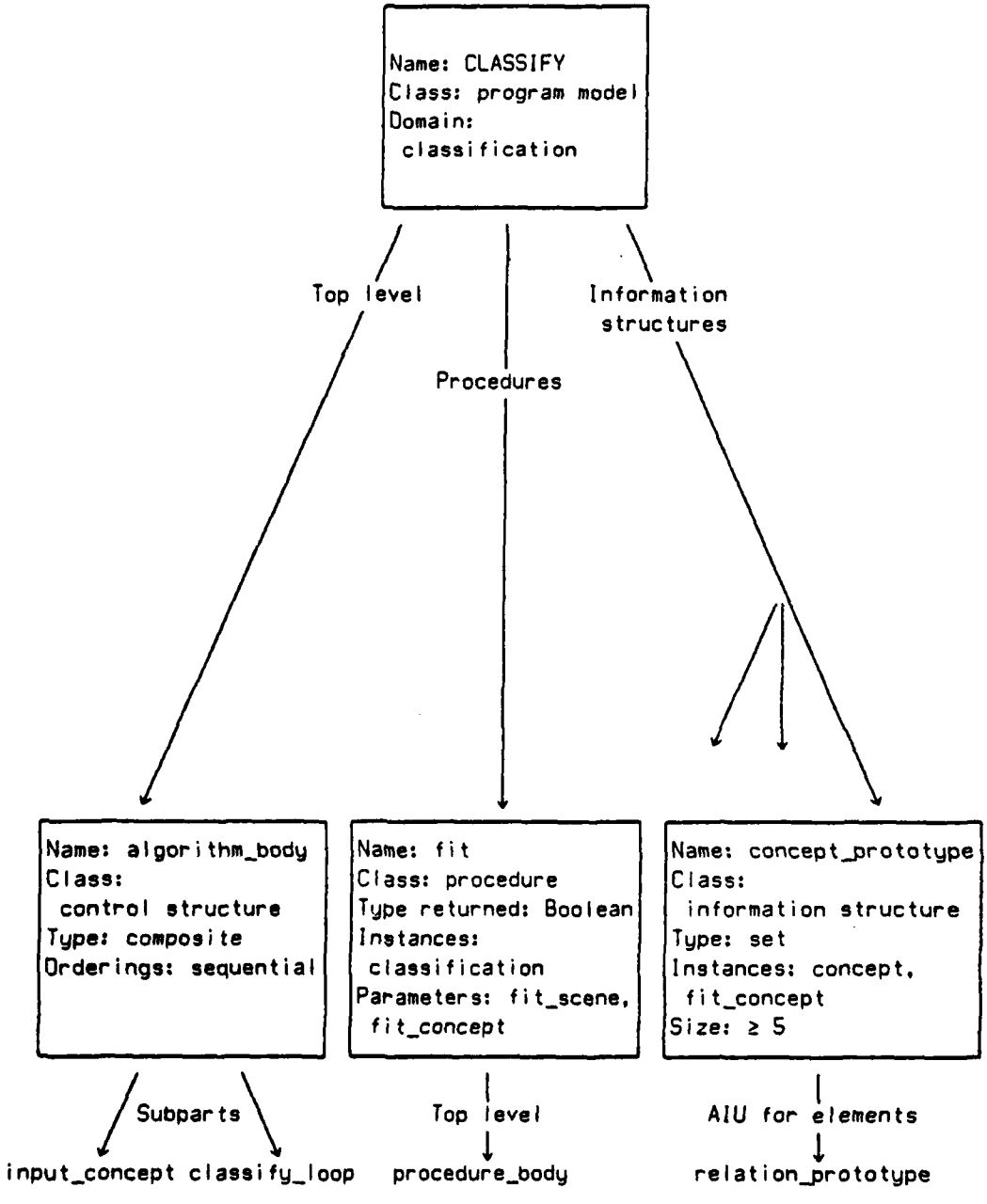
The actual program model is a tree[2] of templates, each containing a number of slots and associated values. The top level of the CLASSIFY model is shown below.

Each template has a *class*, and most classes have a particular *type*. The remaining required or optional slots depend upon the class and type. Some slots take simple values, and some point to other templates.

---

[2] More accurately a graph because of the cross-reference pointers that are added to the basic top down program structure

Internal Structure of Prograin Model



4.2.4 Typescript of Sample Interpreted Execution

Below is one execution of the CLASSIFY program model shown in the preceding section. CLASSIFY was executed interpretively by the program model interpreter [Nelson-76], which always uses the same default data structure implementations. The example used in this run is the job classification task discussed earlier. Here the minimum qualifications for the job we have in mind are lots of programming experience and the enjoyment of LISP over ALGOL. Other qualifications (e.g., an interest in artificial intelligence) are optional.

Typescript of CLASSIFY Execution

```
←INTERPRET(CLASSIFY)

Initializing CLASSIFY ...

Entering Interpreter at ALGORITHM-BODY:

READY FOR CONCEPT
*((ENJOYS-MORE (LISP ALGOL))(PROGRAMMING-EXPERIENCE (MUCH)))

READY
*((ENJOYS-MORE (LISP ALGOL)))

DIDN'T FIT

READY
*((PROGRAMMING-EXPERIENCE (MUCH)))

DIDN'T FIT

READY
*((ENJOYS-MORE (APL LISP))(PROGRAMMING-EXPERIENCE (MUCH)))

DIDN'T FIT

READY
*((ENJOYS-MORE (LISP ALGOL))(PROGRAMMING-EXPERIENCE (LITTLE)))

DIDN'T FIT

READY
*((ENJOYS-MORE (LISP ALGOL))(PROGRAMMING-EXPERIENCE (MUCH)))

FIT

READY
*((ENJOYS-MORE (LISP ALGOL))(INTEREST (AI))(PROGRAMMING-EXPERIENCE (MUCH)))

FIT

READY
*QUIT

Exiting Interpreter ...
```

```
OKAY
←
```

4.3 Annotated Trace of Model Building

This section presents an annotated trace of PMB building the program model for CLASSIFY. Included in the trace are all 32 program fragments in the order they were received by PMB from the PSI parser/interpreter, an English description of any important or novel processing that took place for each fragment, and a snapshot of the partial program model and outstanding demons after each group of related fragments was processed. Most other details of the original trace (e.g., explicit rule invocations) have been left out for the sake of brevity and clarity.

This example provides a good demonstration of PMB's control structure because the order in which fragments arrive is particularly perverse: All of the information structure declarations arrive *after* the algorithm itself.[3] This forces PMB to create many demons to wait for the information structures to be defined before consistency checks, type coercion, etc., can be completed. Also, instead of the one procedure being defined before the main algorithm, the procedure is defined in the middle of the top down definition of the main algorithm, immediately after the procedure is called. Although information structures and procedures can be processed in any order, the most efficient order for PMB to process fragments is all information structures first, then all procedures, and the main algorithm body last.[4] But information structures and procedures may be interspersed throughout the main algorithm body in any order.

PMB's control structure is a loop that inputs and then processes program fragments. The processing of the first two fragments is traced in greater detail than the rest, in order to show PMB focusing on individual slots of a fragment, creating model templates, and creating subgoals. The discussion of later fragments emphasizes the inference of pieces of program and the creation and triggering of demons. Although only fourteen demons are discussed, dozens of other, more mundane ones are created and executed during the building of CLASSIFY.

Comments about the trace are indented. Demons are assigned unique identification numbers from 1 through 14 in the same order as demon creation. Fragments and models are printed in a PASCAL-like notation, although they are maintained internally as property lists, as evidenced below in the low level trace of processing. Most computer generated symbols have been replaced by more mnemonic words corresponding to the program model presented in an earlier section. Each partial program model is printed at the top of a new page so that the incremental building up of the model can be more easily discerned. Lines that are new or have changed from the immediately preceding model and demon list are denoted by the character "|" at the right margin. Missing parts of the partial program model that are still to be filled in by later fragments are denoted by "???".

---

[3] Note that in this example the parser/interpreter did not generate fragments in precisely the same order as their contents were specified in the English dialog.

[4] This observation of a computer model correlates well with one typical programming style.

Annotated Trace of Building CLASSIFY Model

The first fragment simply names the program to be written and the top level algorithm.

```
Inputting fragment:

program classify;
   algorithm_body
```

PMB is initialized with the goal of wanting to know the name of the program model. The first fragment fulfills this goal, so the root template of the program model tree is created and given the name CLASSIFY. A subgoal is created to determine its top level algorithm. The domain of the CLASSIFY program is an example of a simple default value. The default value ("unknown") is stored in the program model, but a subgoal (question) is also created to allow the default to be overridden. This subgoal is marked as optional since a valid answer is already known.

```
Processing PROGRAM-MODEL.NAME = CLASSIFY
    Creating template CLASSIFY with value
        CLASSIFY.CLASS = PROGRAM-MODEL
        CLASSIFY.DOMAIN = UNKNOWN
    Creating subgoal:
        CLASSIFY.DOMAIN = ???
    Creating subgoal:
        CLASSIFY.TOP-LEVEL = ???
Done processing PROGRAM-MODEL.NAME = CLASSIFY
```

The name of the top level of the algorithm occurs in the same fragment. Since there is an outstanding subgoal that wants to determine the top level, it is processed next. Since a template named "algorithm_body" doesn't exist already, a new template is created and a two way pointer between it and the CLASSIFY template is inserted. Because the new template represents the top level algorithm of the model, PMB assigns it a class of operational unit. An operational unit can be any control structure, primitive operation, or procedure call. Then PMB asks for what specific type within this class the template is.

```
Processing CLASSIFY.TOP-LEVEL = ALGORITHM-BODY
    Creating template ALGORITHM-BODY with value
        ALGORITHM-BODY.CLASS = OPERATIONAL-UNIT
    Creating subgoal:
        ALGORITHM-BODY.TYPE = ???
Done processing CLASSIFY.TOP-LEVEL = ALGORITHM-BODY
```

In partial program models, the first of which is given below, names preceding colons are unique template names (analogous to statement labels in ALGOL) that allow fragments to refer to different parts of the model. Besides the explicit question about the top level algorithm of CLASSIFY (denoted below by "???"), there are always two implicit (i.e., set up internally by PMB and always present) questions that will handle definitions of information structures and procedures. These questions aren't shown.

```
Current program model:

program classify;

    algorithm_body: ???

Current demons active:
    None
```

The second fragment describes the top level algorithm as a composite of two steps.

```
Inputting fragment:

algorithm_body:
   begin
      input_concept;
      classify_loop
   end
```

A **composite** is a compound statement with a partial ordering on the execution of its subparts. The partial ordering is optional and defaults to sequential.

```
Processing ALGORITHM-BODY.TYPE = COMPOSITE
    Creating subgoal:
        ALGORITHM-BODY.SUBPARTS = ???
    Creating subgoal:
        ALGORITHM-BODY.ORDERINGS = ???
Done processing ALGORITHM-BODY.TYPE = COMPOSITE
```

The two subparts must be operational units because they are the steps of a **composite**. However, their specific types (e.g., *input*, test) are still unknown.

```
Processing ALGORITHM-BODY.SUBPARTS = (INPUT-CONCEPT CLASSIFY-LOOP)
    Creating template INPUT-CONCEPT with value
        INPUT-CONCEPT.CLASS = OPERATIONAL-UNIT
    Creating subgoal:
        INPUT-CONCEPT.TYPE = ???
    Creating template CLASSIFY-LOOP with value
        CLASSIFY-LOOP.CLASS = OPERATIONAL-UNIT
    Creating subgoal:
        CLASSIFY-LOOP.TYPE = ???
Done processing ALGORITHM-BODY.SUBPARTS = (INPUT-CONCEPT CLASSIFY-LOOP)
```

After this point, we won't show the details of PMB processing particular slots of fragments, creating templates, and creating subgoals. Rather, we will emphasize the inference of pieces of program and the creation and triggering of demons. Of the subgoals remaining at this point, the two in brackets below are implicit.

```
Subgoals still remaining:
     CLASSIFY.DOMAIN = ???
     ALGORITHM-BODY.ORDERINGS = ???
     INPUT-CONCEPT.TYPE = ???
     CLASSIFY-LOOP.TYPE = ???
     [CLASSIFY.INFORMATION-STRUCTURES = ???]
     [CLASSIFY.PROCEDURES = ???]
```

The two required questions regarding the types of "input_concept" and "classify_loop" also appear below in the current program model.

```
Current program model:

program classify;

   begin
      input_concept: ???;
      classify_loop: ???
   end

Current demons active:
    None
```

The next fragment defines the *input* primitive operation that reads in the *concept*. The three arguments of an *input* operation are the prototype of the information structure to be input, the source of the input, and a prompt string to be output just prior to input.

```
Inputting fragment:

input_concept: input(concept_prototype, user, concept_prompt)
```

PMB infers that the object of type *concept_prototype* should be saved in an instance of that type. So it creates one that is called *concept* here for mnemonic reasons and puts the *input* inside a *remember* operation (denoted by "←" below).

Current program model:

```
program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        classify_loop: ???
    end
```

Current demons active:
None

Inputting fragment:

```
classify_loop:
   until exit (exit_condition)
      repeat loop_body
      finally exit:
   endloop
```

The above fragment defines a loop with a jump out of "loop_body" to exit block "exit" when the Boolean condition "exit_condition" is true.

At this point PMB can't tell where "exit_condition" is located, so it is shown separately from the main algorithm. Demon 1 is created to make sure that "exit_condition" is contained within the body of the loop. Since the context of "exit_condition" within the algorithm isn't known yet, the demon is set up to await the definition of the control structure that contains "exit_condition". In general, a demon may require the values of many undefined slots in order to evaluate its antecedents. However, a demon is implemented "linearly", i.e., it only waits for one slot at a time, moving from slot to slot until all required slot values are defined.

Demon 2 is created to put the exit condition inside a test with an *assert_exit_condition* as its true branch. This will cause the loop to be exited when the exit condition becomes true.

```
Current program model:

program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                loop_body: ???
            finally
                exit:
        endloop
    end

    exit_condition: ???

Current demons active:
    Demon 1:  awaiting control structure containing "exit_condition"
    Demon 2:  awaiting control structure containing "exit_condition"

Inputting fragment:

loop_body:
    begin
        loop_input;
        exit_condition;
        classification;
        output_classification
    end
```

Since the location of "exit_condition" within the algorithm is now defined, Demon 1 is triggered and finds that "exit_condition" is within the composite "loop_body". Since this is not the loop that Demon 1 was hoping to find, it creates a new instance of itself to await the definition of what control structure "loop_body" is inside. Since this is already known, the new instance of the demon doesn't have to wait. It immediately discovers that "loop_body" is inside the desired loop and thus, by transitivity, so is "exit_condition". Now Demon 1 is destroyed, as are most demons after they succeed.

Demon 2 creates a test with "exit_condition" as its predicate and an *assert_exit_condition* that will leave the loop as its true action. Then Demon 2 goes out of existence.

Current program model:

```
program classify;

    type
        concept_prototype: ???,
        concept_prompt: string = ???;

    var
        concept: concept_prototype;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    loop_input: ???;
                    if exit_condition: ??? then assert_exit_condition(exit);
                    classification: ???;
                    output_classification: ???
                end
            finally
                exit:
        endloop
    end
```

Current demons active:
None

Inputting fragment:

loop_input: *input*(*input_data_prototype*, user, *input_data_prompt*)

The *input* fragment above is handled similarly to the previous *input*.

Inputting fragment:

exit_condition: *input_data_prototype* = *quit_prototype*

Demon 3 is created to specialize the generic operator *are_equal* into the appropriate primitive operation, depending on the types of its arguments. In this case, Demon 3 specializes *are_equal* into an *is_of_type* when *input_data_prototype* is determined to be an information structure alternative and *quit_prototype* one of its subalternatives. To succeed, this demon has to wait for the definition of *input_data_prototype*.

Inputting fragment:

classification: *fit*(*scene_prototype*, *concept_prototype*)

PMB infers that a procedure named *fit* exists. Demon 4 will ensure that there are the same number of actual and formal parameters when *fit* is defined.

A disambiguation of the second argument occurs at this point. Since *concept_prototype* is known to be an information structure prototype, it can't be used as the argument in a procedure call. Instead, PMB replaces it in the call with the only instance of *concept_prototype*, *concept*. In addition, *concept* is designated the "primary instance" of *concept_prototype*, to be used if a similar situation should arise in the future.

In contrast, *scene_prototype* is mentioned for the first time in the current fragment. Since PMB can't see into the future, *scene_prototype* is assumed to be an instance now. Later, *scene_prototype* will be defined as a prototype, and an instance will be created for it then.

Demons 5 and 6 will ensure that the types of the two actual parameters, *scene_prototype* and *concept*, are in agreement with those of the two formal parameters. If a type is not specified for a formal parameter, then its type will be coerced by (inherited from) the corresponding actual parameter.

Demon 7 is created to coerce the type returned by the *procedure_instance* (procedure call) to be the same as that returned by the procedure definition.

```
Current program model:

program classify;

    type
        input_data_prototype: ???,
        concept_prototype: ???,
        input_data_prompt: string = ???,
        concept_prompt: string = ???,
        quit_prototype: ???;

    var
        input_data: input_data_prototype,
        scene_prototype: ???,
        concept: concept_prototype;

    procedure fit(???): ???;
        ???;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, input_data_prompt);
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    fit(scene_prototype, concept);
                    output_classification: ???
                end
            finally
                exit:
        endloop
    end

Current demons active:
    Demon 3:  awaiting type of input_data_prototype
    Demon 4:  awaiting formal parameters of fit
    Demon 5:  awaiting formal parameters of fit
    Demon 6:  awaiting formal parameters of fit
    Demon 7:  awaiting type returned by fit
```

Now the top down exposition of the main algorithm is interrupted in order to define the *fit* procedure, which is called in the previous fragment.

```
Inputting fragment:

procedure fit(fit_scene, fit_concept): Boolean;
  procedure_body
```

Demon 4 succeeds when it finds that the number of formal parameters of *fit* is the same as the number in the call to it.

PMB infers that the two formal parameters are the names of information structure instances, and creates them. Demons 5 and 6, the type coercers, are moved ahead to wait for the definitions of the prototypes of the actual parameters, *scene_prototype* and *concept*, so that type coercion can be done on the corresponding formal parameters, which are now known.

Demon 7 stores the type of procedure *fit*, Boolean, into the *procedure_instance*.

Inputting fragment:

procedure_body: (∀ *relation_prototype*) true_for_all_body

This is a *true_for_all* Boolean test, which determines whether some condition is true for all elements of a collection. Demon 8 is created to canonize *true_for_all* into an *is_subset* if this becomes appropriate based on the definition of "true_for_all_body".

Inputting fragment:

true_for_all_body: antecedent_is_element ⊃ consequent_is_element

Demon 8 moves ahead to await the definition of "antecedent_is_element".

Inputting fragment:

antecedent_is_element: *relation_prototype* ∈ *fit_concept*

Demon 8 notices that "antecedent_is_element" is an *is_element* test whose element argument is *relation_prototype*, which is the referent variable of the *true_for_all*. So Demon 8 moves ahead to await the definition of *fit_concept*.

Demon 9 is created to guarantee type consistency between *relation_prototype*, the element of the *is_element*, and the as yet unknown prototypic element of the collection *fit_concept*.

Inputting fragment:

consequent_is_element: *relation_prototype* ∈ *fit_scene*

This *is_element* fragment is handled similarly to the previous one. Demon 10 is set up to check for type consistency.

```
Current program model:

program classify;

    type
        input_data_prototype: ???,
        concept_prototype: ???,
        input_data_prompt: string = ???,
        concept_prompt: string = ???,
        quit_prototype: ???;

    var
        input_data: input_data_prototype,
        scene_prototype: ???,
        fit_scene: ???,
        concept: concept_prototype,
        fit_concept: ???,
        relation_prototype: ???;

    procedure fit(fit_scene, fit_concept): Boolean;
        ∀ relation_prototype | relation_prototype ∈ fit_concept ⊃ relation_prototype ∈ fit_scene;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, input_data_prompt);
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    fit(scene_prototype, concept);
                    output_classification: ???
                end
            finally
                exit:
        endloop
    end

Current demons active:
    Demon 3:  awaiting type of input_data_prototype
    Demon 5:  awaiting prototype of scene_prototype
    Demon 6:  awaiting prototype of concept
    Demon 8:  awaiting prototype of fit_concept
    Demon 9:  awaiting prototype of relation_prototype
    Demon 10:  awaiting prototype of relation_prototype

Inputting fragment:

output_classification:
    case
        fit_true;
        fit_false
    endcase
```

```
Inputting fragment:
```

fit_true: *fit_result:* print_fit

```
Inputting fragment:
```

print_fit: *output*(*fit_prototype*, user)

> Demon 11 is created to transform the *output* into the simpler *inform_user* operation, if *fit_prototype* turns out to be a string constant.

```
Inputting fragment:
```

fit_false: not_fit_result: print_didn't_fit

```
Inputting fragment:
```

not_fit_result: ¬*fit_result*

```
Inputting fragment:
```

print_didn't_fit: *output*(*didn't_fit_prototype*, user)

> Demon 12 is created to transform the *output* into an *inform_user* operation, if *didn't_fit_prototype* is a string constant.

Current program model:

```
program classify;

    type
        input_data_prototype: ???,
        concept_prototype: ???,
        input_data_prompt: string = ???,
        concept_prompt: string = ???,
        quit_prototype: ???,
        fit_prototype: ???,
        didn't_fit_prototype: ???;

    var
        input_data: input_data_prototype,
        scene_prototype: ???,
        fit_scene: ???,
        concept: concept_prototype,
        fit_concept: ???,
        relation_prototype: ???;

    procedure fit(fit_scene, fit_concept): Boolean;
        ∀ relation_prototype | relation_prototype ∈ fit_concept ⊃ relation_prototype ∈ fit_scene;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, input_data_prompt);
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    fit(scene_prototype, concept);
                    case
                        fit_result: ???: output(fit_prototype, user);
                        ¬fit_result: ???: output(didn't_fit_prototype, user)
                    endcase
                end
            finally
                exit:
        endloop
    end
```

Current demons active:

```
    Demon 3:  awaiting type of input_data_prototype
    Demon 5:  awaiting prototype of scene_prototype
    Demon 6:  awaiting prototype of concept
    Demon 8:  awaiting prototype of fit_concept
    Demon 9:  awaiting prototype of relation_prototype
    Demon 10:  awaiting prototype of relation_prototype
    Demon 11:  awaiting type of fit_prototype
    Demon 12:  awaiting type of didn't_fit_prototype
```

Finally PMB receives the fragments that define information structure prototypes. Now many demons that were set up earlier in the trace are fired off and succeed with their appointed checks and transformations.

An instance of the alternative *input_data_prototype* is at any time an instance of exactly one of the alternative prototypes, *scene_prototype* or *quit_prototype*.

Inputting fragment:

type *input_data_prototype:* alternative of {*scene_prototype, quit_prototype*}

Input_data_prototype is defined as an alternative prototype, and *quit_prototype* already is a prototype. *Scene_prototype* should either already be a prototype or be defined as one now. But *scene_prototype* was previously defined as an information structure instance. So PMB copies this instance to a new template, which will be called *scene*, that is created by PMB. All pointers to the instance that used to point to *scene_prototype* from other templates are updated to point to *scene*. Then PMB creates the prototype in its place. *Scene* is marked as the primary instance of *scene_prototype*.

Now that *scene_prototype* is known to be the prototype of *scene*, one of the two type coercion demons, Demon 5, coerces *fit_scene* to be of the same type as *scene* by making *fit_scene* be an instance of *scene_prototype* too.

Demon 3 specializes the *are_equal* operator into an *is_of_type*, which is specifically for testing which option an instance of an alternative is.[5]

Demon 13 is created to guarantee that *quit_prototype* is in the tree of alternatives for *input_data_prototype*. It is, so Demon 13 succeeds.

Demon 14 is created to insert a *select_alternative* operation (denoted by the "←" operator) after the *is_of_type*. It succeeds because there are only two alternatives, and if *input_data* isn't of type *quit_prototype*, then it must be of type *scene_prototype*.

Inputting fragment:

type *quit_prototype:* string = "quit"

[5] The "=" symbol is still used to denote the operation in the program model.

Current program model:

```
program classify;

    type
        input_data_prototype: alternative of {scene_prototype, quit_prototype},
        scene_prototype: ???,
        concept_prototype: ???,
        input_data_prompt: string = ???,
        concept_prompt: string = ???,
        quit_prototype: string = "quit",
        fit_prototype: ???,
        didn't_fit_prototype: ???;

    var
        input_data: input_data_prototype,
        scene, fit_scene: scene_prototype,
        concept: concept_prototype,
        fit_concept: ???,
        relation_prototype: ???;

    procedure fit(fit_scene, fit_concept): Boolean;
        ∀ relation_prototype | relation_prototype ∈ fit_concept ⊃ relation_prototype ∈ fit_scene;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, input_data_prompt);
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    scene ← input_data;
                    fit(scene, concept);
                    case
                        fit_result: ???: output(fit_prototype, user);
                        ¬fit_result: ???: output(didn't_fit_prototype, user)
                    endcase
                end
            finally
                exit:
        endloop
    end
```

Current demons active:

```
    Demon 6:  awaiting prototype of concept
    Demon 8:  awaiting prototype of fit_concept
    Demon 9:  awaiting prototype of relation_prototype
    Demon 10:  awaiting prototype of relation_prototype
    Demon 11:  awaiting type of fit_prototype
    Demon 12:  awaiting type of didn't_fit_prototype
```

Inputting fragment:

type *didn't_fit_prototype:* string = "Didn't fit"

At this point Demon 12 transforms the *output* operation that prints out *didn't_fit_prototype* into an *inform_user* that prints the string constant "Didn't fit".

Inputting fragment:

type *fit_prototype:* string = "Fit"

Demon 11 transforms the other *output* into an *inform_user*.

Inputting fragment:

type *relation_prototype:* plex of <relation_name: *relation_name*, arguments: *arguments*>

Relation_prototype is already defined as an instance, so the instance is copied to a new template called *relation*, and the new prototype *relation_prototype* takes its place.

The *is_element* consistency demons, Demons 9 and 10, now know what the element slots of the *is_element* operations are, but they must still wait for the definitions of the prototypic elements of their collection slots.

Inputting fragment:

type *relation_name:* atom

Atoms are treated equivalently to strings by PMB.

Inputting fragment:

type *argument:* atom

Argument is an unknown template name because it is defined before *arguments* and isn't otherwise referenced by any fragment. So PMB creates a new information structure prototype and assumes it will get referenced later.

Inputting fragment:

type *arguments:* list of *argument*

Argument is referenced here.

Current program model:

```
program classify;

    type
        input_data_prototype: alternative of {scene_prototype, quit_prototype},
        scene_prototype: ???,
        concept_prototype: ???,
        relation_prototype: plex of <relation_name: string, arguments: list of string>,
        input_data_prompt: string = ???,
        concept_prompt: string = ???,
        quit_prototype: string = "quit";

    var
        input_data: input_data_prototype,
        scene, fit_scene: scene_prototype,
        concept: concept_prototype,
        fit_concept: ???,
        relation: relation_prototype;

    procedure fit(fit_scene, fit_concept): Boolean;
        ∀ relation | relation ∈ fit_concept ⊃ relation ∈ fit_scene;

    begin
        concept ← input(concept_prototype, user, concept_prompt);
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, input_data_prompt);
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    scene ← input_data;
                    fit(scene, concept);
                    case
                        fit_result: ???: inform_user("Fit");
                        ¬fit_result: ???: inform_user("Didn't fit")
                    endcase
                end
            finally
                exit:
        endloop
    end

Current demons active:
     Demon 6:  awaiting prototype of concept
     Demon 8:  awaiting prototype of fit_concept
     Demon 9:  awaiting prototype of fit_concept
     Demon 10:  awaiting scene_prototype

Inputting fragment:

type concept_prompt: string = "Ready for concept"
```

Inputting fragment:

type *input_data_prompt:* string = "Ready"

Inputting fragment:

fit_result ← classification

A Boolean information structure instance called *fit_result* is created and the result of the classification test is remembered in it.

Inputting fragment:

type *scene_prototype:* set of *relation_prototype*

Now Demon 10, the *is_element* consistency check, succeeds because the prototypic element of the collection slot of the *is_element* is now defined and matches the type of the element slot.

Current program model:

```
program classify;

    type
        input_data_prototype: alternative of {scene_prototype, quit_prototype},
        scene_prototype: set of relation_prototype,
        concept_prototype: ???,
        relation_prototype: plex of <relation_name: string, arguments: list of string>,
        quit_prototype: string = "quit";

    var
        input_data: input_data_prototype,
        scene, fit_scene: scene_prototype,
        concept: concept_prototype,
        fit_concept: ???,
        relation: relation_prototype,
        fit_result: Boolean;

    procedure fit(fit_scene, fit_concept): Boolean;
        ∀ relation | relation ∈ fit_concept ⊃ relation ∈ fit_scene;

    begin
        concept ← input(concept_prototype, user, "Ready for concept");
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, "Ready");
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    scene ← input_data;
                    fit_result ← fit(scene, concept);
                    case
                        fit_result: inform_user("Fit");
                        ¬fit_result: inform_user("Didn't fit")
                    endcase
                end
            finally
                exit:
        endloop
    end
```

Current demons active:

```
    Demon 6:  awaiting prototype of concept
    Demon 8:  awaiting prototype of fit_concept
    Demon 9:  awaiting prototype of fit_concept
```

Inputting fragment:

```
type concept_prototype: set of relation_prototype
```

Now that the prototype of *concept* is defined, the second type coercion demon, Demon 6, coerces *fit_concept* to be of the same type as *concept* by making *fit_concept* be an instance of *concept_prototype* too.[6]

Knowing that *fit_concept* is an instance of *concept_prototype* also allows Demons 8 and 9 to proceed. Demon 9, one of the two *is_element* consistency checks from the *fit* procedure, succeeds because the prototypic element of the collection slot of the *is_element* is now defined and matches the type of the element slot, *relation_prototype*.

Finally, all information structures referenced in the procedure body are defined and all consistency demons have succeeded. So Demon 8 canonizes the *true_for_all* by transforming it, along with the *implies* and the two *is_elements* it contains, into an *is_subset*.[7]

```
Program model complete.
```

At the end of model building, 37 optional questions and no required questions are left, so CLASSIFY is complete. The optional questions that remain are mostly about information structure prototypes that have already been inferred, sizes of collections, probabilities of conditions, optional constant values of primitives, etc.

Although all fourteen demons that have been tracked throughout the trace are gone, a few other demons are left alive at the end of model building. Some consistency demons were designed to watch for certain conditions forever. For example, when the *true_for_all* Boolean test was created earlier, a group of demons was created to guarantee that the referent information structure of the *true_for_all* wasn't ever referenced outside the body of the *true_for_all*. Sometimes transformation demons are created to watch for possible canonizations, but the antecedents of these demons are never satisfied. There weren't any such transformation demons in this example.

---

[6] This demon could also have been designed to succeed earlier rather than waiting for *concept_prototype* to be explicitly defined, since *concept_prototype* was assumed to be the prototype of *concept*.

[7] It is interesting to observe the power of a few additional rules. In addition to the *is_subset* transformation demon, Demon 8, another demon could have been set up to watch the *true_for_all* within procedure *fit*. Both *fit_concept* and *fit_scene* occur as collections in *is_element* tests with the same element argument, *relation*. From this the demon would infer that *fit_concept* and *fit_scene* are collections with the same prototypic element. When PMB's question about the prototypic element of *scene_prototype* was answered, the question about the prototypic element of *concept_prototype* would become superfluous, and vice versa. However, the two collections might still have other characteristics, e.g., one might be ordered and the other unordered.

Final program model:

```
program classify;

    type
        input_data_prototype: alternative of {scene_prototype, quit_prototype},
        scene_prototype: set of relation_prototype,
        concept_prototype: set of relation_prototype,
        relation_prototype: plex of <relation_name: string, arguments: list of string>,
        quit_prototype: string = "quit";

    var
        input_data: input_data_prototype,
        scene, fit_scene: scene_prototype,
        concept, fit_concept: concept_prototype,
        fit_result: Boolean;

    procedure fit(fit_scene, fit_concept): Boolean;
        fit_concept ⊆ fit_scene;

    begin
        concept ← input(concept_prototype, user, "Ready for concept");
        until exit
            repeat
                begin
                    input_data ← input(input_data_prototype, user, "Ready");
                    if input_data = quit_prototype then assert_exit_condition(exit);
                    scene ← input_data;
                    fit_result ← fit(scene, concept);
                    case
                        fit_result: inform_user("Fit");
                        ¬fit_result: inform_user("Didn't fit")
                    endcase
                end
            finally
                exit:
        endloop
    end

Current demons active:
    None
```

Chapter 5. The Input: Program Fragments

Input to PMB is in the form of a sequence of informal pieces of program description called *program fragments*. The program fragment language provides a uniform means of feeding information to PMB from any other program acquisition knowledge source. Fragments are designed to convey small chunks of information corresponding to, for example, all or part of a sentence from a specification dialog. The flexibility of fragments allows a program specification to be given incrementally in an arbitrary order when the user is in control of the specification process.

For a procedure to be considered an algorithm, it must possess the quality of "definiteness". This property requires that each step of the procedure be known and be precisely, rigorously, and unambiguously defined [Knuth-73A, page 5]. An incremental and informal program specification using fragments does not constitute an executable algorithm. Until all of the fragments are assembled, the procedure they specify is incomplete. In addition, it is not well-defined if any informalities remain.

A fragment consists of two parts: *what* is to be done to the program model that is under construction (i.e., defining or modifying some part) and *where* this action is to take place (i.e., which part of the model is to be affected). Specification of the "where" part is limited to explicitly naming unique points in the model now, but a language for referring to the parts of a program has been designed. We will discuss the format of fragments and then their content, in terms of what is to be done and where.

5.1 Format of Fragments

To understand fragments, one must understand the program model, which they are used to specify. By the end of model building, the program model constitutes a computer program that is (1) executable; (2) written at a very high level (VHL); and (3) made up of information structure definitions, procedure declarations, a main algorithm consisting of control structures and primitive operations, and assertions. The model is represented as a parse tree. Elements of the tree are *templates*, which define the program in a hierarchy of information structures, control structures, and primitive operations. Each template consists of a set of slot/value pairs. The value of the *type* slot (e.g., loop) determines the other slots that are required or optional. Depending on the slot, a slot value may be such things as a string, a pointer to another template, or a list structure that points to more than one template.

A fragment specifies a template in the model and one or more slot/value pairs of that template. In the most perverse case, the fragments specify templates in an arbitrary order, and each fragment contains only one new piece of information about a template. In the most straightforward case, the fragments specify the templates in a top down traversal (either depth or breadth first), and the information about each template is contained in a single fragment. Of course, the typical sequence of fragments lies somewhere between these extremes. For example, an information structure may be described first, followed by the algorithm that uses it, or vice versa. A procedure may be defined before or after the *procedure_instances* that call it.

As an example, we give the two extremes for fragments that specify (1) the prototype of a set

(an unordered collection without repetitions) of *relations* and (2) a specific instance of an operation that checks if *relation* is an element of the collection *concept*. First are their definitions in just two fragments.

Minimum Number of Fragments

```
where:   name: relations
what:    type: collection
         ordered: nil
         repetitions: nil
         AIU_for_elements: relation

where:   name: membership_test
what:    type: is_element
         element: relation
         collection: concept
```

Following is the same information expressed in a number of fragments in a nearly arbitrary order.[1]

Maximum Number of Fragments

```
where:   name: relations
what:    type: collection

where:   name: membership_test
what:    type: is_element

where:   name: relations
what:    AIU_for_elements: relation

where:   name: relations
what:    repetitions: nil

where:   name: membership_test
what:    collection: concept

where:   name: membership_test
what:    element: relation

where:   name: relations
what:    ordered: nil
```

No formal experiments have been performed to determine the adequacy of program fragments for specifying programs in the most desirable ways. However, it is clear that fragments come a long way toward providing the necessary flexibility. Fragments provide a means for specifying parts of a program in the minutest detail and in an arbitrary order. The program reference

[1] The only important restriction on the order of slot definitions is that the type of a template must be contained in the first fragment about that template.

language is based on observations of common ways that people refer to parts of a program. The other features of fragments, such as incompleteness, inconsistency, and ambiguity, are also based on common assumptions that people make when specifying programs. There is no claim that this set of properties is complete, however.

In contrast to the fragment language, the typical computer program is specified to a programming language processor all at once, as a computer file or deck of cards. This program is a sequence of statements in a high level language, appearing in a very restricted order and using a precise and limited syntax. To be accepted by the processor, the program must be complete, consistent, and unambiguous.

Human users are not expected to interface directly to PMB. PMB was designed so that it may operate in a robust acquisition environment featuring many other knowledge sources, such as experts on natural language, inference from traces, and specific programming domains. In a more straightforward environment, fragments would be produced by a deterministic parser for an informal VHL surface language. For example, the two fragments presented above might look like this in a surface language modelled after PASCAL:

type *concept:* set of *relation*;

membership_test: *relation* ∈ *concept*

Throughout the thesis, fragments are expressed in this language.

5.2 Program Specification Information

This section discusses the information conveyed by a fragment about a particular template in the program model. The first subsection discusses the types of fragments. The next three subsections discuss the properties of fragments that distinguish them from program model templates: incompleteness, inconsistency, and variety of specification.

5.2.1 Types of Fragments

The kinds of fragments (i.e., as determined by the "type" slot of the fragment) form a superset of the kinds of templates in the program modelling language. Fragments without informalities correspond directly to parts of templates in the program modelling language. Hence, the class of programs that can be specified with fragments is a superset of those that can be specified in the program modelling language.

Below is a list of the types of fragments. Types followed by an asterisk correspond to model constructs by the same name. A complete definition of these types is deferred until Chapter 7 on the program modelling language. However, any aspect of informality possessed by a type is discussed here. Types not followed by an asterisk are unique to the fragment language and provide for some form of informality in specification. These types are discussed below.

Constructs of the Program Fragment Language

Abstract Information Units (AIUs)	Abstract Control Units (ACUs)
primitive»	composite»
string	test»
atom	case»
Boolean	condition/action
collection»	loop»
set	exit_pair
multiset	procedure»
ordered_set	program_model»
list	
correspondence»	
plex»	
subpart	
alternative»	
AIU_instance»	

Primitive Operations (POPs) with Boolean Values	Primitive Operations with Non-Boolean Values
not»	result_of
and»	correspondent_of»
implies»	
is_element»	
is_subset»	
true_for_all»	
has_correspondent»	
is_of_type»	
are_equal»	
part_of	

Primitive Operation with Side Effects	Value Labelling Primitive Operation
select_alternative»	remember»

I/O Primitive Operations	Control Flow Primitive Operations
input»	assert_exit_condition»
output»	procedure_instance»
inform_user»	return»

Some fragment types merely allow abbreviated specification of a program model construct with a particular set of predefined parameters (slot values). **String**, **atom**, and **Boolean** are abbreviations for an information structure of type **primitive** and subtype string, string, and Boolean, respectively. Similarly, **set**, **multiset**, **ordered_set**, and **list** are abbreviations for the four types of **collections**.

Some model templates have slots that take complicated values. Since a slot value cannot be modified once it has been defined, the contents of a slot cannot be incrementally specified. In order to allow remedy this situation, separate fragment types exist for complicated slots. These types are for the **subparts** of a **plex**, the **condition/actions** of a **case**, and the **exit_pairs** of a **loop**.

Result_of is an informal operator that takes as its argument the name of a construct that returns a value somewhere else in the model. *Result_of* returns the last value that its argument did, whether or not this value has been explicitly stored for such retrieval. Typically an information structure instance is created to hold the value, and a *remember* is placed around the construct to store its latest value in the instance, each time the construct is computed.

5.2.2 Incompleteness

All fragments, including the low level ones that map directly into model templates, require fewer details than eventually appear in the corresponding model. The simplest example of this is the extensive cross-references of the program model that are automatically generated by PMB.

Slots defining simple or optional program properties may be omitted from fragments; default values are provided by PMB. For example, the format for inputting a collection will default to a LISP list. Size information about a collection is optional, hence the default is none.

Slots can often be omitted if the appropriate value can be inferred from the context provided by the program model, whether immediately or after processing future fragments. A good example of inference is the type coercion of information structures such as procedure parameters and quantified variables.

5.2.3 Inconsistency

Certain inconsistencies that appear in fragments are automatically removed by PMB. One such inconsistency is called "type/token ambiguity" or, in the language of program models, prototype/instance ambiguity. The modelling language distinguishes between the prototype of an information structure and the one or more actual instances of it that are manipulated by the algorithm. If there is only one instance of a prototype, PMB allows fragments to skip defining the instance and make all data references directly to the prototype instead.

One type of informality of specification allows for many interpretations of a program fragment, depending upon the program model context. PMB specializes a generic operator that is not part of the modelling language into the appropriate primitive operation in the modelling language, based upon the operands. The semantics of these operators depend on the types of their arguments, so PMB does the appropriate operator coercion. For example, the fragment predicate *part_of* may be translated into one of the primitive operations *is_element*, *is_subset*, *has_correspondent* (does a domain element map to anything), or *is_component* (of a plex), depending upon its arguments and how they are represented in the program model. The fragment predicate *are_equal* may result in one of the following primitive operations: *is_empty* (is a collection empty), *are_equal*, *are_components_equal*, and *is_of_type* (is an alternative instance of a particular type).

5.2.4 Variety of Specification

Within the modelling language there may be a number of formally equivalent ways to encode the same expression or information structure. PMB recognizes the most common forms of such information and procedural structures and transforms them into concise, high level, canonical forms. The intent is to map equivalent expressions into one canonical form whenever they can be detected. For example, expressions that are quantified over elements of a set are canonized to the corresponding expression using set notation:

$$(\forall\, x)\; x \in A \supset x \in B$$

(a Boolean operation that determines if every element x in collection A is also in collection B) becomes

$$A \subseteq B$$

(is A a subset of B).

Below is another example, an English description of a set of fragments describing a set of marked elements.

> A *concept* is a set of *relations*. A *relation* is a plex (record structure) consisting of a *relation name* and *arguments*. Additionally, each *relation* is marked by a *label*.

The notion of "marking" an information structure is merely one way of creating a mapping from that structure into another. Since the modelling language has an explicit correspondence structure to handle this type of mapping, the canonization is done:

> A *concept* is a correspondence from *relations* to *labels*. A *relation* is a plex consisting of a *relation name* and *arguments*.

5.3 Program Reference Information

We have seen that program fragments can update parts of the program model in arbitrary order. The particular part, or template, currently must be referred to explicitly by its unique name. In addition, fragments referring to two different entities of a program model cannot be specified to occur in exactly the same location in the model: One must explicitly occur before the other. Thus, we have not addressed the problem of inferring control structures or sequencing constraints [Wile et al.-77].

There are many ways to refer to a point of interest in a program other than by explicit name. The more useful of these ways have been incorporated into a *program reference language* for specifying part or parts of a program. This language has *not* been implemented in PMB because a general reference capability wasn't necessary for PMB to function within its original context of the PSI program synthesis system. The techniques include reference to the program considered as a linear string of text and as a static syntactic structure, reference based upon the current context in the model (i.e., position of the last previous reference), reference based upon the historical order of prior references, reference based upon the semantics of the program

model, and reference based upon the pragmatics of the program model (i.e., involving domain knowledge).

Pointing directly at the desired program piece on a display screen with a pointing device (e.g., light pen or mouse) is obviously useful too, but can't be used to refer to the parts of a large system that can't fit on the screen or be searched visually by scrolling the screen. Pointing isn't adequate for specifying qualifying predicates either. Perhaps most important, a general purpose reference language should be usable by other programs, not just humans with a pointer. This form of reference is not considered further here.

Only a few examples of patterns in the reference language are given here; the entire language is outlined in greater detail in Appendix A. The first example makes use of syntactic reference. The following expression might be used to represent the statement, "the fifth *output* operation in the program that occurs somewhere after a conditional that is three levels down inside some block":

template n | (type=output, {*foo* $\downarrow^2$ *bar* .* n}, *foo*.type=composite, *bar*.type=test)$_5$

The expression within parentheses above matches *all* such *outputs*, and the subscript on the expression selects the fifth one (if it exists). The expression within braces is a template pattern in which the names of templates are separated by special pattern variables. ".*" constrains the templates or either side of it to be separated by zero or more templates in lexical order. "$\downarrow^2$" constrains its neighbors to have exactly two intervening lexical scope levels.

Using the symbol "*" to denote the current template, the following contextual expression specifies the closest test template above the current template:

template n | ({n $\downarrow^*$ *}, type=test)$_1$

As an example of semantic reference, the following pattern finds all (control structure) templates that contain below them a set operation that returns a Boolean value:

template n | {n $\downarrow^*$ x}, *type*(x)$\in$set_operations, *returns*(x)=Boolean

Chapter 6. Control Structure: The Rule Interpreter

The Program Model Builder is a rule based problem solver. It consists of two main parts, a set of antecedent (data driven) *rules* containing the system's model building knowledge and a *rule interpreter* control structure for choosing and applying the rules. The rule interpreter of PMB is the subject of this chapter; the details of the rules themselves are covered in Chapter 8.

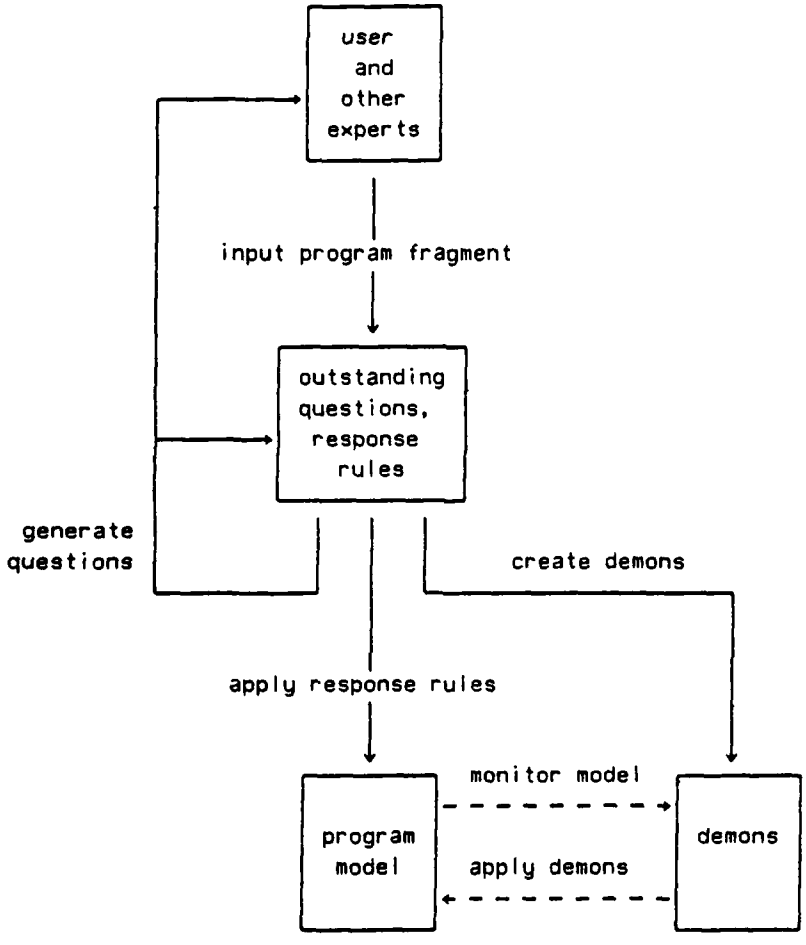
A fundamental assumption of this work is that the user of PMB has the *option* of controlling the interaction, in other words, that the program specification dialog allows mixed initiative [Carbonell-70A, Carbonell-70B, Bobrow et al.-77, Steinberg-79]. One result of this was discussed in the previous chapter: the design of the program fragment language, which is PMB's input language.

The other result of the decision to allow mixed initiative is a control structure that can deal with subgoals in any order and that can therefore deal with an almost arbitrarily incomplete program model. This means that the model may be incomplete at any level, from an entire procedure being missing to a single parameter of a single operation being left unspecified. PMB delays its operations only until the required arguments are completely specified. Incremental operation would be lost if PMB waited longer than this. This technique may be applicable to other problems that require a symbolic knowledge base to be acquired, maintained, and modified incrementally. Examples of possible applications are speech understanding, image understanding, other signal processing domains, and knowledge acquisition domains.

The problem of building a complete program model consists of a sequence of subproblems of the form: Given a partial program model and a new program fragment, update the model by making appropriate use of the new information in the fragment. As the program model is built, each new fragment answers some questions that were outstanding, but in turn may lead to new questions. Thus, the process of building the program model can be viewed as one of problem reduction or subgoaling, with the root of the goal tree being the completion of the program model, the set of unanswered questions representing the unexpanded subgoal nodes, program fragments providing the information necessary either to complete a node or expand it into further subgoals, and PMB's knowledge base about programming providing the basis for generating potential subgoal trees. To process subgoals that are completely internal to PMB (e.g., consistency checks), demon rules are created that delay execution until their prerequisite information in the program model has been filled in from fragments.

The overall dataflow within PMB is depicted below. The boxes represent the major databases within the system, except that the one labelled "user and other experts" is external to PMB. Lines denote the transfer of information between boxes, with arrows denoting the direction of flow. Labels denote the activities that cause the associated transfers. Activities under explicit control of the rule interpreter are denoted by solid lines; those that are implicit are denoted by dashed lines.

PMB Control Structure



Observe that PMB's subgoaling is data driven [Charniak-72]. PMB doesn't try any complicated goal driven reasoning on its own. This would result in hypothesizing about what the program model might look like, in the form of trees of incomplete subgoals. The space of possible programs is much too large for this approach to be useful. Instead, PMB maintains incomplete subgoals only at the leaves of the subgoal tree and then patiently waits for the user to get around to fulfilling these subgoals. The rules for processing the solution to a particular subgoal include all of the pertinent actions such as creating new subgoals, fulfilling other subgoals in the tree leaves by inference, setting up internal demons, etc.

The remainder of the chapter covers in turn data driven subgoaling, demons, and a comparison to structured programming, related problem solvers, and other possible approaches.

6.1 External Control Structure: Data Driven Subgoaling

The control structure of PMB may be viewed as a hybrid problem solver with two modes of operation. The default mode is a search of a limited AND/OR goal tree in which outstanding subgoals are worked on in an arbitrary order determined externally to PMB. The other mode allows new subgoals to be created by the user and then worked on. The program below is a simplified[1] specification of the top level control structure of PMB. It will be explained throughout the rest of this section.

Top Level Control Structure

```
create_template(program_model);
outstanding_questions ← nil;
ask_question(program_model.name);
ask_question(program_model.top_level);
while input(fragment) do
   ∀ template ∈ match(fragment,program_model) do
      if ∃ question ∈ outstanding_questions | answers(fragment,question)
         then ∀ question ∈ outstanding_questions | answers(fragment,question) do
            ∀ rule ∈ response_rules(question) do until invoke(rule) finally
               outstanding_questions ← outstanding_questions - {question}
         else create_subtree(template);
output(program_model);
return(if ∃ question ∈ outstanding_questions | type(question) = required then false else true)
```

We examine the default operation of PMB first. The top level goal of PMB is to form a complete program model, as represented by a unique template in the model called the *program_model* template. In this discussion the variable *program_model* refers to both the top level template and the entire tree of templates making up the model. In order to attain this goal, PMB creates this top level *program_model* template and generates two subgoals (via the function *ask_question*) for acquiring (1) the *name* of the program and (2) a *top_level* algorithm in terms of control structures and primitive operations operating on instances of the information structures.[2] For each subgoal, *ask_question* also generates a question that is sent out to PMB's external environment (e.g., the user or other expert programs). A *question* has the form of a template name and the name of a slot within the template that has yet to be filled in. PMB adds a new *question* onto the front of its global list of *outstanding_questions*. Associated with each *question* is stored a set of *response_rules* for dealing with the possible responses to that *question*. In addition, the *question* is tagged as "optional" if a default response has already been assumed. At the end of model building, the model is considered complete only if no "required" questions remain on the list of *outstanding_questions*, since any "optional" questions have already been answered by default.

---

[1] For example, checks for error conditions are omitted.

[2] As is discussed later in this section, at the level of the *program_model* template there are always implicit goals outstanding for new procedure and information structure definitions. There are also two optional slots that take atomic values defining the domain of the program (e.g., concept learning) and the user's name for this template. The latter slot is an optional slot in every template.

Recall that a fragment consists of two parts: (1) a pattern that specifies which template in the model is being referred to and (2) one or more slot/value pairs of that template. When PMB *inputs* a new program *fragment*, the program reference part is *matched* against the current *program_model* to determine a set of *templates* ordered by the order in which they were found by the matcher, according to the program reference language introduced in Chapter 5. Currently this *match* is only done on unique template names, so exactly one *template* matches. Then for each *template* suggested, the list of *outstanding_questions* is searched linearly to determine if the new *fragment answers* one of the *outstanding_questions*. It does so if the template name in *question* is the same as that in *fragment* and if the slot name in *question* is one of those present in *fragment*. If *fragment* and *question* correspond, then the *response_rules* associated with *question* are applied in the fixed order in which they were stored until one succeeds (i.e., returns true), and *question* is deleted from the list of *outstanding_questions*. The successful *rule* either completely answers the *question* by filling in the value of a template slot or causes a new level of subgoals (and questions) to be generated. This process continues until all *outstanding_questions* of *type* "required" have been answered, indicating the *program_model* is complete. Then the final model is *output*.

A *rule* is invoked (by the function *invoke*) within a simple context consisting of a set of global variables whose names are known by the rules. For response rules, these are the name of the current *template*, the name of the slot being filled in *template*, and the value from *fragment* to go in that slot. From this context, the *rule* currently being *invoked* may reference the current *fragment*, test and modify the entire *program_model*, ask new questions (and add them to the *outstanding_questions* list), and create demons (discussed in the next section). If the model is modified as a side effect of invoking *rule*, a demon that was created by an earlier rule may be indirectly triggered by the act of modification. The rule invocation function *invoke returns* a value of true if and only if the rule it invoked succeeded. This allows *invoke* to be used as a predicate, as in *invoke*(*rule*) in the algorithm above.

But what happens when a *fragment* doesn't answer any of the *outstanding_questions* (subgoals)? This occurs, for example, when a procedure or an information structure is defined before it has been referenced in the algorithm portion of the *program_model*. In this case PMB creates a new *template* and a new goal tree (denoted by the function *create_subtree* in the algorithm above) to acquire the new piece of *program_model* created by this program *fragment*. Also, by storing the name of the new *template* in the list of all information structures or procedures, a list kept in the *program_model* template, PMB notes that the implicit goal of completing these lists has been furthered. Later on, presumably, the new template and the independent subgoal tree of which it is the root will prove to be the solution to a subgoal in the main tree. Not all types of procedural templates can be defined before they are referenced. This mode has been limited to **procedure** definitions, since control structures and primitive operations can be put inside a **procedure** declaration if it is necessary to define them out of order.

Notice that if the first question in the list of outstanding questions is always answered next, then a depth first expansion of the goal tree ensues. Changing to breadth first would be trivial: just add new questions to the end of the list instead of the beginning. In fact, most of the mechanism is in place to handle a best first order if appropriate ordering heuristics were given. They haven't been written because topic coverage during the specification dialog is directed by other means, such as the user or another knowledge source (e.g., the PSI moderator [Steinberg-79]).

The rule interpreter is as incremental as possible. It processes one fragment completely before

accepting another. In the course of processing one fragment, a response rule will typically succeed for each slot/value pair in the fragment. Each of these rules may generate one or more questions, but handling the answer to any of them must wait until they are all asked (i.e., put on the list of outstanding questions). This has the advantage that related subgoals are generated at the same time, so that expectations are set up that allow PMB to deal with any of a number of related questions that the user might choose to address. To achieve the utmost in incremental operation, if this is desired, each fragment would be limited by its originator to only one slot/value pair.

Program modification is another case in which no question is answered by a fragment. However, in this case the template named already exists. The fragment's information is meant to supplant what was in the template before. A general program modification capability has not been implemented.

6.2 Internal Subgoals: Demons

You've got your demons; you've got desires. Well, I've got a few of my own.
—"One of These Nights"[3]
by Don Henley and Glenn Frey (The Eagles)

In addition to the rules that handle responses to questions sent externally, other rules handle subgoals that are remain internal to PMB, such as consistency tests between two or more templates, inference rules that allow a subgoal to be inferred indirectly when a different subgoal is answered explicitly, and transformation rules that delete or modify entire templates. If such a rule can't succeed because information is missing from the model, the rule waits until the necessary information is provided. Such *simple demon rules* are attached, along with a simple context of global parameters, to a particular slot in a particular template, waiting for the value there to change.[4] Whenever this happens, all the demons awaiting that slot value are reinvoked within their own stored context.[5] This is much more efficient than invoking all demons each time a slot value changes anywhere in the model. Each demon that succeeds is removed from the demon list.[6]

---

[3] Benchmark Music and Kicking Bear Music

[4] Response rules could also be handled by attaching them to the appropriate template and slot. However, there are theoretical arguments for keeping response rules, which represent externally asked questions about slots required for completing the program model, at a separate and higher level than the demons [Rieger-77]. Pragmatically, it is then easier to see if any questions are left unanswered, change priorities of questions, periodically ask again questions that haven't been answered yet, allow the user to ask questions about the questions, etc.

[5] To guarantee that all changes to slots are noticed, rules use a standard set of accessing functions to make all references to the program model. Those functions that modify the model call the demon invocation mechanism immediately afterward.

[6] Demons may also be explicitly deleted ("garbage collected") by other rules.

Demon rules are of two types, those that may and may not make structural modifications to the model. These types create two priority classes for demon execution, corresponding to those rules that make consistency checks or small inferences (e.g., fill in a slot) and those that make program transformations that alter a template's type (and hence its slots) or even its existence. A *consistency demon* (or a slot filler) can't be executed until the slot whose value will be referenced by the demon has been filled in by another rule. A *transformation demon* can't be executed until (1) the slot to be referenced has been filled in by another rule and (2) all outstanding consistency checks for the entire template have been done. The following demon invocation mechanism is used each time some *slot* in *template* changes value.

Demon Invocation Mechanism

∀ *demon* ∈ *template.demons* | *awaited_slot*(*demon*)=*slot* do
  if *type*(*demon*)=consistency
    then if *invoke*(*demon*) ∧ ¬∃ *other_demon* ∈ *template.demons* | *type*(*other_demon*)=consistency
      then ∀ *transform* ∈ *template.demons*
        | *type*(*transform*)=transformation ∧ *ready*(*transform*) **do**
        if ¬*invoke*(*transform*) **then** *mark_not_ready*(*transform*)
    else if ¬∃ *other_demon* ∈ *template.demons* | *type*(*other_demon*)=consistency
      then *invoke*(*demon*)
      else *mark_ready*(*demon*)

Since a transformation demon may not be fired if any consistency demons remain in the same template, in such a case a transformation demon whose slot has changed is marked ready instead of invoking it. Then, when the last consistency demon of a template has succeeded, the algorithm goes back and tries all of the ready transformation demons.

No consideration has been made in PMB for the incorporation of metalevel heuristics, especially those for efficiency [Kant-79A, Kant-79B]. They would prove useful mainly in the ordering of feasible transformations. Currently all transformations relevant to a given template are applied in an arbitrary order, and there is no backtracking. The first transformation to succeed will be carried out; the others will be forgotten. Efficiency heuristics should guide the transformation process by attempting to (1) minimize the running time and space of the program model at its very high level (e.g., when interpreted using default implementations of information structures) and (2) maximize the degree to which the model uses very high level (VHL) constructs so as to maximize the freedom to choose efficient implementations when the model is coded [Barstow-79A].

6.2.1 Compound Demons

Often a number of slot values must be filled before a demon can perform as intended. In this case, a *compound demon rule* is used. A compound demon may be thought of as a "cascade", an ordered set of simple demons such that only one is active at a time. The first of these simple demons waits (only if necessary) for the first required slot to be filled in. Then the antecedent that references this slot value is evaluated. If it is true, then the first simple demon completes its activities by setting up the second simple demon to wait for the next slot value to arrive, and so on. After the last slot value is defined, the last simple demon evaluates the final antecedent. If this antecedent is true, then the action part of the compound demon is executed and the entire compound demon goes out of existence. If any antecedent evaluates to false, then the

else_action of the compound demon is executed instead. Compound demons have the same two types as simple demons–consistency and transformation.

The technique of attaching demons to different templates and slots in succession is greatly automated by the rule expander. This rule "compiler" converts a compound demon that contains a declarative antecedent pattern into a series of separate simple demons, one per slot value used in the pattern. These demons utilize the same invocation mechanism as other simple demons, discussed above.[7]

There is often a natural partial ordering to the slot values required by the antecedents of a compound demon. The numerous slot values needed by a demon are often closely related in the space of model templates. In particular, it is typical for a pattern to first bind pattern variables to the values of one or more slots in one template, then the values of slots in templates pointed to by the first group of slots, etc., until there are hooks into an entire subtree of the model. Since the subtree will often be defined in some top down order, it is natural to order the slots this way too. From all the complete orderings of slot value bindings that are possible within the constraints of the partial ordering, the rule expander selects one arbitrarily for implementation of the pattern match.

A problem with the current implementation of compound demons arises from the interaction of the ordering of slot value monitoring and the existence of an else_action, an action taken whenever an antecedent fails. If the else_action contains error processing for a consistency check that has failed, then we wish this failure to be noted as soon as possible, i.e., as soon as one of the demon's antecedents is known to be false. Since the visiting of slot values is strictly ordered, a conjunct that is false may not be discovered until much later in the specification dialog, thus defeating the desire for incremental operation. An obvious, if not pleasing, way to restore fully incremental operation would be to create separate demons for the negation of each antecedent of a compound demon. The else_action would be removed from the original compound demon and made the action of each of the new demons. The compound demon, sans else_action, would remain to handle the case of success. Another solution would be to have all antecedents of a compound demon represented by demons, one per antecedent, all active in parallel. Whenever one of them failed, it would have to garbage collect the rest.

6.3 Comparison to Structured Programming

If questions are answered in the same order as they are posed, then a top down, depth first specification of the program model occurs. The first thing specified is the name of the program. Then the control structure at the top level of the program is defined. Then the first control structure or primitive operation just beneath the top level is completely defined, followed by the second, and so on. Whenever a previously unmentioned information structure or procedure is given as a slot value, questions pursue the details of this new entity until it is completely defined.

This ordering of subgoals provides a simple form of top down, structured program development. Whether or not this particular search order is a reasonable one for structured programming is not the point. Even structured programmers don't agree on the order in which

[7] The syntax and semantics of compound rules are discussed more fully in Chapter 8.

parts of a program should be filled out. The point is that PMB's technique accommodates nearly any order for defining the parts of the program.

The philosophy is one that assumes that only the user can ultimately know when to do what. For example, the user may suddenly remember an important detail required by a part of the program not currently under discussion and may want to add it to that part before the detail is forgotten again. Notice that this same flexibility is essential for program debugging and modification.[8] One must be able to access and change a particular program model part on demand. It might be claimed that the proper use of structured programming and verification will someday make debugging obsolete, but the need for program modification in the light of changing design specifications can't be dismissed so easily.

Because of the simplicity and generality of PMB's control structure, it should be straightforward to impose particular subgoal orderings. For depth first, the list of outstanding questions would be restricted to being a queue in which only the question at the front can be answered next. For best first tree expansion, a set of heuristics would reorder the queue of outstanding questions each time a new question was added.

Experimentation along these lines might aid our understanding of the human programming process. For example, we might choose a target programming task in a particular domain, have a number of expert (or some other level) programmers use PMB in its mixed initiative mode, and monitor how they go about building up the program [Brooks-75A, Brooks-75B, Brooks-77]. From these experiments, heuristics would be developed to be incorporated as metarules governing PMB's question asking. We might learn that programming is too idiosyncratic to make such heuristics useful in general. Or the appropriate set of heuristics for most programmers might vary from domain to domain or even from program to program within the same domain.

6.4 Related Problem Solvers

6.4.1 The Recognition Paradigm

The problem solving techniques used in PMB comprise one instance of the "recognition paradigm" [Minsky-75, Bobrow & Winograd-77]. The key aspect of recognition is the acquisition of knowledge by pattern matching and forward inference, as exemplified in UNDERSTAND [Hayes & Simon-74] and HEARSAY-II [Lesser et al.-75, Lesser & Erman-77]. The general methodology involves a system that accepts bits of information about some domain from a user and integrates this information into a global database (e.g., a "blackboard" or set of frames) that represents the system's understanding of the situation up until then. The system recognizes the relevance of the new input to its internal model of the world. The system uses a recognition process of some kind (e.g., pattern matching) to determine to what parts of the database the new input is relevant, then updates those parts appropriately.

---

[8] We distinguish between debugging and modification. *Debugging* is the process of changing a program that doesn't meet its design goals so that it will. *Modification* is the process of changing a program that already meets its original design goals so that it will meet new ones. In either case, the same system flexibility is needed.

PMB fits into the recognition paradigm very well. The global database is the program model tree being constructed. Inputs in the form of program fragments are matched against this tree, and then the tree is updated and inferences are made. The major new twist that PMB adds is the capability for having the user (or external system) specify, via the program reference language, which part of the model is to be updated. In earlier applications of the recognition paradigm, natural language and speech understanding, the possible order of topics is much more constrained by what has come before.

6.4.2 GUS

The reasoning portion of GUS, the "Genial Understander System" [Bobrow et al.-77], is a simple recognition system with a control structure similar to that of PMB. During a mixed initiative natural language dialog, GUS acquires the parameters of a simple round trip, such as a travel agent might set up over the telephone. The canonical trip scenario is represented by a set of prototype frames that get instantiated into the equivalent of PMB's templates. A slot in a frame instance may have a simple value or point to another frame. GUS starts out with a single frame representing a trip and proceeds to ask questions about all the slots in that frame until all required slots are filled in. This process typically involves the creation of more frames, filling in their slots, etc.

GUS expects every dialog to be about trip planning and, in fact, has only one trip scenario. PMB expects every dialog to specify a program, but has no preconception of the program beyond the language in which it must be written. The typical model built by PMB is larger than the trip scenario used by GUS.

Although both GUS and PMB are mixed initiative systems, GUS's default is for the system to control the dialog, and PMB's default is to let the user control it. GUS asks for slot values in a depth first order. GUS processes the next information received, regardless of whether this information is related in any way to the question just asked. Then GUS repeats a depth first search for the next outstanding question. GUS will (eventually) ask the previous question again if an indirect answer didn't lead to inferences that fill in the slot asked for. In contrast, PMB has no predisposition to any particular order for answering questions. Although it asks questions in a depth first order (at least locally), it doesn't ask a question a second time (unless required questions are left outstanding when fragments stop arriving) and doesn't try to gain control over the interaction.[9]

GUS will ask a question ahead of its default, depth first order if a slot value is required for some reasoning process currently taking place. Otherwise, it won't deviate from the default order even when a new frame instance is created. Because a trip plan is still useful when only partially complete, it is possible for a dialog to be completed without all questions being answered. PMB, because it is building an executable program model, has a fixed set of slots for each template that are required to be filled in by the end of the dialog. Since PMB knows which slots will be needed, it asks for them when a template is created.

GUS's knowledge resides primarily in its frame prototypes. This knowledge includes the slot names, simple goal seeking "servant" routines for filling in the slots, and simple demons for

[9] It would be simple to have a command that causes PMB to ask again the first question on its queue.

making inferences or consistency checks when a slot gets filled in. There are no compound demons. PMB's knowledge resides in its rule base: rules that create templates and explicit subgoals (questions), response rules that process answers, and rules that are attached to templates as demons. GUS's prototypes are easy to understand because all of the relevant information is in one place. But the sharing of knowledge among templates is probably easier when it is in the form of rules. For example, this allows PMB to create a template of unknown type and specialize it later into a template with known slots. More important is the fact that a GUS instance always refers back to its prototype for the names of relevant servants and demons. PMB attaches its demons to each template, which allows different instances of each template type to have a different set of demons attached at a given time. This allows demons to come and go as needed during model building. Of course, since GUS's demons are arbitrary programs, they could be given enough preconditions so as to have the same effect.

A final distinction between the two systems is that GUS never modifies or destroys frame instances once they have been created. PMB's templates can be deleted or altered by a transformation demon, once enough slots are filled in to allow the transformation to proceed.

6.4.3 Demon Regimes

The compound demons of PMB are one kind of "trigger pattern", consisting of associative, nonassociative, and computable components, as introduced in Rieger's theory of spontaneous computation [Rieger-77]. The associative component of our compound demons is restricted to *triggering when simple* predicates involving a single slot value become true. The nonassociative component is limited to evaluating the same simple predicates for slot values that are known to have values already. The computable component is one or more **require** statements, each with an arbitrary LISP expression to be evaluated.

Rieger introduces the notion of "pressure" versus "pulse" activation of demons. The simpler, pulse model reevaluates all antecedents (or at least enough to know that one isn't true) each time something changes that might bear on the value of one of them, until they are all true. The pressure model uses the notion of a "trigger tree" that allows each antecedent to be evaluated just once after it becomes true. The fact that it is true is stored so that when the final antecedent becomes true, it will notice that all the others are too and will proceed to execute the demon's body.

The matching mechanism of the ARS (Antecedent Reasoning System) language [Stallman & Sussman-77] uses pulse activation so that its internal state can be updated to reflect changes in old assumptions. ARS simultaneously monitors all changes in the database that might make true one of the antecedents of a demon. All demons are stored in a single decision tree that determines which demons are triggered by a new fact. This is but one example of how monolithic, general purpose knowledge bases (e.g., those whose basic unit is an S-expression), associated demon bases, etc. are less efficient than frame oriented structures. In a frame system, demons may be attached to particular slots of a frame, thus affording constant retrieval time.

When ARS notices a change that might affect a demon, all of its antecedents are evaluated to see if the demon body can be executed. If not, no partial information is retained about which antecedents are already true. But in some cases, an ARS demon can proceed even though some information is missing because antecedents are divided into mandatory and optional ones. This scheme has an advantage over that used in PMB in that ARS automatically checks that

enough of the antecedents are true just prior to executing the demon's body. But this is done at the expense of needlessly reevaluating many true conjunots each time some of the others are false. Assuming all of the conjuncts are required and none becomes false after once becoming true, the best case occurs when the first of the n conjuncts to be evaluated becomes true last. Then only that conjunct will be reevaluated—n times—and the total number of evaluations is approximately $2n$. The worst case happens when the conjuncts become true in the same order as they are evaluated. Then the evaluation time is order n^2.

PMB uses a form of pressure activation in which antecedents are ordered so that only one is actively monitoring the database at a time, rather than all of the nontrue conjuncts as with trigger trees.[10] Thus, exactly n evaluations are done. Since only one conjunct is active at a time, some storage space is saved and, depending on how the database is implemented, monitoring time may be saved also.

In general, however, this scheme could be dangerous in an environment in which database changes are allowed, such as in program modification. A change could result in a demon conjunct that was true earlier now becoming false, after the demon has already found it true. The cure for this potential problem is to reevaluate all conjuncts once, just before executing the demon body. In the case in which the demon succeeds (i.e., all the conjuncts are still true), this algorithm does exactly $2n$ - 1 evaluations. The case in which one or more conjuncts are now false can most easily be handled by reinitializing the demon to wait to evaluate all conjuncts again. Let p be the probability that, when the last conjunct becomes true, all of the rest are still true. Then the expected value for the number of evaluations is about $2n/p$. Thus, as long as a lower bound on p exists, we are assured that the algorithm is linear for all cases.

6.4.4 Processes

Another way to view demons is as processes that block until particular conditions are met. This view has resulted in a compiler for ALGOL68 that uses demon-like processes instead of multiple compiling passes to handle information that arrives in nonlinear order [Banatre et al.-79]. This compiler handles the problems of type coercion and storage allocation in a language in which data declarations aren't required to occur at the start of blocks.

6.5 Other Approaches

6.5.1 MYCIN

The MYCIN system for diagnosing bacterial infections [Shortliffe-76] is a well-known rule based system. However, it's rules are mostly in consequent (or backward chaining) format [Davis et al.-77], rather than the antecedent (or forward chaining) format primarily used by PMB. This bias reflects a design decision that accepted the constraint that the system is in complete control of the problem solving dialog. Thus, MYCIN's top down subgoaling is equivalent to a PMB specification dialog in which there is no user initiative and for which

[10] PMB's passing of bindings from already matched conjuncts to the currently active one is a simple form of Rieger's notion of the "splitting" of spontaneous computations.

straightforward subgoaling therefore works. PMB's inferencing approach was developed because we felt that, given the difficult task domain of writing computer programs, any rigid dialog format would stifle programmer creativity and lead to too much user frustration to prove successful.

6.5.2 SAFE

The Specification Acquisition from Experts (SAFE) system [Balzer et al.-78] takes a novel approach to some of the program synthesis issues dealt with by PMB. SAFE is a three phase, noninteractive program synthesis system that converts a program specification into a VHL language version of it. The three phases cover linguistic, planning (ordering), and meta-evaluation (completion and consistency) knowledge. SAFE has acquired programs for message processing and for scheduling timeshared resources.

SAFE's first phase inputs parsed English sentences specifying relatively complete, independent parts of a program. This phase infers domain knowledge in the form of the objects that exist and the relations that hold between them [Goldman et al.-77]. In contrast, PSI was designed to have precodified domain knowledge available from a separate domain expert, and no attempt is made to learn anything new about the domain while the system is running. PMB itself has no knowledge of the particular application domain of the program model.

The second phase of SAFE analyzes where program variables are produced and consumed, in order to create a partial ordering of the pieces of the program into an executable form [Wile et al.-77]. This is required because ambiguous, demon-like, parallel specifications may be input to SAFE, mainly because of the application domain. Although no work in this direction has been done for PMB, it would certainly be useful.

The final phase partially symbolically executes ("meta-evaluates") the program to determine its completeness and consistency [Balzer et al.-77]. This phase can discover missing procedure parameters and often infer what they are from type information and context. The final program is written in an AI language called AP/1, in which relation is the only data type. The fact that this language is quite a bit higher level than the program modelling language produced by PMB (e.g., it has a demon-like control structure) may make AP/1 program acquisition somewhat easier. SAFE doesn't have a coding phase that transforms the AP/1 program into an efficient implementation; AP/1 wasn't especially designed with this in mind.

The meta-evaluation phase of SAFE corresponds most closely to PMB in terms of the types of processing it does on programs. The fundamental design difference is that SAFE is not an interactive, incremental system. The meta-evaluator isn't called until the program is nearly complete. Hence, it makes sense to attempt to evaluate the program symbolically (or even interpretively using test data [Wilczynski-75]) to see if any problems remain. This approach will not work in a system such as PMB, which is designed to work incrementally, gleaning as much information from each new program fragment as possible and providing immediate feedback. To summarize the fundamental difference, SAFE defers inferences about completeness and consistency until the program can be symbolically evaluated; PMB makes inferences as soon as enough information is present to do so.

Chapter 7. The Output: Program Modelling Language

The knowledge base of PMB may be divided into two parts: dynamic and static. The dynamic knowledge is that which varies from one run of PMB to the next. This is represented during a run by the partial program model that is being built up and at the end of model building by the completed program model. The nature of the program modelling language in which these models are written is the subject of this chapter. The static knowledge base is the body of rules that are used to build a program model from fragments. These rules are the topic of the next chapter.

The *program modelling language* includes (1) information structures, (2) control structures, (3) *primitive operations* (which are embedded within control structures and which operate on information structures), and (4) optional assertions attached to any of the other types of model elements.[1] Each of these four topics is treated in a separate section.

The program model produced by PMB is so termed because of a desire to closely model the corresponding program in the user's head and because it is an abstract, implementation independent program specification that may actually lead to many different concrete implementations. A program model is essentially a highly annotated program written in a very high level (VHL) language. A model can actually be executed, albeit slowly.[2]

The program model is written in a VHL language in order to (1) keep the parts of the model at a level not too far below the user's own conceptualization of the problem and free of unnecessary detail; (2) allow detailed algorithm and data structure selection to proceed in a separate, nonincremental phase; and (3) keep the synthesis problem tractable for PMB. The use of a VHL language, either by machine or human, is part of the natural evolution of expressibility in computer programming from programming in raw binary machine language through assembly language, macro languages, FORTRAN, and the more advanced high level languages of today (e.g., ALGOL, LISP, PASCAL). In fact, until only ten years ago the term "automatic programming" referred to the development of the assemblers, macro expanders, and compilers for these earlier languages.

The modelling language is designed to easily express common programs in the general area of symbolic computation (e.g., set and list processing, symbolic concept formation, information retrieval). In addition, its constructs are amenable to machine codification and automatic synthesis. The VHL nature of the language comes mainly from its information structures and the primitive operations that are allowed on them, rather than from complex control structures (e.g., backtracking, pattern directed function invocation) and the database mechanisms necessary to support them (e.g., context trees) that are found in AI languages [Bobrow & Raphael-74].

---

[1] The original specifications of the information structures, control structures, and primitive operations are due to David R. Barstow. The language has evolved considerably from that point.

[2] A program model interpreter exists for helping the user verify the correctness of the model and for gathering runtime statistics on collection sizes and branching probabilities for efficiency analysis [Nelson-76]. This interpreter can be used to execute all or just selected parts of a model. Bruce Nelson wrote the interpreter, and Richard E. Pattis added a general information structure parser for *input* operations.

The information structures were also influenced by those of predecessor set oriented languages such as SETL [Kennedy & Schwartz-75, Schwartz-75, Schonberg et al.-79] and VERS2 [Earley-73A, Earley-73B, Earley-74]. Sets and mappings are examples of information structures that meet our requirements.

Below is a list of all of the types of information structures (*abstract information units*, or AIUs), control structures (*abstract control units*, or ACUs), and *primitive operations* (POPs), classified by how they obtain their effects. The primitive and collection AIUs have a number of subtypes, which are also listed. A primitive operation may either return a value (which may be used in an expression where the operation was called) or operate by side effect (in which case it modifies one or more of its arguments). A small number of special primitive operations gain effect by such side effects as enumerating elements in a collection, doing input/output, changing the flow of control, or remembering the value of another operation in an instance for later use. All five AIUs, all six ACUs, and sixteen of 51 POPs have been implemented. These are designated by asterisks following their names.

Constructs of the Program Modelling Language

Abstract Information Units (AIUs)
primitive* (string, Boolean, integer)
collection* (set, multiset, ordered set, list)
correspondence* ("mapping", "function")
plex* ("record")
alternative* ("union")

Abstract Control Units (ACUs)
composite* ("compound")
test* ("conditional")
case*
loop*
procedure*
program_model*

Primitive Operations (POPs) with Boolean Values
not*
or
and
is_empty
is_element*
precedes
is_subset*
true_for_some
true_for_all*
has_correspondent*
correspond
is_component
are_components_equal
is_of_type*
are_equal*

Primitive Operations with Non-Boolean Values
remembered_value*
new_primitive
new_collection
convert
element_of
subset
union
intersection
difference
new_correspondence
correspondent_of*
domain_of
inverse_of
new_plex
component

Primitive Operations with Side Effects
add_element
add_elements
remove_element
remove_elements
replace_element
transfer_element
transfer_elements
establish_correspondence
remove_correspondence
change_correspondence
replace_component
select_alternative*

Primitive Operations That Enumerate
for_some_do
for_all_do

I/O Primitive Operations
input*
output*
inform_user*

Control Flow Primitive Operations
assert_exit_condition*
procedure_instance*
return

Value Labelling Primitive Operation
remember*

7.1 Information Structures

The information structures of the modelling language were designed to handle a variety of simple symbolic computations. Information structure prototypes may be defined recursively to build quite complex structures. Each structure may have associated with it a list defining one or more legal values that the structure may take on. If there is exactly one such value, then the structure is a constant. Each prototype information structure may have zero or more concrete instances that are assigned values and manipulated by the primitive operations of the language.

A **primitive** requires no subordinate information structures to complete its definition. A **primitive** may be of subtype **string**, **Boolean**, or **integer**, which determines the kind of atomic value the **primitive** represents.

A **collection** is a group of semantically similar elements, where an element is an arbitrary information structure (e.g., a **primitive**, another **collection**). A **collection** is categorized by whether its elements are ordered and whether they may be repeated. The resulting four types of **collections** are **set**, **multiset** (or "bag" [Rulifson et al.-72]), **ordered set**, and **list** (or "ordered multiset"), as determined by the table below.

Types of Collections

	No Repetitions	Repetitions
Unordered	set	multiset
Ordered	ordered set	list

It is advantageous to group these four related information structures into one basic entity, the **collection**, with two binary subtype specifiers. Since the four subtypes have much in common, the knowledge involving their definition and use (i.e., by primitive operations) can be factored so as to be nonredundant.

An *explicit* information structure has its value explicitly stored somehow; an *implicit* one must compute its value when needed. A **correspondence** is an explicit function consisting of mappings from the elements of an implicit domain **set** to those of an implicit range set. A **correspondence** may be thought of as a set of ordered pairs of domain and range elements. The implicit domain is then the set of all first elements of ordered pairs, the implicit range the set of all second elements. A particular domain value can occur in at most one ordered pair at a time. This information structure has widespread applicability. For example, such common structures as arrays, property lists, symbol tables, and simple databases may be expressed as **correspondences**.[3]

A **plex** is a group of semantically dissimilar elements, like the record structure found in many high level languages such as PL/I, PASCAL, and INTERLISP. Each field of a **plex** has a unique (within that **plex**) constant name and an associated value. The type of each value is defined by another information structure in the model. A field of a **plex** can be referred to only by its name, and not by less meaningful indices, such as its position in a list.

An **alternative** is a group of information structure choices, similar to the union declaration of

[3] It is desirable to extend the notion of **correspondence** by creating a new information structure for expressing general, explicit relations, rather than limiting them to only functional ones.

ALGOL68 [Van Wijngaarden et al.-69]. The value of an instance of an **alternative** must be of a type corresponding to the selection of exactly one of these information structures. This allows a program to handle (e.g., read in) an information structure whose precise structure isn't known at the time the program is specified.

7.1.1 Information Structures of the TF Program Model

As an illustration of most of the important aspects of the program modelling language, we will present a fairly complex program model of a concept formation program called TF. TF (for "Theory Formation") is a simplified version of Winston's concept formation program [Winston-75]. The goal of TF is to form an internal model (in the form of a simple information structure—not to be confused with a program model) of a concept that may be used to discriminate between "scenes" that are and are not part of the concept. TF builds up its internal model by repeatedly reading in a scene that may or may not be an instance of the concept. TF determines whether each scene fits the current internal model of the concept and verifies this guess with the user. The internal model is then updated based on whether or not the guess was correct. The internal model consists of a set of relations, each marked as (or mapped into) one of the labels, "necessary" and "possible". A scene fits the model if all of the "necessary" relations are in the instance; "possible" relations are optional.

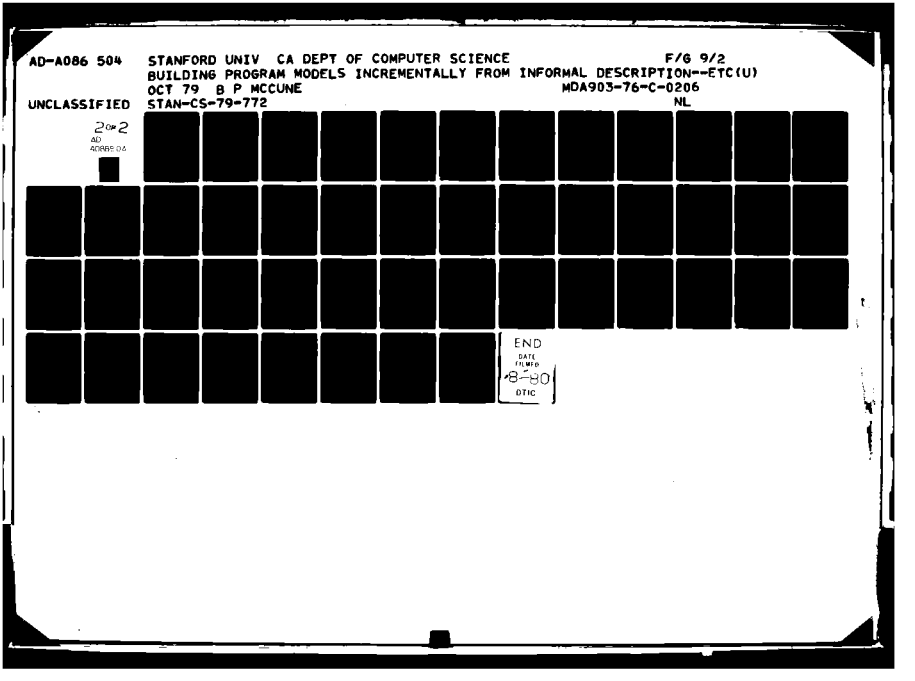
As an example of an internal model, one plausible model of the concept of a blocks world "arch" is the **correspondence** (set of mappings)

$$\{cube(a) \rightarrow necessary,\ cube(b) \rightarrow necessary,\ cube(c) \rightarrow possible,\ pyramid(c) \rightarrow possible,$$
$$supports(a,c) \rightarrow necessary,\ supports(b,c) \rightarrow necessary,\ not\_touching(a,b) \rightarrow necessary\}$$

This **correspondence** indicates that an arch must have three blocks. Two of them are cubes that are not touching and that support the third block, which may be either a cube or a pyramid.

The information structures of TF are given below. They are presented in a PASCAL-like notation that is produced from the actual program model by the readable program model generator[4]. The program model itself is rather unreadable, being maintained internally in a parsed form as a tree of templates, each of which is an association list. The procedural part of TF is shown in the next section.

[4] This "prettyprinter", written by Thomas T. Pressburger, provides concise, understandable versions of program models [Pressburger-78]. Any or all of the parts of a partial model may be printed, and cross-reference tables are available to index the line numbers of the concise listing and the template names of the original model.


AD-A086 504
STANFORD UNIV CA DEPT OF COMPUTER SCIENCE
F/G 9/2
BUILDING PROGRAM MODELS INCREMENTALLY FROM INFORMAL DESCRIPTION--ETC(U)
OCT 79 B P MCCUNE
MDA903-76-C-0206
UNCLASSIFIED
STAN-CS-79-772
NL
2 OF 2
AD
A086504
END
DATE FILMED
8-80
DTIC

Information Structures of the TF Program Model

```
type
    input_data_prototype: alternative of {scene_prototype, string = "quit"},
    scene_prototype: set [3 ≤ size] of relation_prototype,
    concept_prototype: correspondence [1 ≤ degree] of
        relation_prototype to string = {"necessary", "possible"},
    relation_prototype: plex [size = 2] of
        <relation_name: string, arguments: list [0 ≤ size] of string>,
    user_response_prototype: alternative of {string = "correct", string = "wrong"};

var
    input_data: input_data_prototype,
    scene: scene_prototype,
    concept: concept_prototype,
    relation: relation_prototype,
    user_response: user_response_prototype,
    necessary, possible: string = {"necessary", "possible"},
    fit_result: Boolean
```

An information structure definition in a program model creates a new prototype of the abstract information unit (AIU), of which there may be any number of actual instances with various values during the execution of the model. In the example above, the AIU prototypes are defined after the reserved word type, and the AIU instances after the word var. For conciseness, the definitions of simple prototypes are expanded where the prototypes are referenced instead of giving them names to be referenced elsewhere.

We see that *input_data_prototype* is either a *scene_prototype* or the string constant "quit". *Scene_prototype* is a collection of at least three unordered, unrepeated elements called *relation_prototypes*. *Concept_prototype* is a many-to-one mapping from an implicit set of *relation_prototypes* to the implicit set containing the strings "necessary" and "possible". *Relation_prototype* is a plex with two fields, a string relation name and a list of string arguments. *User_response_prototype* is either of the string constants "correct" or "wrong".

Most of the AIUs have only one instance. The AIU that is a string with two possible values, "necessary" and "possible", has two instances, unimaginatively called *necessary* and *possible*. *Fit_result* is an instance of a Boolean primitive.

In order to exemplify the detail contained in an information structure, listed below is the detailed AIU prototype for the correspondence *concept_prototype*, along with its sole instance, *concept*. They are presented as two templates, each consisting of a list of slot/value pairs. Note that the domain and range elements of the correspondence are names of templates defining these AIUs (not shown here). The where entries of *concept* provide a cross-reference to each template in the model where *concept* is used (i.e., created, destroyed, read accessed, or changed). These templates are not shown. Also notice that assertions about the sizes of the implicit domain and range sets are maintained.

Details of Two Information Structure Templates

name: *concept_prototype*
class: AIU
type: correspondence
super-AIUs: none
instances:
    concept
domain_AIU: *relation_prototype*
domain_size:
    minimum: 0
    maximum: unknown
    mean: unknown
    variance: unknown
range_AIU: *label_prototype*
range_size:
    minimum: 0
    maximum: 2
    mean: unknown
    variance: unknown
many_to_one: true

name: *concept*
class: AIU_instance
instance_of: *concept_prototype*
where_remembered:
    $initialize\_concept_{1}$
where_forgotten: none
where_referenced:
    $fit_{1,1,1}$
    $update_{2,1,1,1,1}$
    $update_{2,2,1}$
    $update_{4,1,1}$
    $update_{4,3,1}$
    $update_{7,1,1}$
    $update_{7,3,1}$
where_modified:
    $update_{2,2}$
    $update_{4,3}$
    $update_{7,3}$

7.2 Control Structures

The control structures of the program modelling language are generalizations of types common to block-structured languages such as ALGOL.

A **composite**, or compound statement, is a set of operations to be performed, with a partial ordering on their execution. The two extremes, no orderings and fully ordered, provide the useful special cases of fully parallel and fully sequential execution. The default is fully sequential.

A **test** is just like an **if-then-else** biconditional in ALGOL. A **case** is a multiway conditional whose conditions are independent and unordered, with exactly one of them true. One of the cases may use a default condition that is true whenever none of the others is [Barth-74].

A **loop** is a generalized loop structure that (1) has an explicit initialization part in addition to the body of the loop and (2) allows exiting on any number of exit conditions, with special exit actions associated with each condition [Zahn-74, Knuth-74]. Standard **for**, **while**, and **until** loops all fall within this framework.

A **procedure** may return a value or not and may have parameters or not. All parameters are considered to be called by reference.

7.2.1 Procedural Part of the TF Program Model

To help clarify the procedural notions in a program model, below is a PASCAL-like version[5] of the abstract control units (ACUs) and primitive operations (POPs) of the TF program introduced in the previous section. This example represents about half of the program model. All information structure definitions are given in the preceding section; cross-references and most other annotations have been omitted.

Procedural Part of the TF Program Model

```
until exit
    first
        parbegin
            necessary ← "necessary";
            possible ← "possible";
            concept ← concept_prototype{}
        parend
    repeat
        begin
            input_data ← input(input_data_prototype, user, "Ready");
            if input_data = "quit" then assert_exit_condition(exit);
            scene ← input_data;
            fit_result ← (concept⁻¹[necessary] ⊆ scene)[p = 0.5];
            if fit_result then inform_user("Fit") else inform_user("Didn't fit");
            user_response ← input(user_response_prototype, user, "Is this correct or wrong?");
            case
                fit_result ∧ user_response = "correct":
                    ∀ relation ∈ scene | relation ∉ domain(concept)
                    do concept[relation] → possible;
                fit_result ∧ user_response = "wrong":
                    ∃ relation ∈ concept⁻¹[possible] | relation ∉ scene
                    do concept[relation] → necessary;
                ¬fit_result ∧ user_response = "correct": ;
                ¬fit_result ∧ user_response = "wrong":
                    ∀ relation ∈ concept⁻¹[necessary] | relation ∉ scene
                    do concept[relation] → possible
            endcase
        end
    finally
        exit:
endloop
```

The main body of the model is a loop, shown delimited by the reserved words **until** and **endloop**. The three parts of the loop, the initialization, body, and exit blocks, follow the reserved words **first**, **repeat**, and **finally**, respectively. There is one exit block, called "exit", which is empty. The initialization consists of a fully parallel composite that initializes the two instances of string primitives, *necessary* and *possible*, to the values "necessary" and "possible" and initializes the *concept* instance of the correspondence called *concept_prototype* to have no mappings.

---

[5] produced by the readable program model generator

The body of the loop is a fully sequential composite that first inputs from the user an instance of the *input_data_prototype* alternative and stores it in *input_data*. *Input_data* may either be an instance of the set *scene_prototype* or the string "quit". If it is "quit", then the body of the loop is left via an *assert_exit_condition* operation and the exit block called "exit" is executed. If it isn't "quit", then *input_data* must be an instance of *scene_prototype*. In this case, the *select_alternative* operation is used to rename *input_data* as *scene*, a particular instance of *scene_prototype*. From this point on, *input_data* is no longer defined.

The fourth statement of the composite tests whether *scene* fits the current *concept* and stores the result of the test in the Boolean primitive *fit_result*. The test checks whether the set of all elements in the domain of the correspondence *concept* that map into the string *necessary* is a subset of *scene*. This computation uses the *inverse_of* operation on a correspondence and a particular range element. The *is_subset* operation is annotated to show that the probability that it is true is 0.5.

The next two statements print out the result of the test and then input the user's agreement or disagreement with that result, in the form of one of the strings "correct" or "wrong". This string is stored in the instance *user_response* of the prototype *user_response_prototype*, which is an alternative of the two possible strings.

The final statement in the loop body is a case with four possibilities for updating *concept* based upon the cross-product of the two possible values of *fit_result* and the two for *user_response*. The first case puts every *relation* in *scene* that isn't already in *concept* into *concept*, with a mapping into *possible*. The second case chooses a *relation* that is marked *possible* in *concept* and is not in *scene*, if one exists, and changes its marking to *necessary*. The third case doesn't have any action. The fourth case is similar to the second, but changes each *relation* marked *necessary* in *concept* and not a member of *scene* so that it is marked *possible* instead.

Below are the details of one control structure template, the main loop of TF. Note that this template points to the control structures both above and below it in the program model tree. The root template, *TF*, is directly above it, and *initialize* and *input_and_process_body* are the templates directly below it, which define the subparts of the loop. The relevant AIU instances listed are those that are only used locally to this subtree of the total control structure of the model.

Details of a Control Structure Template

name: *input_and_process*
class: ACU
type: loop
super-ACU: *TF*
relevant_AIU_instances:
    necessary
    possible
    concept
initialization: *initialize*
body: *input_and_process_body*
exit_pairs:
    <exit, nil>

7.3 Primitive Operations

There are approximately fifty primitive operations[6], including some of a fairly high level. A number of these were presented in the program model example of the preceding section.

There are operations for creating, accessing, changing, and combining the components of information structures in various ways; making standard Boolean tests; doing input and output; and calling and returning from procedures. For example, the standard set operations *union*, *intersection*, and *difference* are defined on collections.

Here are some examples of higher level operations. *Inverse_of* takes the inverse of a correspondence under a particular range element, i.e., *inverse_of* returns a subset of the domain of the correspondence consisting of each element that maps into the chosen range element. *Subset* returns all elements of a collection that satisfy a given predicate. *True_for_all* and *true_for_some* allow a collection to be examined to see if a condition holds over all or at least one of its elements. Examples of the universal and existential enumeration operations, *for_all_do* and *for_some_do*, were presented in the preceding section within the case statement of TF.

7.4 Assertions

Certain metalevel information in the form of assertions may be attached to program model templates. *Assertions* are not required for a complete model, e.g., the model can be interpreted without the existence of any assertions. However, assertions can provide valuable information to PMB, other acquisition experts, or a later coding phase. Currently assertions may only state user assumptions or estimates (e.g., of collection sizes or probabilities of conditions being true in control structures or primitive operations). It would be trivial to add a definitional type of assertion that acts as a command to PMB or later coding stages, e.g., "This information structure should be implemented as a linked list.". Assertions are represented by special slots in templates, one slot name per assertion type. These slots are handled like other slots.

[6] A complete list was given at the beginning of this chapter.

Chapter 8. The Knowledge Base: Rules for Building Program Models

There are two aspects of rules: form and function. First we define the format and types of rules in PMB's knowledge base. Then types and examples of specific knowledge about programming are presented. These include facts about incremental construction of program models, completeness, semantic consistency, and canonization. English paraphrases of many more PMB rules are given in Appendix B.

8.1 Format and Types of Rules

PMB's expertise is implemented as a set of procedural rules that are scheduled by the rule interpreter discussed in Chapter 6. All rules (i.e., both response and demon rules) use a standard *antecedent/consequent* format:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow c$$

Each antecedent (or precondition) a_i is either a simple Boolean test on the state of the current fragment or program model or an explicit call to a Boolean function. The consequent c is a sequence of actions that access the current fragment and partial program model, modify the model, ask questions, call other rules (perhaps creating demons), and return a Boolean value. Each antecedent is evaluated in order. If one is false in a response rule, then the rule fails. If one is false in a demon, then the demon blocks at that point, awaiting a later change that might make the antecedent true. Only when all of the antecedents are true is the consequent executed. A rule achieves success only if all of its antecedents and its consequent are true.

Being procedural, the rules run compiled and hence very efficiently. However, it is difficult to add, modify, or generalize rules in this form. So a high level "rule expander" has been written that generates procedural demon rules from a declarative antecedent pattern. On the average, one of these compound demon rules translates into about five simple rules.

The rest of this section discusses the two types of rules, simple and compound. The current rule base of PMB consists of approximately 200 simple rules and twenty compound demons. Perhaps twenty of the simple rules are demons that were written by hand before the existence of the rule expander. If rewritten, these would result in four or five new compound demons.

8.1.1 Response Rules

PMB knows many facts about program models in order to be capable of building them and guaranteeing that they are complete and consistent. PMB knows all of the types of control structures, information structures, and primitive operations in the modelling language; the properties of each construct; the legal values of each property; which properties are required and which are optional; and default values for properties. Other knowledge takes the form of checks or transformations on one or more properties.

Most of this knowledge is organized into response rules according to the program model construct involved. The following table contains paraphrases of what PMB knows about an example control structure and an example information structure.

Examples of Knowledge of Legal Program Models

A **loop** consists of an optional initialization, required body, and required pairs of exit tests and exit blocks. Each exit test must be a Boolean expression occurring within the body.

A **collection** may be ordered or not, may allow repetitions or not, may have a size estimate, must define the prototypic element, and may have instances. The default I/O format is a LISP list of elements.

The nature of response rules will be covered in more detail by discussing one representative response rule. This rule handles the acquisition of an *is_subset* primitive operation. To provide context for the discussion, the completed *is_subset* template from the CLASSIFY program model is shown below. This template represents an operation that tests whether the **collection** *fit_concept* is a subset of the **collection** *fit_scene*.

Completed Is_Subset Template

name: *procedure_body*
class: primitive operation
type: *is_subset*
super-ACU: *fit*
type_returned: Boolean
subcollection: *fit_concept*
collection: *fit_scene*

Processing of the **name** and **super-ACU** slots is trivial and isn't shown. Although processing of the **type_returned** slot for a Boolean condition is also trivial, the general case is discussed in the next section.

The rule is designed to be invoked whenever an *is_subset* is a legal type for a new template, i.e., whenever a Boolean value is permitted. For this to happen, the name of this rule is included in the list of response rules associated with any question that asks for a Boolean expression. For example, when the subgoal for the condition part of a test is created, a question asks for a Boolean expression. The name of the function that is the rule below will be a member of the list of response rules stored along with this question. In terms of the slot values below, the *is_subset* operation takes the form SUBCOLLECTION $\subseteq$ COLLECTION. TEMPLATE is the name of the new template being defined by the latest fragment.

Response Rule for Handling Is_Subset Operation

If the TYPE of the new template is *IS-SUBSET*, *then*
(1) store POP (primitive operation) as the CLASS of TEMPLATE;
(2) store IS-SUBSET as the TYPE of TEMPLATE;
(3) add TEMPLATE to the list of all IS-SUBSETs in the program model;
(4) ask the external environment what the SUBCOLLECTION slot of TEMPLATE is, storing the question away along with appropriate response rules to handle the answer, which is required;
(5) ask what the required COLLECTION is similarly;
(6) guarantee that the two types of collections returned by SUBCOLLECTION and COLLECTION have the same prototypic element;
(7) assume that there will be no PROBABILITY given;
(8) ask what the optional PROBABILITY is that the IS-SUBSET returns a value of true; and
(9) return success.

Note that Steps 1, 2, and 7 store values into slots in the template and thus may trigger demons that are waiting for those slots to be filled in. Step 3 generates a global cross-reference to this template by its type. Steps 4, 5, and 8 generate subgoals. Step 6 sets up a compound demon that is discussed in the following section. Step 8 asks an optional question (i.e., one not required to be answered) because Step 7 has already assumed a default answer.

Steps 4 and 5 create questions that are equivalent except for the slot name involved (SUBCOLLECTION versus COLLECTION). Each of these questions has five response rules (discussed individually below) to handle the various possible states of the program model when a template name is provided in answer to the question. These states arise because templates and parts of templates may or may not exist at any particular time in the midst of model building, since program fragments may arrive in any sequence. Note that exactly one of the response rules succeeds.

Response Rules for Handling Arguments of Is_Subset

(1) If the template doesn't exist, create one and give it a class of "collection". This is a pseudo-class that serves as a constraint on the class and type allowed for this template.[1] Other rules notice this pseudo-class and restrict the template to be either an instance of a collection information structure or a primitive operation that returns an instance of a collection as its value.[2]

(2) If the template has pseudo-class "collection", then don't do anything. It is already guaranteed to return an instance of a collection.

(3) If the template is an information structure instance, then set up a demon to guarantee that it is eventually defined to be an instance of a collection.

(4) If the template is a primitive operation, then set up a demon to guarantee that it is eventually defined as returning an instance of a collection.

(5) If the template exists but isn't of one of the types discussed above, then there is an error because the template can't possibly return an instance of a collection.

8.1.2 Simple Demons

Simple demons are used whenever an operation may have to wait for a single slot to be filled in.[3] There are few simple demons.

As an example, all templates that return a value have a TYPE-RETURNED slot. The value of this slot is determined from the values of other slots in the template in various ways. An *input* operation has an AIU slot defining the information structure that is input, which is the type returned by the *input*. When the *input* template is created, a simple demon is set up that waits for the AIU slot of the template to be filled in and then copies its value into the TYPE-RETURNED slot of the same template.

In many cases, the TYPE-RETURNED slot of a template will simply be the same as the TYPE-RETURNED slot of a second template pointed to by the first. For example, the TYPE-RETURNED by a *procedure_instance* (or call) will be whatever the TYPE-RETURNED of the procedure itself is. In such cases, a simple demon is set up that will copy the value from one template to the other when it becomes defined in the former.

---

[1] Looked at from the point of view that a number of possible template types are still allowed by the pseudo-class, it can be considered a form of implicit OR.

[2] The use of a pseudo-class makes the constraint more explicit than simply using consistency demons to check the template's class and type after the fact. This explicitness is important in defining the type of a template. However, this means that every rule that might do something with that template must be aware of the possible constraint.

[3] When more than one slot may be missing, a compound demon is used.

8.1.3 Compound Demons and the Rule Expander

A compound demon is used whenever the antecedents of a rule may have to wait for more than one slot to be filled in before being evaluated. The consequent or action part of a compound demon is procedural, as in all rules. The antecedents are written in a concise, declarative language that can express any pattern of program model templates and slots to be matched. Slot values may be bound and compared using a small number of predicates. Then the rule expander (or compiler)[4] translates the declarative form into a linear cascade of simple demons, each of which waits for one new slot value to become available, tests all antecedents that can now be tested at that point, and then sets up the next simple demon in the order.

The rule expander takes into account ordering constraints on the antecedents and makes sure that the rule consequent is executed as soon as all antecedents are satisfied. A partial ordering of the antecedents is determined, based upon the constraint that no antecedent can be evaluated until all of the variables that it references have been bound, often by another antecedent. From this partial ordering, one complete ordering of antecedents is chosen. By writing a compound demon and using the rule expander, rather than writing a sequence of simple demons by hand, the number of rules required has been reduced by factors from two to seventeen. When this expansion ratio is large, say, five or greater, the ease of writing and modifying the single compound demon becomes an important factor in maintaining and expanding the knowledge base.

Input to the rule expander is in the form of one logical PMB rule:

$$a_1 \wedge a_2 \wedge \ldots \wedge a_n \rightarrow c$$

Output takes the form of a compound demon implemented as n ordered simple demons (LISP functions). The order in which the antecedents are evaluated is denoted by the permutation function p below.

Format of Compound Demons

d_1: $a_{p(1)} \rightarrow d_2$
d_2: $a_{p(2)} \rightarrow d_3$
.
.
.
d_n: $a_{p(n)} \rightarrow c$

As an example, one might want to transform into tests all case statements that have exactly two mutually exclusive conditions. Such a rule might be expressed as shown below.

[4] Written by Steve T. Tappel

Example Compound Demon

if
(1) the statement is a case,
(2) the case has two condition/action pairs, and
(3) the first condition is the negation of the second condition,

then
change the case into a test.

As discussed in Chapter 6, there are two types of demons, consistency and transformation. The distinction is that transformation demons make structural changes to the program model (i.e., they add or remove entire templates), whereas consistency demons do not (i.e., they only access or store into existing slots). Examples of a compound demon of each type follow.

Example Consistency Compound Demon

This consistency demon is set up by the response rule for handling *is_subset* operations, which was discussed in the previous section on "Response Rules". The actual demon from PMB's rule base is listed below. The context of the demon is that IS-SUBSET is an *is_subset* primitive operation, a predicate of the form SUBCOLLECTION $\subseteq$ COLLECTION. An English paraphrase of the rule is

> Require that the SUBCOLLECTION and COLLECTION expressions, which are the two arguments of the *is_subset* operation called IS-SUBSET, both return collections of the same prototypic element.

Compound Demon for Checking Consistency of Is_Subset

```
(NAME IS-SUBSET-CONSISTENCY)
(TYPE CONSISTENCY)
(VARS IS-SUBSET)
(PATTERN (IS-SUBSET (SUBCOLLECTION = SUBCOLLECTION)
                    (COLLECTION = COLLECTION))
         (SUBCOLLECTION (TYPE-RETURNED = COLLECTION-1))
         (COLLECTION (TYPE-RETURNED = COLLECTION-2))
         (COLLECTION-1 (CLASS = 'AIU)
                       (TYPE = 'COLLECTION)
                       (AIU-FOR-ELEMENTS = AIU-FOR-ELEMENTS-1))
         (COLLECTION-2 (CLASS = 'AIU)
                       (TYPE = 'COLLECTION)
                       (AIU-FOR-ELEMENTS = AIU-FOR-ELEMENTS-2)))
(REQUIRE (AIU-FOR-ELEMENTS-1 = AIU-FOR-ELEMENTS-2))
(ELSEACTION (HELP IS-SUBSET
         "SUBCOLLECTION and COLLECTION have different prototypic elements."))
```

The rule expander translates this compound demon into twelve LISP functions that will be executed in order. The first function is given the name IS-SUBSET-CONSISTENCY and is the rule invoked with the name of the *is_subset* template of interest as a parameter in order to initialize the compound demon. When the first function is called, this template name is bound to the variable IS-SUBSET, which is declared in the VARS section of the listing above. The

first function is not a demon itself, but sets up the first simple demon. The next ten functions are simple demons that await the ten slot values referenced in the PATTERN section above. Finally, the twelfth and final function is created for the ELSEACTION.

The heart of a compound demon is its PATTERN part. A pattern defines relations on one or more slot values of one or more templates. For each template specified, there may be any number of slot triples, each consisting of a slot name, relation, and value. The slot name is implicitly quoted. The relation may be equality, set membership, and their negations. The value of a slot triple may be either a constant (which is explicitly quoted) or a variable (which is not quoted). The first time a variable is encountered—at runtime, not compile time—it is bound to the value of the slot.[5] After this occurrence, the relation will be tested to see if it holds, when the slot finally has a value. If the relation holds, the compound demon proceeds to the next test, which often means setting up the next simple demon. If the relation doesn't hold, then the ELSEACTION is executed.

A REQUIRE section takes an arbitrary LISP expression to be evaluated as soon as all of its external variables have bindings. This expression must be true; otherwise, the compound rule's ELSEACTION is executed.

The optional ACTION and ELSEACTION parts contain arbitrary LISP code. The ELSEACTION is executed if the PATTERN isn't matched successfully or if a REQUIRE statement is false. Otherwise, the ACTION is executed. The demon above doesn't need an ACTION because it is only making a consistency check.

Example Transformation Compound Demon

This transformation demon is set up by the response rule for handling *output* operations. Again, the actual demon is listed below. The context of the transformation demon is that OUTPUT is an *output* primitive operation that outputs AIU-INSTANCE to DESTINATION. An English paraphrase of the rule is

> If the *output* operation called OUTPUT is merely outputting a string constant to the user, then transform OUTPUT into an *inform_user* operation.

[5] The relation in this first occurrence must be equality.

Compound Demon for Transforming Output to Inform_User

```
(NAME OUTPUT-TO-INFORM-USER)
(TYPE TRANSFORM)
(VARS OUTPUT)
(PATTERN (OUTPUT (SUPER-OU = SUPER-OU)
                 (DESTINATION = 'USER)
                 (AIU-INSTANCE = REMEMBERED-VALUE))
         (REMEMBERED-VALUE (TYPE = 'REMEMBERED-VALUE)
                           (TYPE-RETURNED = STRING))
         (STRING (TYPE = 'PRIMITIVE)
                 (SPECIFIER = 'STRING)
                 (VALUE ~= NIL)))
(ACTION (DELETE-AM-TEMPLATE OUTPUT)
        (SET-AM-TEMPLATE OUTPUT (create INFORM-USER-TEMPLATE SUPER-OU←SUPER-OU
                                        MESSAGE←(GET-AM-SLOT STRING 'VALUE)))
        (ADD-TEMPLATE-REFERENCE AM-NAME 'POP 'INFORM-USER OUTPUT)
        (DELETE-FROM-AM-SLOT (GET-AM-SLOT REMEMBERED-VALUE 'AIU-INSTANCE)
                             'WHERE-REFERENCED REMEMBERED-VALUE)
        (DELETE-AM-TEMPLATE REMEMBERED-VALUE))
```

Here we see that ACTION is a list of LISP expressions to be evaluated. Usually they are restricted to a small set of primitives that manipulate the program model. This particular ACTION deletes two old templates and creates a new one to replace them.

8.2 Incremental Building

Since the specification process doesn't constrain the order in which topics are covered, PMB is capable of dealing with fragments received in virtually any order. There are two mechanisms for dealing with this problem: (1) demons and (2) response rules that respond appropriately whether or not a template that is referenced already exists. Examples of these were given earlier in this chapter.

8.3 Completeness

One major goal of PMB is to produce a complete program model. This means that every required piece of information about a construct that has been put in the model must be determined eventually. There are several ways to achieve completeness: by default, inference, and questioning. Cross-referencing is a required part of each model template and is done by PMB.

8.3.1 Default

The simplest way information can be determined is by default. If some detail is omitted from a fragment, PMB fills in a default value if one is known. For example, the format for inputting a collection will default to a LISP list. Size information about a collection is optional, hence the default is none.

A slot default is stored as soon as a template is created. But a question about that slot is still asked. Thus, if the question does get answered, the default value is overridden. If the question doesn't get answered, then the default value remains in force. Such a question is explicitly marked as an optional question in the list of outstanding questions. Optional questions need not be answered for model completeness. The user cannot override a previously stored slot value that isn't a default because no outstanding question will remain for that slot.

8.3.2 Inference: Type Coercion

The second way slots may be completed is by inference. The best example of completion by inference is the coercion of the types of information structures [Reynolds-69]. If the type of a formal parameter of a procedure isn't known, it will be inferred from the type of the corresponding actual parameter in a *procedure_instance*, and vice versa. The type of a referent (quantified variable) in a *true_for_some*, *true_for_all*, *for_some_do*, or *for_all_do* may be coerced by the way in which it is referenced in the body of the operation. In an *is_subset* operation if the names of both set arguments are known and the prototype of one of these instances is known, then the prototype of the other is inferred to be the same.

There is an interesting analogy between the propagation of data type constraints through a model (smart type coercion) and the propagation of constraints in an electronic circuit [Stallman & Sussman-77]. In the former case, a step in the propagation is based on what operations two data types enter into together. In the latter, it is derived from equations defining the relationships (e.g., voltages, current flow) between neighboring points in a circuit.

8.3.3 Questioning

The final—and usually the most frequent—way to attain completeness is to ask the external environment for more information. For example, when a loop is added to the model, PMB requests its exit conditions. When a collection is created, its prototypic element is immediately requested.

8.3.4 Cross-References

As a model is being built, cross-references are added. One type of cross-reference keeps track of the current scope of each information structure instance, i.e., the control structure that contains all references to the instance, but contains no other control structure that also contains all such references. Scope information is useful for variable allocation during coding.

Each instance also contains a list of each primitive operation that creates, destroys, references, or modifies it. This information is required for some consistency checks and transformations.

Global lists of all objects are maintained by class (e.g., control structure, information structure, and primitive operation) and type (e.g., loop, collection, and *add_element*). This allows all constructs of one type to be found quickly.

8.4 Semantic Consistency

After completeness, the next major goal of PMB is to guarantee the consistency of the model produced, i.e., that everything done in the model is legal with the respect to the semantics of the program modelling language and permissible with respect to the rest of the current model. This sort of consistency checking is one level smarter than the simple syntactic checks that are done by incremental compilers and some editors. Obviously this is only a step in the right direction. There are many possible tests that might be considered within the scope of "consistency", but most (e.g., formal program verification) require much more information from the user about the intent of the program. These are not covered here.

8.4.1 Consistency Checking

PMB does standard type checking and will complain, for example, if a number is given as the value of a string primitive. Similarly, an *is_element* operation, e.g., $x \in S$, requires that x be of the same type as the prototypic element of the collection S.

8.4.2 Inconsistency Resolution

There are situations in which inconsistencies are corrected by PMB. One example is the sorting out of prototype/instance ambiguities. The modelling language distinguishes between the prototype of an information structure and the one or more actual instances of it that are manipulated by the algorithm. For one of these instances (the "primary instance"), PMB allows fragments to skip defining the instance and make all data references directly to the prototype instead. This prototype/instance ambiguity is also called a type/token mixup.

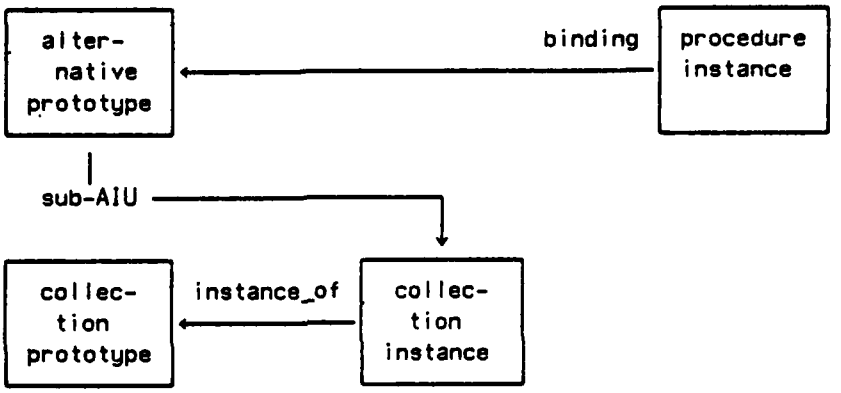
In the diagram shown below, two separable problems arise. In the first, fragments have defined, first, the prototype of an alternative and, second, a *procedure_instance* that uses the prototype as an actual parameter. Unfortunately prototypes aren't allowed as *procedure_instance* arguments. Since no instances of the alternative prototype exist, PMB creates one and changes the *procedure_instance* to use this new instance as its argument. This instance is marked as the primary instance in the information structure prototype so that, even if other instances are defined in the interim, future prototype references that should be references to an instance will be translated into the primary one. Two way cross-references are filled in also, as denoted by the additional arrowheads.

If a primary instance of the alternative had already existed, it would have been used. If no primary instance but exactly one regular instance had existed, PMB would have marked it as the primary instance instead of creating a new one. If more than one regular instance had existed, none marked primary, then PMB would have given up, since there are no heuristics at present to attempt a disambiguation.

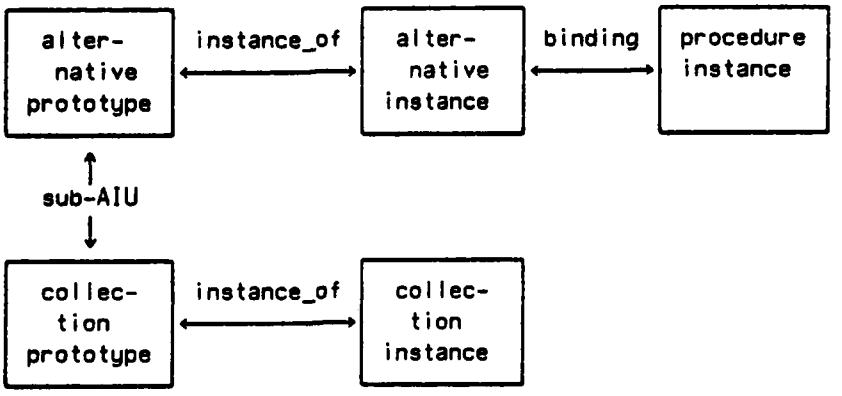
The second problem occurs in defining the alternative information structures (or sub-AIUs) of an alternative. In this case, one of them is defined as a collection instance (see diagram below). In the modelling language, information structure prototypes may be recursively defined only in terms of other information structure prototypes, not particular instances of them. In this case the collection prototype is known, so PMB changes the alternative prototype to point to it instead of the collection instance. If the prototype weren't known, then the instance would have been copied to another template, the prototype created in its place, and the instance marked as primary.

Resolution of Prototype/Instance Ambiguity

Program Fragments



Program Model



8.4.3 Specialization of Generic Operators

Similar to inference based on consistency is the specialization of generic operators. A *generic operator* is one that has several different but closely related semantics, depending upon the types of its arguments.[6] This problem can be thought of as the inverse of type coercion on variables. Instead of coercing operands according to what operators they are arguments of, the operator is coerced (specialized) based on the types of its arguments.

The fragment operation *are_equal* may result in one of the following primitive operations: *is_empty* (is a collection empty), *are_equal* (are two instances of the same prototype equal in value), *are_components_equal* (are two components of two instances of the same plex prototype equal), and *is_of_type* (is an alternative instance of a particular type).

The fragment operation *part_of* may be translated into one of the primitive operations *is_element*, *is_subset*, *has_correspondent* (does a domain element map into anything), or *is_component* (of a plex), depending upon its arguments and how they are represented in the program model:

$$\begin{aligned} \textit{part\_of}(x,Y) &\rightarrow x \in Y \\ &\rightarrow x \subseteq Y \\ &\rightarrow Y[x] \text{ defined?} \\ &\rightarrow \textit{is\_component}(x,Y) \end{aligned}$$

8.5 Canonization

PMB also has knowledge of program model equivalence transformations. This knowledge allows PMB to modify the normal results of model building in order to map constructs into higher level, more concise forms. The intent is to map equivalent expressions into one canonical form whenever they can be detected. The higher the level of an expression, the more implementation alternatives are afforded the coding process.

We term this type of model transformation *canonization* because equivalent constructs are transformed into the same canonical form. Other words don't convey quite the right meaning. "Abstraction" connotes a loss of detail. "Generalization" also connotes a loss of detail, e.g., as in learning programs and unification algorithms. "Simplification" connotes the small, localized, syntactic transformations, often contained in the simplifier module of a program verifier, that always result in a simpler (according to some metric) expression. Loveman has coined the term "evolution" to denote transformations that discover higher level constructs in lower level code [Loveman-77].

Below are the input and output from one transformation.[7]

---

[6] Since its semantics may take various forms, another name for this kind of operator is "polymorphic".

[7] Some other examples are discussed in [McCune-77].

$(\forall\ relation)\ relation \in concept \supset relation \in input\_scene$

$concept \subseteq input\_scene$

VERS2 [Earley-73A, Earley-73B, Earley-74] proposed similar transformations to convert element mapping operations into set operations, but the language was never implemented.

There are three possible ways to perform such transformations: (1) Immediately force the user to provide any additional information needed to proceed; (2) assume the information that is required if there is no evidence to the contrary (this may require inverse transformations or backing up later if the assumption proves unfounded); or (3) wait until all information is known and the transformation is permissible. PMB uses approach (3). Doing a transformation earlier wasn't felt critical enough to burden the user with answering additional questions immediately. And making assumptions is too dangerous to do without a model of when they are reasonable.

Canonization uses transformation demons to catch the obvious (and hopefully most useful) equivalences. A matching process occurs during the building of the model, looking for appropriate occasions in which to apply these transformations. A scheme to recognize a wider class of equivalences would need a theorem prover [Barstow-79B]. Canonization is done by attaching demons to a template whenever it may be possible to canonize it, e.g., to the *true_for_all* template (represented by ∀) in the example above. Attachment to a model template is done *after* any informalities have been removed (i.e., the template is a legal construct in the modelling language). This avoids having canonization rules that can also handle informalities.

Chapter 9. Conclusion

The thesis concludes with a description of the types of program models that have been successfully built by the Program Model Builder (PMB), a listing of the scientific contributions made by the research, and a discussion of limitations of the work and the directions for future research that these limitations suggest.

9.1 Program Models Built

PMB has built very high level program models for various versions of the algorithms described below. Most models were built during runs of the entire PSI program synthesis system. The table lookup model was constructed by PMB running separately.

An identity algorithm inputs and then outputs a single information structure (e.g., a set of record structures). Program models for this algorithm class make use of most of the information structure types in the program modelling language. The exception is the alternative, or union, structure. In addition, simple inferences about input/output operations are performed.

A membership algorithm first inputs a set of elements. Then the algorithm loops, inputting an element and testing whether it is in the set. This algorithm might be used to determine who should be admitted to a restricted place (e.g., a bank vault or a posh discotheque), based upon a set of names of authorized people. Models for this algorithm class use all of the control structures in the modelling language, plus the alternative information structure. Most of the expertise exhibited in the example of Chapter 4 is required for this class of models.

A subsetting algorithm inputs a set and then loops, inputting another set and testing whether the first is a subset of the second. This algorithm might be used to determine whether a set of job requirements is met by the set of qualifications of any job applicant. Program models for this class of algorithms are comparable in complexity to models for membership. However, in some cases there are more opportunities to transform expressions into canonical form. One model for subsetting contains 52 program constructs (information structures, control structures, and primitive operations).

A table lookup or search algorithm inputs a mapping (e.g., a set of ordered pairs of domain and range elements). The program then loops, inputting a domain element and printing the corresponding range element, if any. This algorithm could be used to look up recipes in a file that is indexed by recipe names. Program models for this algorithm class utilize much of the knowledge base about mappings and operations on them. One model for table lookup contains 54 modelling language constructs.

9.2 Contributions

We have presented a solution to one aspect of the program acquisition problem: the incremental construction of program models from informal descriptions. *Incremental* implies not only that information arrives a piece at a time, but that the pieces may arrive in an almost arbitrary order. *Informal* implies that even if all program pieces were put together, they might still not form an algorithm. The solution to this problem is a framework for incremental program acquisition that includes a language for expressing programs informally and incrementally (the *program fragment language*), a control structure for recognizing fragments in which new information arrives in an arbitrary order, and a knowledge base of rules for using this new information to update the program under construction. A computer system, the Program Model Builder, has been developed to test these aspects of the framework.

9.2.1 A Framework for Program Acquisition

Defining a framework for program acquisition is important. This is a new field, and not all of its aspects discussed herein have been fully appreciated previously. The general framework for program acquisition is described here briefly. Specifications are done at a very high level (compared to typical high level programming languages) and allow informalities such as minor incompleteness, inconsistencies, and ambiguities. These two features require the acquisition system to do more of the work in arriving at an efficient program, but the user to do less. The user must be in control of what is specified, how much is specified, and when it is specified. Specification is decoupled as much as possible from implementation considerations such as target language and target computer.

Our program acquisition framework assumes that transferring a program from human to computer requires the use of one or more languages for program specification. In addition to knowledge of these specification methods, any system of program acquisition requires knowledge of the programming language in which the acquired program is to be represented and knowledge of the domain of the program. It is possible to study, codify, and build systems that use the programming knowledge and interface with the other kinds of knowledge necessary for acquisition, yet not be constrained to work with any particular other kinds.

This thesis has explored one point in the space of this acquisition framework. A number of new techniques have been tried. In particular, the notion of combining informal specification with incremental specification is new. To allow this type of specification, a language for representing program fragments was designed and implemented. Then a pattern language for referring to parts of a program was designed when the need became evident. Supporting this type of specification has required the invention of incremental semantic consistency checking. Finally, the idea has been introduced of putting a program in canonical form to simplify subsequent automatic coding.

9.2.2 Program Fragment Language

The program fragment language supports two important notions, incremental and informal program specification. The informalities include incompleteness, inconsistency, and variety of specification. The language appears to have succeeded in its goal of providing a method for specifying the smallest amount of new information possible about a program. The fragment

language is also a start toward a representation of programming informality that is independent of a particular specification technique. This notion isn't conclusively a success, however, since the informalities have arisen mainly from observations of specifications using natural language and, secondarily, execution traces.

The pattern matching primitives of the *program reference language* provide a simple, uniform framework for accessing a piece of a program by a combination of textual, syntactic, contextual, historical, semantic, and pragmatic indices. This language is the result of ideas borrowed from systems such as text and program editors, combined with ideas derived from observations of how people actually specify programs informally. Since the program reference language is only in the early stages of incorporation into the program editor of the CHI program synthesis system [Phillips-79], the successor to PSI, no results are known. However, the need is clear and the idea seems promising.

9.2.3 Control Structure

PMB uses a problem solving technique that allows subgoals to be dealt with in an order defined by the user, rather than the system. This recognition method may be applicable to other problems that require a symbolic knowledge base to be acquired incrementally (e.g., speech understanding, image understanding, other signal processing domains, knowledge acquisition domains). Although subgoaling without any well-planned default for ordering the subgoals is not desirable, it is much closer to the ideal for recognition or knowledge acquisition than an absolute ordering that is fixed by the program.

Having only one demon for one antecedent active at a time (*linear demon activation*) is an efficient mechanism for those cases in which all antecedents must be true before any action is taken. Use of this method results in only one location in the database being monitored at a time. Evaluation time is linear in the number of conjuncts, even in the case in which some conjuncts sometimes become false after having been true. Average time for the typical method, in which all conjuncts are reevaluated whenever one may have changed, is quadratic in the number of conjuncts.

Demon priorities are useful when one class of demons must (locally) succeed before another class is allowed to awaken. The demon invocation mechanism was only implemented for two priority classes, but a general scheme for an arbitrary number of classes would be straightforward and useful.

9.2.4 Knowledge Base

PMB demonstrates the feasibility of having two forms of knowledge, a static base of rules about program acquisition and a dynamic base of templates that constitute the program that is being acquired. Except for the declarative antecedents used in compound demons, all rules are procedural. However, for the most part the types of processing done are quite restricted. It makes sense to extract the syntactic knowledge of the language in which program models are written (the *program modelling language*). This knowledge would be represented declaratively. The next step would be to develop a special language for doing incremental semantic consistency checks, program transformations, etc. Any processing that didn't fit the first two categories would be done by special purpose procedures.

The rule base has a fairly large array of rules about various programming constructs and associated consistency checks.[1] Incrementally acquiring and checking pieces of a program seems to work out well. A few rules for a number of different types of informalities have been written. The emphasis was on breadth rather than depth. The result is not conclusive in this area. Many more rules need to be tried before the problems will even be fully understood, let alone solved.

The technique used by the rule expander for "compiling" compound demons allows complicated antecedent patterns to be specified declaratively, yet executed efficiently as procedures. This technique is simply an application of the tenets of program acquisition to the building of the program acquisition system itself! Use of the rule expander has resulted in rules being written much faster and with fewer errors than when they were written out individually by hand.

9.2.5 Implementation

The implementation of PMB has provided a testbed for experimenting with our approach to the program acquisition problem. This experiment has been a qualified success. A number of programs have been successfully acquired by PMB working as part of the PSI program synthesis system. But a number of deficiencies have been observed in PMB's capabilities and implementation. These are outlined in the next section.

9.3 Limitations and Future Work

The limitations given below point out the most obvious and important future work needed on PMB and related research.

9.3.1 Role of Model Building in Other Systems

PMB assumes that program fragments come from other knowledge sources or possibly directly from the user. But there is no mechanism for feedback or sharing of the knowledge in the program model, except for PMB asking questions. Whether the best approach is a single acquisition system with many types of knowledge or a distributed system with a communications mechanism, the need for interchange of information is evident.

Just as in natural language understanding, the need for pragmatic domain support in program acquisition is real. Although a system may get by with just its programming knowledge, lack of knowledge about the specific application domain forces the user to do too much work. This has been one of the weakest points of the PSI system, and was probably a result of the lack of information sharing discussed in the preceding paragraph.

The role of the type of capabilities found in PMB should definitely be explored for less grandiose, programming aid systems such as program editors. Even if full-fledged automatic program acquisition systems never flourish, their concepts should be transported to the less automatic systems that may be practical today.

---

[1] Although such obvious consistency checks as producer-consumer analysis on variables are not done

9.3.2 Control Structure

The ability of the system to make changes to parts of a program that have already been acquired is very restricted. Once a fragment has been successfully processed, the result of that processing cannot be undone. The capability to modify the results of previous fragments is important, both for recovery from programming errors during initial specification and for general program modification at a later time. Straightforward backtracking or, worse, redoing the entire model building process with the new fragment are unappealing approaches. One solution might be to use heuristic rules that would examine the most likely causes of an error and then modify the model. Another possibility to be explored is to maintain a database of dependencies and make those changes that are necessary. Many data dependencies are already stored as cross-references in the program model. These could be used by specific rules for simple information structure modifications, e.g., changing an existing program by adding a new part to an information structure. When such a redefinition occurred, references to it by the algorithm would be updated to maintain the consistency of the program model. A similar approach has been taken in the domains of incremental circuit analysis [Stallman & Sussman-77] and program verification [Moriconi-77, Moriconi-79].

Work should be done on problem solving that is more flexible, that can be totally user controlled, system controlled, or some shade of each. In addition to the underlying problem solving techniques that will support such flexibility, models need to be developed so that the system can help determine at which end of the control spectrum it should be. At the user end of the spectrum, a user model of incremental, informal programming needs to be developed. In the area of greater system control, models of structured programming or other programming methodologies are needed.

Experience with the transformations made by PMB on fragments and models has pointed out the need for efficiency knowledge at the model level. As observed in [Long-77], it appears that efficiency considerations can never be completely divorced from the acquisition problem and hidden away in a lower level automatic coding system. In an acquisition system with a large body of transformations, more than one could be applicable to a part of the program being acquired. The choice of which transformation, if any, to perform may rest in part on high level "efficiency" heuristics. The right metric to be minimized is not obvious. Some possible metrics are (1) program complexity and length, (1) execution time of the program when interpreted using default data structure implementations for its high level information structures, (3) cost of subsequent acquisition, (4) cost of coding, and (5) execution times of potential coded target programs. Efficiency knowledge for the particular programming domain would also be useful. The new CHI system is one attempt to consider efficiency at a higher level than in PSI.

9.3.3 Program Modelling Language

An obvious limitation in almost any system of this type is the scope of the language accepted. Two examples of useful constructs not in the program modelling language at present are recursive procedures and arbitrary relations.

A general assertion mechanism would be a useful addition to PMB. Such assertions may be thought of as user supplied consistency checks that would be monitored either during model building or program execution. This type of assertion would consist of a predicate on elements

(i.e., *templates, slots, and values*) of the program model and on the associated state at runtime (e.g., the values of information structure instances, execution counts of statements, etc.). Such a predicate could be asserted globally or locally to one or more templates. The assertion would be evaluated however often and wherever in the model was necessary to assure that it remained true. Assertions could be so guaranteed during model building, during model interpretation on test data, or during execution of the coded version of the program, if appropriate statements were added to the model itself[2].

9.3.4 Knowledge Base

Much more detailed observation and precise codification of rules about programming informalities are needed. For informal specification to succeed, much richer languages for informality than are available now are needed. Otherwise, users get frustrated. An analogous problem has been observed with natural language understanding systems that aren't robust.

Any coding system has a finite amount of knowledge about how to implement various combinations of elements found in a program model. So there will always be cases in which two program models that behave the same but are syntactically different are implemented differently because the automatic coder didn't recognize some special case in one of them. In many of these cases, PMB should recognize the equivalence and transform them into a canonical form. On the other hand, putting a program in a canonical form may remove special cases about which the automatic coding system has special knowledge. In such cases canonization should not be done. The conclusion is that the two phases, if they are kept separate, should know about each other's capabilities so that they can work together, rather than be at odds.

Only a few rules dealing with canonization have been written. Experiments should be done using a sizable set of such rules in order to determine the gain in model conciseness and subsequent coding capability. Then the tradeoff between adding more rules at the model building versus coding level could be examined. Incorporating a general deduction mechanism to help recognize when canonization may be done might also be explored.

Rules for such things as type consistency and coercion should be merged. One rule could either attempt coercion or only check for consistency, depending upon models of how conservative the user wants the system to be and how successful the system has been previously.

A much smarter demon compiler would be very useful. It would allow the antecedents of a demon to be monitored either one at a time or in parallel. Demons could automatically be created to trigger as soon as the failure, as well as the success, or a compound antecedent was known.

Demons are just a form of simple concurrency, i.e., they are procedures blocking for an event (i.e., a change in a database) to occur. Other concepts from concurrent programming and operating systems (e.g., message passing) should be explored as ways to improve the technology of demons.

And finally, every good artificial intelligence system needs an explanation system and a

[2] The last method is a feature provided by ALGOLW via the ASSERT statement [Sites-72].

knowledge acquisition capability. The explanation system should be incremental, of course. It should provide answers to specific questions in the form of a program and/or natural language. The explainer should also be able to give a summary of the entire program.

9.4 Concluding Thoughts

The concepts and techniques that have been presented here will hopefully have an impact on such software systems areas as intelligent program editors and incremental compilers. The notion that a programming system should provide incremental semantic support, not just textual and syntactic support, should become more widespread. The idea of designing programming languages that allow informalities, not just higher and higher level primitives, is also an important avenue to explore.

It is my hope that the work described here, along with the related research of others, will lay the groundwork for the development of practical program acquisition systems. Without this advance, the promise of computers is destined to be limited by their accessibility only to the programmer elite.

Chapter 10. References

[Balzer et al.-74] Robert Balzer, Norton Greenfeld, Martin Kay, William Mann, Walter Ryder, David Wilczynski, and Albert Zobrist, "Domain Independent Automatic Programming", in Jack L. Rosenfeld, editor, Software, *Information Processing 74: Proceedings of IFIP Congress 74*, Volume 2, American Elsevier Publishing Company, Inc., New York, New York, 1974, pages 326-330.

[Balzer et al.-76] Robert Balzer, Neil Goldman, and David Wile, "On the Transformational Implementation Approach to Programming", *Proceedings, Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, pages 337-344.

[Balzer et al.-77] Robert Balzer, Neil Goldman, and David Wile, "Meta-Evaluation As a Tool for Program Understanding", *Fifth International Joint Conference on Artificial Intelligence—1977 (IJCAI-77): Proceedings of the Conference*, Volume 1, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, page 398-403.

[Balzer et al.-78] Robert Balzer, Neil Goldman, and David Wile, "Informality in Program Specifications", *IEEE Transactions on Software Engineering*, Volume SE-4, Number 2, March 1978, pages 94-103.

[Banatre et al.-79] J. P. Banatre, J. P. Routeau, and L. Trilling, "An Event Driven Compiling Technique", *Communications of the ACM*, Volume 22, Number 1, January 1979, pages 34-42.

[Barstow-79A] David R. Barstow, *Knowledge Based Program Construction*, Elsevier North Holland, Inc., New York, New York, 1979.

[Barstow-79B] David R. Barstow, "The Roles of Knowledge and Deduction in Program Synthesis", *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 37-43.

[Barth-74] C. Wrandle Barth, "Notes on the Case Statement", *Software—Practice and Experience*, Volume 4, Number 3, July-September 1974, pages 289-298.

[Biermann-76] Alan W. Biermann, "Approaches to Automatic Programming", in Morris Rubinoff and Marshall C. Yovits, editors, *Advances in Computers*, Volume 15, Academic Press, Inc., New York, New York, 1976, pages 1-63.

[Bobrow & Raphael-74] Daniel G. Bobrow and Bertram Raphael, "New Programming Languages for Artificial Intelligence Research", *Computing Surveys*, Volume 6, Number 3, September 1974, pages 153-174.

[Bobrow & Winograd-77] Daniel G. Bobrow and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language", *Cognitive Science*, Volume 1, Number 1, January 1977, pages 3-46.

[Bobrow et al.-77] Daniel G. Bobrow, Ronald M. Kaplan, Martin Kay, Donald A. Norman, Henry Thompson, and Terry Winograd, "GUS: A Frame Driven Dialog System", *Artificial Intelligence*, Volume 8, Number 2, April 1977, pages 155-173.

[Breitbard & Wiederhold-69] Gary Y. Breitbard and Gio Wiederhold, "The ACME Compiler", in A. J. H. Morrell, editor, Mathematics, Software, *Information Processing 68: Proceedings of IFIP Congress 1968*, Volume 1, North-Holland Publishing Company, Amsterdam, The Netherlands, 1969, pages 358-365.

[Brooks-75A] Ruven Brooks, *A Model of Human Cognitive Behavior in Writing Code for Computer Programs*, Ph.D. thesis, Psychology Department, technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1975.

[Brooks-75B] Ruven Brooks, "A Model of Human Cognitive Behavior in Writing Code for Computer Programs", *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, Volume 2, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1975, pages 878-884.

[Brooks-77] Ruven Brooks, "Towards a Theory of the Cognitive Processes in Computer Programming", *International Journal of Man-Machine Studies*, Volume 9, Number 6, November 1977, pages 737-751.

[Burstall & Darlington-77] R. M. Burstall and John Darlington, "A Transformation System for Developing Recursive Programs", *Journal of the Association for Computing Machinery*, Volume 24, Number 1, January 1977, pages 44-67.

[Carbonell-70A] Jaime R. Carbonell, *Mixed Initiative Man/Computer Instructional Dialogs*, Ph.D. thesis, Electrical Engineering Department, Massachusetts Institute of Technology, Report 1971, Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts, 31 May 1970.

[Carbonell-70B] Jaime R. Carbonell, "AI in CAI: An Artificial Intelligence Approach to Computer Assisted Instruction", *IEEE Transactions on Man-Machine Systems*, Volume MMS-11, Number 4, December 1970, pages 190-202.

[Charniak-72] Eugene Charniak, *Toward a Model of Children's Story Comprehension*, Ph.D. thesis, Electrical Engineering Department, TR-266, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1972.

[Dahl et al.-72] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, Inc., New York, New York, 1972.

[Davis et al.-77] Randall Davis, Bruce Buchanan, and Edward Shortliffe, "Production Rules As a Representation for a Knowledge Based Consultation Program", *Artificial Intelligence*, Volume 8, Number 1, February 1977, pages 15-45.

[Donzeau-Gouge et al.-75] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A Structure Oriented Program Editor: A First Step Towards Computer Assisted Programming", *International Computing Symposium 1975*, American Elsevier Publishing Company, Inc., New York, New York, 1975.

[Earley-73A] Jay Earley, "Relational Level Data Structures for Programming Languages", *Acta Informatica*, Volume 2, 1973, pages 293-309.

[Earley-73B] Jay Earley, *An Overview of the VERS2 Project: High Level Languages in Automatic Programming*, Memo ERL-M416, Electronics Research Laboratory, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, December 1973.

[Earley-74] Jay Earley, "High Level Operations in Automatic Programming", **Proceedings of a Symposium on Very High Level Languages**, *SIGPLAN Notices*, Volume 9, Number 4, April 1974, pages 34-42.

[Elschlager & Phillips-79] Robert Elschlager and Jorge Phillips, *Automatic Programming*, Memo HPP-79-24, Report STAN-CS-79-758, Heuristic Programming Project, Computer Science Department, Stanford University, Stanford, California, November 1979.

[Ginsparg-78] Jerrold M. Ginsparg, *Natural Language Processing in an Automatic Programming Domain*, Ph.D. thesis, Memo AIM-316, Report STAN-CS-78-671, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1978.

[Goldman et al.-77] Neil Goldman, Robert Balzer, and David Wile, "The Inference of Domain Structure from Informal Process Descriptions", **Proceedings of the Workshop on Pattern Directed Inference Systems**, *SIGART Newsletter*, Number 63, June 1977, pages 75-82.

[Green-69] Claude Cordell Green, *The Application of Theorem Proving to Question Answering Systems*, Ph.D. thesis, Electrical Engineering Department, Memo AIM-96, Report STAN-CS-69-138, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1969 [reprinted by Garland Publishing, Inc., New York, New York, 1979].

[Green-76A] Cordell Green, "The Design of the PSI Program Synthesis System", *Proceedings, Second International Conference on Software Engineering*, Computer Society, Institute of Electrical and Electronics Engineers, Inc., Long Beach, California, October 1976, pages 4-18.

[Green-76B] Cordell Green, "An Informal Talk on Recent Progress in Automatic Programming", *Lectures on Automatic Programming and List Processing*, PIPS-R-12, Electrotechnical Laboratory, Tokyo, Japan, November 1976, pages 1-69.

[Green et al.-74] C. Cordell Green, Richard J. Waldinger, David R. Barstow, Robert Elschlager, Douglas B. Lenat, Brian P. McCune, David E. Shaw, and Louis I. Steinberg, *Progress Report on Program Understanding Systems*, Memo AIM-240, Report STAN-CS-74-444, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, August 1974.

[Green et al.-79] Cordell Green, Richard P. Gabriel, Elaine Kant, Beverly I. Kedzierski, Brian P. McCune, Jorge V. Phillips, Steve T. Tappel, and Stephen J. Westfold, "Results in Knowledge Based Program Synthesis", *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 342-344.

[Hammer et al.-74] M. M. Hammer, W. G. Howe, and I. Wladawsky, "An Interactive Business Definition System", Proceedings of a Symposium on Very High Level Languages, *SIGPLAN Notices*, Volume 9, Number 4, April 1974, pages 25-33.

[Hayes & Simon-74] J. R. Hayes and H. A. Simon, "Understanding Written Problem Instructions", in Lee W. Gregg, editor, *Knowledge and Cognition*, Halsted Press Division, John Wiley and Sons, Inc., New York, New York, 1974, pages 167-200.

[Heidorn-72] George E. Heidorn, *Natural Language Inputs to a Simulation Programming System*, Ph.D. thesis, Report NPS-55HD72101A, Naval Postgraduate School, Monterey, California, October 1972.

[Heidorn-74] George E. Heidorn, "English As a Very High Level Language for Simulation Programming", Proceedings of a Symposium on Very High Level Languages, *SIGPLAN Notices*, Volume 9, Number 4, April 1974, pages 91-100.

[Heidorn-75] George E. Heidorn, "Simulation Programming through Natural Language Dialog", in Murray A. Geisler, editor, Logistics, *North-Holland/TIMS Studies in the Management Sciences*, Volume 1, North-Holland Publishing Company, Amsterdam, The Netherlands, 1975, pages 71-85.

[Heidorn-76] G. E. Heidorn, "Automatic Programming through Natural Language Dialog: A Survey", *IBM Journal of Research and Development*, Volume 20, Number 4, July 1976, pages 302-313.

[Hewitt & Smith-75] Carl E. Hewitt and Brian Smith, "Towards a Programming Apprentice", *IEEE Transactions on Software Engineering*, Volume SE-1, Number 1, March 1975, pages 26-45.

[Hobbs-77A] Jerry R. Hobbs, *From "Well Written" Algorithm Descriptions into Code*, Research Report 77-1, Computer Sciences Department, City College, City University of New York, New York, New York, July 1977.

[Hobbs-77B] Jerry R. Hobbs, "What the Nature of Natural Language Tells Us about How to Make Natural Language–Like Programming Languages More Natural", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 85-93.

[Kant-79A] Elaine Kant, "A Knowledge Based Approach to Using Efficiency Estimation in Program Synthesis", *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 457-462.

[Kant-79B] Elaine Kant, *Efficiency Considerations in Program Synthesis: A Knowledge Based Approach*, Ph.D. thesis, Memo AIM-331, Report STAN-CS-79-755, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, Technical Report SCI.ICS.U.79.1, Computer Science Department, Systems Control, Inc., Palo Alto, California, September 1979.

[Kennedy & Schwartz-75] K. Kennedy and J. Schwartz, "An Introduction to the Set Theoretical Language SETL", *Computers and Mathematics, with Applications*, Volume 1, Number 1, 1975, pages 97-119.

[Kibler-78] Dennis Francis Kibler, *Power, Efficiency, and Correctness of Transformation Systems*, Ph.D. thesis, Computer Science Department, University of California, Irvine, California, 1978.

[Kibler et al.-77] D. F. Kibler, J. M. Neighbors, and T. A. Standish, "Program Manipulation via an Efficient Production System", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 163-173.

[Knuth-69] Donald E. Knuth, Seminumerical Algorithms, *The Art of Computer Programming*, Volume 2, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1969.

[Knuth-73A] Donald E. Knuth, Fundamental Algorithms, *The Art of Computer Programming*, Volume 1, Second Edition, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[Knuth-73B] Donald E. Knuth, Sorting and Searching, *The Art of Computer Programming*, Volume 3, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[Knuth-74] Donald E. Knuth, "Structured Programming with Go To Statements", *Computing Surveys*, Volume 6, Number 4, December 1974, pages 261-301 [reprinted in Raymond T. Yeh, editor, Software Specification and Design, *Current Trends in Programming Methodology*, Volume 1, Chapter 6, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977, pages 140-194].

[Lenat-76] Douglas B. Lenat, *AM: An Artificial Intelligence Approach to Discovery in Mathematics As Heuristic Search*, Ph.D. thesis, Memo AIM-286, Report STAN-CS-76-570, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, July 1976.

[Lenat-77] Douglas B. Lenat, "Automated Theory Formation in Mathematics", *Fifth International Joint Conference on Artificial Intelligence—1977 (IJCAI-77): Proceedings of the Conference*, Volume 2, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, pages 833-842.

[Lesser & Erman-77] Victor R. Lesser and Lee D. Erman, "A Retrospective View of the HEARSAY-II Architecture", *Fifth International Joint Conference on Artificial Intelligence—1977 (IJCAI-77): Proceedings of the Conference*, Volume 2, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, pages 790-800.

[Lesser et al.-75] Victor R. Lesser, Richard D. Fennell, Lee D. Erman, and D. Raj Reddy, "Organization of the HEARSAY-II Speech Understanding System", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Volume ASSP-23, Number 1, February 1975, pages 11-24.

[London-77] Ralph L. London, "Perspectives on Program Verification", in Raymond T. Yeh, editor, **Program Validation**, *Current Trends in Programming Methodology*, Volume 2, Chapter 6, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977, pages 151-172.

[Long-77] William James Long, *A Program Writer*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, TR-187, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1977.

[Loveman-77] David B. Loveman, "Program Improvement by Source-to-Source Transformation", *Journal of the Association for Computing Machinery*, Volume 24, Number 1, January 1977, pages 121-145.

[Luckham-77] David C. Luckham, "Program Verification and Verification Oriented Programming", invited paper, in Bruce Gilchrist, editor, *Information Processing 77: Proceedings of IFIP Congress 77*, Elsevier/North-Holland Publishing Company, Inc., New York, New York, 1977, pages 783-793.

[Masinter-79] Larry M. Masinter, *Global Program Analysis in an Interactive Environment*, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, California, September 1979.

[McCune-77] Brian P. McCune, "The PSI Program Model Builder: Synthesis of Very High Level Programs", **Proceedings of the Symposium on Artificial Intelligence and Programming Languages**, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 130-139.

[Minsky-75] Marvin Minsky, "A Framework for Representing Knowledge", in Patrick Henry Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill Book Company, Inc., New York, New York, 1975, pages 211-277.

[Moriconi-77] Mark S. Moriconi, *A System for Incrementally Designing and Verifying Programs*, Ph.D. thesis, University of Texas, Austin, Texas, Volume 1, Research Report RR-77-65, Information Sciences Institute, University of Southern California, Marina del Rey, California, November 1977.

[Moriconi-79] Mark S. Moriconi, "A Designer/Verifier's Assistant", *IEEE Transactions on Software Engineering*, Volume SE-5, Number 4, July 1979, pages 387-401.

[Nelson-76] Bruce Nelson, *The PSI Interpreter*, M.S. project report, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1976.

[Phillips-77] Jorge V. Phillips, "Program Inference from Traces Using Multiple Knowledge Sources", *Fifth International Joint Conference on Artificial Intelligence—1977 (IJCAI-77): Proceedings of the Conference*, Volume 2, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, page 812.

[Phillips-79] Jorge V. Phillips, *Knowledge Based Algorithm Development*, Ph.D. thesis, Electrical Engineering Department, Stanford University, Stanford, California, technical report, Computer Science Department, Systems Control, Inc., Palo Alto, California, 1979 (in progress).

[Pressburger-78] Thomas T. Pressburger, *The Readable Program Model Generator*, M.S. project report, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, June 1978.

[Reynolds-69] John C. Reynolds, "Automatic Computation of Data Set Definitions", in A. J. H. Morrell, editor, Mathematics, Software, *Information Processing 68: Proceedings of IFIP Congress 1968*, Volume 1, North-Holland Publishing Company, Amsterdam, The Netherlands, 1969, pages 456-461.

[Rich-79] Charles Rich, *A Library of Programming Plans with Applications to Automated Analysis, Synthesis, and Verification of Programs*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1979 (in progress).

[Rich & Shrobe-78] Charles Rich and Howard E. Shrobe, "Initial Report on a LISP Programmer's Apprentice", *IEEE Transactions on Software Engineering*, Volume SE-4, Number 6, November 1978, pages 456-467.

[Rich et al.-79] Charles Rich, Howard E. Shrobe, and Richard C. Waters, "Overview of the Programmer's Apprentice", *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 2, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 827-828.

[Rieger-77] Chuck Rieger, "Spontaneous Computation in Cognitive Models", *Cognitive Science*, Volume 1, Number 3, July 1977, pages 315-354.

[Robinson & Parnas-73] L. Robinson and D. L. Parnas, *A Program Holder Module*, technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June 1973.

[Rulifson et al.-72] Johns F. Rulifson, Richard J. Waldinger, and Jan A. Derksen, "A Language for Writing Problem Solving Programs", in C. V. Freiman, editor, Foundations and Systems, *Information Processing 71: Proceedings of IFIP Congress 71*, Volume 1, North-Holland Publishing Company, Amsterdam, The Netherlands, 1972, pages 201-205.

[Schonberg et al.-79] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir, "Automatic Data Structure Selection in SETL", *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, Association for Computing Machinery, New York, New York, January 1979, pages 197-210.

[Schwartz-74] J. Schwartz, Structureless Programming, or The Notion of "Rubble" and the Reduction of Programs to Rubble, *SETL Newsletter*, Number 135A, SETL Project, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, New York, New York, 12 July 1974.

[Schwartz-75] Jacob T. Schwartz, *On Programming: An Interim Report on the SETL Project*, revised, technical report, SETL Project, Computer Science Department, Courant Institute of Mathematical Sciences, New York University, New York, New York, June 1975.

[Schwartz-78] J. T. Schwartz, "Program Genesis and the Design of Programming Languages", in Raymond T. Yeh, editor, Data Structuring, *Current Trends in Programming Methodology*, Volume 4, Chapter 7, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978, pages 185-215.

[Selfridge-59] O. G. Selfridge, "Pandemonium: A Paradigm for Learning", *Mechanization of Thought Processes*, Volume 1, Her Majesty's Stationery Office, London, England, 1959, pages 511-531.

[Shortliffe-76] Edward Hance Shortliffe, *Computer Based Medical Consultations: MYCIN*, American Elsevier Publishing Company, Inc., New York, New York, 1976.

[Shrobe-79A] Howard Elliot Shrobe, *Dependency Directed Reasoning for Complex Program Understanding*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, TR-503, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1979.

[Shrobe-79B] Howard Elliot Shrobe, "Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures", *IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Volume 2, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 829-835.

[Sites-72] Richard L. Sites, *ALGOLW Reference Manual*, Report STAN-CS-71-230, Computer Science Department, Stanford University, Stanford, California, February 1972.

[Stallman & Sussman-77] Richard M. Stallman and Gerald J. Sussman, "Forward Reasoning and Dependency Directed Backtracking in a System for Computer Aided Circuit Analysis", *Artificial Intelligence*, Volume 9, Number 2, October 1977, pages 135-196.

[Steinberg-79] Louis I. Steinberg, *A Dialog Moderator for Program Specification Dialogs in the PSI System*, Ph.D. thesis, Artificial Intelligence Laboratory, Computer Science Department, Stanford University, Stanford, California, 1979 (in progress).

[Tappel-79] Steve T. Tappel, *Intelligent Guidance of Algorithm Design*, technical report, Computer Science Department, Systems Control, Inc., Palo Alto, California, 1979 (in progress).

[Teitelman-72A] Warren Teitelman, "'Do What I Mean': The Programmer's Assistant", *Computers and Automation*, Volume 21, Number 4, April 1972, pages 8-11.

[Teitelman-72B] Warren Teitelman, "Automated Programmering: The Programmer's Assistant", 1972 Fall Joint Computer Conference, *AFIPS Conference Proceedings*, Volume 41, Part 2, AFIPS Press, Montvale, New Jersey, December 1972, pages 917-921.

[Teitelman-77] Warren Teitelman, "A Display Oriented Programmer's Assistant", *Fifth International Joint Conference on Artificial Intelligence–1977 (IJCAI-77): Proceedings of the Conference*, Volume 2, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1977, pages 905-915.

[Teitelman-78] Warren Teitelman, *INTERLISP Reference Manual*, Palo Alto Research Center, Xerox Corporation, Palo Alto, California, October 1978.

[Van Wijngaarden et al.-69] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, "Report on the Algorithmic Language ALGOL68", *Numerische Mathematik*, Volume 14, Number 2, 1969, pages 79-218.

[Waters-78] Richard C. Waters, *Automatic Analysis of the Logical Structure of Programs*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, TR-492, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1978.

[Waters-79] Richard C. Waters, "A Method for Analyzing Loop Programs", *IEEE Transactions on Software Engineering*, Volume SE-5, Number 3, May 1979, pages 237-247.

[Wilczynski-75] David Wilczynski, *A Process Elaboration Formalism for Writing and Analyzing Programs*, Ph.D. thesis, Computer Science Department, Research Report RR-75-35, Information Sciences Institute, University of Southern California, Marina del Rey, California, October 1975.

[Wile et al.-77] David Wile, Robert Balzer, and Neil Goldman, "Automated Derivation of Program Control Structure from Natural Language Program Descriptions", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Volume 12, Number 8, *SIGART Newsletter*, Number 64, August 1977, pages 77-84.

[Winograd-74] Terry Winograd, "Breaking the Complexity Barrier *Again*", in Richard E. Nance, editor, Proceedings of ACM SIGPLAN-SIGIR Interface Meeting: Programming Languages–Information Retrieval, *SIGIR FORUM*, Volume 9, Number 3, Winter 1974, *SIGPLAN Notices*, Volume 10, Number 1, January 1975, pages 13-30.

[Winston-75] Patrick Henry Winston, "Learning Structural Descriptions from Examples", in Patrick Henry Winston, editor, *The Psychology of Computer Vision*, McGraw-Hill Book Company, Inc., New York, New York, 1975, pages 157-209.

[Wirth-73] Niklaus Wirth, *Systematic Programming: An Introduction*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[Yonke-75] Martin D. Yonke, *A Knowledgeable, Language Independent System for Program Construction and Modification*, Ph.D. thesis, Computer Science Department, University of Utah, Salt Lake City, Utah, Research Report RR-75-42, Information Sciences Institute, University of Southern California, Marina del Rey, California, October 1975.

[Zahn-74] Charles T. Zahn, Jr., "A Control Statement for Natural Top Down Structured Programming", in B. Robinet, editor, *Programming Symposium: Proceedings, Colloque sur la Programmation*, Springer-Verlag, New York, New York, 1974, pages 170-180.

Appendix A. Proposed Program Reference Language

As introduced in Chapter 5, the program reference language is designed to provide a "where" capability for program fragments. A program reference specification given in the program reference language doesn't necessarily identify a unique point in the program model. A reference specification is just a pattern to be matched against the model in order to constrain which model templates are considered.[1] A specification can be universally quantified, which indicates that *all* templates that match should be transformed by the "what" part of the fragment. If a specification is not universally quantified, then there is an error condition if more than one template matches (i.e., the specification is ambiguous).[2]

One way to view a universally quantified fragment is as a demon that monitors those points in the program (or conditions during execution) and is executed when triggered by the appropriate conditions. Seen in this light, fragments are related to Schwartz's notion of independent, parallel chunks or "rubble" [Schwartz-74, Schwartz-78]. However, whereas programs written in terms of pieces of rubble would presumably be compiled or interpreted all at once, fragments are *incrementally* integrated into a modelling language that uses standard *sequential* control structures.

A program reference pattern is a declarative specification of a set of relevant program model templates: no particular search strategy is presumed for how the pattern match is implemented. For example, consider a pattern that specifies all loops in the model with a particular property. A straightforward pattern matcher might match this pattern to the model by searching every template in the model for those that are loops with the required property. A smarter matcher would know that the root template of the program model contains cross-references to every template by type[3] and would therefore limit its search to the list of all loop templates. So we observe that this small part of the great automatic programming problem may be viewed as an automatic programming problem itself, complete with program specification and optimization considerations.

Besides its utility in automatic programming systems, the program reference language may be of use in research on intelligent program editors. The reference capabilities described are related to those provided by many interactive text editors, especially those for specific programming languages. Most general purpose text editors only provide low level editing primitives that treat their data as an arbitrary character string[4]. Language oriented editors introduce syntactic and contextual editing, eliminating most of the need and perhaps even the capability for purely textual editing. An excellent example of such an editor is the one embedded in the INTERLISP system [Teitelman-78]. INTERLISP also has limited methods for specifying

---

[1] The program reference language is a special purpose pattern matching language. Much more general matching schemes are described for KRL [Bobrow & Winograd-77].

[2] Another possibility would be to order the templates and use the one that was the "best" match.

[3] These cross-references correspond to the notion of index keys in KRL [Bobrow & Winograd-77].

[4] Or even worse, the data is treated as a list of individual lines of text.

historically and semantically, using the "history list" feature of the Programmer's Assistant [Teitelman-72B] and the program analysis package called MASTERSCOPE [Masinter-79]. These three subsystems of INTERLISP have somewhat differing goals from the present effort, and in any case are provided via separate mechanisms utilizing separate languages[5].

The rest of this appendix defines the capabilities envisioned for the program reference language by the types of references allowed.

A.1 Textual References

A point in a program can easily be specified (partially, at least) by its position in a text string representing the program. The two standard ways are by line number and by substring match. In the program reference language both of these methods make use of the "readable" program model, a compressed, linear form of the model that can be printed for the user upon request [Pressburger-78]. The model is represented in a PASCAL-like notation that optionally includes line numbers. This form of the model will be referred to as the model *listing* to distinguish it from the program model itself.

The pattern

line 500

in the program reference language limits the search for the specified program model template to those templates represented in line 500 of the readable model listing.

A pattern consisting of the character string

"output"

matches all templates represented by the word "output" in the readable model listing.[6] The universally quantified pattern

∀ "output"

results from a user command such as "Change 'output' to 'print' everywhere.". Note the distinction between these two patterns. Both may match zero or more templates, but the former pattern requests that the associated action in the fragment be done to exactly one template, while the latter pattern requests that it be done to all matching templates.

Most types of patterns can be combined to form conjunctive expressions. For example,

line 2100, "output"

---

[5] The most interesting one is MASTERSCOPE's fill-in-the-blank command language, which accepts limited English phrases.

[6] We say "represented by" because the string pattern itself may never occur in the actual program model. For example, the reserved words "begin" and "end" are used in the readable model listings to represent templates of type composite.

matches only model templates represented by the occurrences of the string "output" in line 2100 of the model listing.

The pattern variable "." is used to match arbitrary individual characters, with a superscript specifying the number to be matched. "." or ".0" matches zero characters, hence is equivalent to concatenating the neighboring strings. The pattern

"begin" $.^{10}$ "end"

will match all blocks with exactly ten characters between the starting and ending keywords. Borrowed from the language of regular expressions, the superscript variable "*" matches zero or more characters, and "+", one or more. For example, the pattern

"begin" .* "end"

matches all blocks.

A pattern may be subscripted by a sequence of numbers specifying which matches of the pattern are to be retained. To do this, a starting point and an ordering on potential matches must be specified. In the case of string matches, this ordering starts at the beginning of text and proceeds linearly through the text to the end. For example,

"output"$_5$

matches only the template corresponding to the fifth occurrence of the string "output" statement in the listing. As a more general example,

"output"$_{1,3:5,N-3}$

matches the first, third, fourth, and fifth templates from the beginning and the fourth template from the last corresponding to "output". We could also extend the syntax to allow the *exclusion* of certain templates that match. As later examples show, subscripts may be added to any expression to constrain the extent of the match.

A.2 Syntactic (Lexical) References

Specifying program parts syntactically requires a shift from specifying text in the model listing to specifying templates in the program model itself. The model is essentially a parse tree of the program, with templates as nodes in the tree. The closest related work to this is the Find command with patterns in the INTERLISP editor [Teitelman-78, Section 9.3.2].

The simplest way to specify a model template is by giving its unique name. For example,

template *update*

specifies the unique template called *update* in the model.

If the name of the template isn't known, then the values of slots may be used to identify the desired template. The phrase "where x is output to disk or tape" may be represented as

template *n* | *n*.type=output, *n*.AIU_instance=*x*, *n*.destination∈{disk,tape}

Pattern match expressions are separated by commas representing logical conjunction. Expressions are matched in order, left to right, to avoid potential ambiguities.[7] A way to visualize the matching process is to consider it as a series of constraints applied to the set of all templates in the model. The first conjunct reduces the set of potentially matching templates to a subset of all the templates. The next conjunct further reduces the size of this subset (constrains the match), and so on until the last conjunct is applied and the final set of matching templates is left.

A slot value expression is of the form

<template>.<slot> <operator> <value>

where the operator can be equality, inequality, set membership, set nonmembership, and other operations where appropriate. If a name appearing in the value position of some expression also appears in the template position of another expression, then that name is assumed to be a variable; otherwise, it is assumed to be a constant.

If a slot name (e.g., type) isn't preceded by a template name, then the template named between **template** and "|" at the beginning of the pattern is assumed. Thus the following pattern is equivalent to the one above:

template *n* | type=output, AIU_instance=*x*, destination∈{disk,tape}

If the action of the fragment is to apply to all such *output* statements, the pattern used is

∀ **template** *n* | type=output, AIU_instance=*x*, destination∈{disk,tape}

Sometimes one wants to obtain a slot value some templates away from the current template, and the names of the intervening templates aren't needed for other purposes. In such a case, slot names may be strung together in one expression, separated by periods. Thus, the following two patterns are equivalent.

template *n* | *n*.true_action=*t*, *t*.body=*b*, *b*.type=composite
template *n* | *n*.true_action.body.type=composite

Of course, as in any pattern match, if the value of the slot doesn't match or the slot doesn't have a value at all, the match fails.

Subscripts may be used to limit the matches that occur with these conjuncts. The default ordering is lexical order, i.e., the depth first traversal order that templates are visited when a listing is being created. Because all information structures and procedures are considered global in the program modelling language, lexical order puts all information structure prototypes first, then information structure instances, then procedure declarations, and finally the main algorithm body. The following three examples demonstrate the power of subscripting:

[7] An example of such an ambiguity appears later in this section.

template n | (n.type=input, n.type_returned=a)$_3$
template n | (n.type=input)$_3$, n.type_returned=a
template n | (n.type=input)$_3$, (n.type_returned=a)$_3$

The first expression matches the third *input* operation of the model that returns a value of type a; otherwise, no template is matched. The second expression is more restrictive. It matches the third *input* operation if that *input* returns a value of type a; otherwise, no template is matched. The final expression can't match anything. It first selects a set of templates containing only the third *input* operation (if such exists) and then tries to select the third element of this set that returns a value of type a. Obviously there can't be a third element in a set with only one member.

Constraining a group of templates to some order is done by enclosing the sequence of templates in braces and separating the template names by an ordering relation (or pattern variable). For lexical ordering, the pattern variable is a period. An unsuperscripted period may be omitted. For example, the two patterns

template n | type=output, {*foo* . *bar* . n}, *foo*.type=composite, *bar*.type=test
template n | type=output, {*foo bar* n}, *foo*.type=composite, *bar*.type=test

are equivalent and match all *output* operations that immediately follow a test that immediately follows a composite.

Similar to the string matching in the previous section, determinate (with a fixed number of elements) and indeterminate (with either zero or more or one or more elements) sequences of templates can be specified; e.g.,

template n | {*foo* .* *bar* .^{2} n}, *foo*.type=input, *bar*.type=output

specifies the template exactly three templates lexically after an *output* that lexically follows an *input*.[8]

If matches are restricted by subscripting, then ambiguous patterns can occur unless the evaluation of conjuncts is limited to a particular order, in this case left to right. For example, in the pattern

template n | {a .* n}, (type=test)$_2$

a left-to-right matching order finds the second *test* after a, while right-to-left order will find no matches if two or more *test*s occur before a.

Template paths through the program model tree are specified as sequences of template names separated by "↓"s. $A \downarrow b$ means that b is a child of a.[9] $A \downarrow^5 b$ means that there are exactly five intervening templates between a and b. $A \downarrow^* b$ means that there is an indeterminate number of

[8] Thus, a single "." is equivalent to ".0".

[9] If one knew that s was the name of the slot in a that pointed to b, then an equivalent form would be $a.s = b$. However, the form $a \downarrow b$ is more general, since it allows b to be located in any slot of a that can point to a child. Typically there is more than one.

intervening templates. "The statement that outputs a collection of size less than twenty" is expressible as

template *n* | type=output, AIU_instance=*x*, {*y* ↓ *x*}, *y*.type=collection, *y*.size<20

"The *output* statement that occurs in a conditional that is three levels down inside some block" becomes

template *n* | type=output, {*foo* $\downarrow^2$ *bar* $\downarrow^*$ *n*}, *foo*.type=composite, *bar*.type=test

Here we also note the following equivalence, for arbitrary pattern variable "•" (e.g., ".", "↓"):

template *y* | ({*x* $\bullet^*$ *y*})$_i$ ≡ **template** *y* | {*x* $\bullet^{i-1}$ *y*}

The lefthand side matches all templates *y* "after" *x* and then selects the *i*th one. The righthand side directly selects the *i*th template after *x*.

A.3 Contextual References

Contextual reference, i.e., specifying a place in the program relative to the last place discussed, is often extremely useful. To refer to the current template explicitly, we introduce the symbol "⋆".[10] We can now refer, for example, to templates before, after, below, and above the current one. The first example below specifies the template that lexically follows the current one. If the current template is the first part of a composite, this statement would specify the second part. Also note the last example, which allows one to specify the closest test template above the current template.

template *n* | {⋆ . *n*}
template *n* | {*n* .$^+$ ⋆}
template *n* | {⋆ $\downarrow^*$ *n*}
template *n* | ({*n* $\downarrow^*$ ⋆}, type=test)$_1$

Another type of useful reference constrains the pattern to match the "closest" template in any direction to the current template context. Closeness is defined in terms of the number of intervening templates along the path between the two templates in question. We introduce two symmetric pattern variables that provide two ways to define paths. *A* .. *b* means that *a* occurs either before or after *b* in lexical order. *A* ↓↑ *b* means that *a* occurs either on the path from the root node of the program model tree to *b* or somewhere in the subtree below *b*. If two or more templates tie for closest, then they all match.

template *n* | ({⋆ .. *n*}, type=composite)$_1$
template *n* | ({⋆ ↓↑ *n*}, type=test)$_1$

The first example above specifies the closest composite lexically to the current location. The second specifies the closest test that either contains or is contained in the current template.

---

[10] This context variable provides a simple but useful form of the KRL notion of a "focus list" of templates [Bobrow & Winograd-77].

A.4 Historical References

Referring to the program model in chronological order of specification requires maintenance of a history list [Teitelman-72B] of fragments (not necessarily the same as the tree of more general topics discussed, cf. the PSI dialog moderator [Steinberg-79]). Along with each fragment, a list of model templates that were affected by the fragment is kept.

Now we introduce the pattern variable "<". *A* < *b* constrains template *a* to occur in the history list immediately before template *b*. Here are some examples of its use:

template *n* | {*n* < ⊛}
template *n* | {*n* <* ⊛}
template *n* | {*n* <^{3} ⊛}
template *n* | ({*n* <* ⊛}, type=test)$_1$

Since "⊛" is the current template, it always refers to the latest entry in the history list. For example, the last expression above refers to the most recently discussed test, excluding the current template.

Historical patterns may constrain matching of the history list to an appropriate order. For this purpose, we introduce the pattern variable "<>". *A* <> *b* means that *b* occurs in the history list somewhere either before or after *a*. As examples, the phrases "the most recent output statement discussed", "the earliest output statement discussed", and "the closest output statement to the point where template *a* was discussed" might be expressed as

template *n* | ({*n* <* ⊛}, type=output)$_1$
template *n* | ({*n* <* ⊛}, type=output)$_N$
template *n* | ({*a* <> *n*}, type=output)$_1$

A.5 Semantic References

Templates can be specified semantically by their functionality in the model and by control and dataflow. Simple forms of these references are available in the MASTERSCOPE package of INTERLISP [Masinter-79]. The notion of semantic functions is exemplified below by *type*, *returns*, and *referents*, but this is not a complete list of the necessary functions.

As an example of reference by function, the following pattern finds all (control structure) templates that contain below them a set operation that returns a Boolean value.

template *n* | {*n* ↓* *x*}, *type*(*x*)∈set_operations, *returns*(*x*)=Boolean

The functional notation used for *type* and *returns* indicates that these are not simple slot values of template *x*, but possibly have to be computed. The following pattern describes all procedures that return a value of type *v*:

template *n* | type=procedure, *returns*=*v*

Control and dataflow order are specified by the "→", "→^{n}", "→*", and "→+" patterns, analogous to

the forms for the ".", "↓", and "<" pattern variables discussed earlier.[11] Whether "→" refers to requires a data or control flow ordering depends upon the context in which it occurs. Here is a pattern that specifies all procedures that may be called before template x is executed:

template n | type=procedure, y.instance_of=n, $\{y \rightarrow^* x\}$

The dataflow examples below specify the first reference to a, the last reference to a, the template that references both a and b, and the template that references a before it is output. The default order for matching is execution order.

template n | $(a \in referents(n))_1$
template n | $(a \in referents(n))_N$
template n | $referents(n)=\{a,b\}$
template n | $a \in referents(n)$, $\{n \rightarrow^* x\}$, x.type=output, x.AIU_instance=a

We could also introduce a "→←" pattern variable analogous to "..", "↓↑", and "<>", but its utility isn't clear.

A.6 Pragmatic References

Referring to a part of a program pragmatically, i.e., by its function or purpose, is probably the most common and useful program reference technique when the user isn't dealing directly with the program itself (e.g., by using an editor of some kind). However, this conclusion is based on an examination of a small number of natural language program specification dialogs held by only a few different people. Unfortunately, pragmatic references require domain knowledge, of which PMB has little. The only way PMB can handle specifications such as "the *output* statement that lists the updated database" is via trivial slot values that provide some domain specific context:

template n | type=output, AIU_instance=*database*, $\{a \downarrow^* n\}$, a.user_name=update
template n | type=output, purpose=output_database

Each template currently has an optional user name slot, which takes an arbitrary (user defined) string. A purpose slot could also be made optional for all templates, with the user or other domain expert allowed to fill it in. The first example pattern above matches the *output* operation that outputs information structure *database* and occurs below a template that is called "update" by the user. The second example matches an *output* operation whose purpose is "outputting the database".

[11] "Convenience" pattern variables such as "↑", ">", and "←" could easily be introduced. For example, $a \uparrow b$ would be defined to be equivalent to $b \downarrow a$. However, these forms add nothing to the power of the language.

Appendix B. Example Rules

This appendix provides a sampler of the rules in PMB's knowledge base. About one-fourth of the over 200 rules are listed. Each rule is represented by an English paraphrase, rather than by the raw LISP code.

The example rules are listed by kind of knowledge (e.g., consistency checking) rather than format of rule. Within a particular category of knowledge, the rules are listed in the order of the class of the primary template being updated: (1) information structures (AIUs), (2) control structures (ACUs), and (3) primitive operations (POPs). Compound demons have mnemonic names, while other rules have numeric names. If a rule sets up a demon that is also listed in this appendix, the rule description so states.

Recall that each rule is invoked in a context that includes a template name, slot name, and value from the current fragment.

B.1 Completeness by Default and Questioning: Response Rules

Where response rules contain such words as "later", "guarantee", and "eventually", a demon is being created.

B.1.1 Response Rules for Information Structures

RULE057: If the current fragment defines a known AIU prototype to be of type primitive, then store the type in the AIU prototype template in the model and ask if there is a specifier slot, but assume it is a string for now.

RULE085: The current fragment defines the specifier of a primitive AIU prototype. If the specifier is "numeric", then store that fact and ask about a numeric value slot.

RULE093: In the current fragment we have the definition of the type of an AIU prototype. If the type is list, then create a collection, make it ordered, and ask about repetitions.

RULE081: We have the subparts of a plex. First make sure the subparts are in the form of a list structure. Process each subpart, making sure that each subpart name is a unique (among the subparts) literal atom and asking questions about each subpart. Then store the entire list of subparts in the plex template.

B.1.2 Response Rules for Control Structures

RULE005: If the slot value specified in the current fragment is not already the name of a template, then create a new template with that name and with a class of operational unit, ask the type of the new template, and set up two-way linkage with the template that calls this operational unit.

RULE010: If the operational unit is a composite, then store that type and ask about the subparts and orderings, but assume sequential orderings for now.

RULE011: If the operational unit is a test, then store the type and ask about the condition, probability that the condition is true, true action, and false action. Assume no false action.

RULE012: If the operational unit is a case, then store that type and ask for a list of case pairs.

RULE013: If the operational unit is a loop, then store that fact and ask about the initialization, body, and exit pairs. Assume no initialization.

RULE023: We have the exit pairs of a loop. Make sure they are in a list, and then store the list. Make sure each exit pair has exactly two elements, a predicate and a corresponding action. For each exit pair, the predicate should be a Boolean expression occurring in the loop body somewhere and should be unique among the exit predicates for this loop. Put the predicate inside a test and add an *assert_exit_condition* (done by RULE191). The action of the exit pair may be any operational unit.

RULE001: If the name of the program model is legal, initialize the algorithm model and associated demon space, set up the program model template using the name given, and ask for the domain and top level ACU. Assume that the domain is "unknown".

B.1.3 Response Rules for Primitive Operations

Primitive Operations That Return Boolean Values

RULE174: We are expecting a construct (either a primitive operation or an AIU instance) that has a Boolean value. If its type is *is_element*, then store it and ask about the element, collection, and probability slots. Eventually check for consistency between the element and collection slots (done by IS-ELEMENT-CONSISTENCY).

RULE019 (discussed in Section 8.1.1): If the new operational unit is an *is_subset*, then store the type and ask about the subcollection, collection, and probability slots. Also guarantee that subcollection and collection have the same prototypic element (done by IS-SUBSET-CONSISTENCY).

RULE222: If the Boolean expression is a *true_for_all*, then store the type and ask about referents, collection, condition, and probability. Later see if the *true_for_all* can be transformed into an *is_subset* (done by TRUE-FOR-ALL-TO-IS-SUBSET).

RULE228: We have the AIU instance that is used as the referent of a POP with a quantifier, e.g., a *true_for_all*. Add cross-references and guarantee that the referent is only remembered once and never forgotten or modified (done by RULE256 and RULE257). Finally, see if its AIU prototype can be inferred from how it is used, by monitoring the where-referenced slot.

RULE275: We are expecting a Boolean POP. If its type is *has_correspondent*, then store the type and ask for the correspondence, domain element, and probability slots. Eventually guarantee that the domain element is of the same type as the domain AIU of the correspondence slot.

RULE020: If the operational unit is an *is_of_type*, then store the type and ask about the value to be checked, the AIU prototype, and the probability that the test is true.

RULE151: We have the value slot of an *is_of_type*. If the template named as the value slot doesn't exist yet, then create an alternative and ask the appropriate questions about it, create an instance of it, create a *remembered_value* above it, and eventually guarantee that the AIU prototype named in the *is_of_type* occurs somewhere in the alternative tree (done by IS-OF-TYPE-CONSISTENCY).

RULE293: If the Boolean expression is a generic *are_equal* operator, then ask about the two instances and the probability the test is true, and eventually decide how the *are_equal* should be specialized (done by RULE300).

Primitive Operations That Return Non-Boolean Values

The three rules below are discussed in Section 8.1.1.

RULE096: We are expecting a POP or instance that has a collection as its value. If the specified template exists and is a POP, then store two-way linkage and guarantee eventually that the POP returns a collection.

RULE098: We are expecting a POP or instance that has a collection as its value. If the template exists and is an AIU instance, then make sure eventually that it is a collection and insert a *remembered_value* above it.

RULE099: We are expecting a POP or instance that has a collection as its value. If the template doesn't exist yet, create a template of class "collection" and store two-way linkage between it and the calling template.

Primitive Operations for Input/Output

RULE036: The operational unit is an *input* POP, so fill in the appropriate slots and then ask questions about the AIU to be input, the source of the input, optional input format, and prompt and reprompt strings. Assume that the source is the user and assume defaults for the prompt and reprompt. Eventually copy the type-returned slot of this POP from its AIU slot. Eventually construct a default format for the AIU and its sub-AIUs.

RULE148: If the prompt or reprompt is an existing AIU prototype, then eventually make sure it is a string primitive with a value, and store the value as the prompt or reprompt.

RULE146: If the prompt or reprompt is a new template, then create a string primitive, ask its value, and eventually store the value in the prompt or reprompt slot.

RULE262: If the operational unit is an *output*, then store the type and ask questions about its AIU instance, destination, and optional format. Assume that the destination is the user. Eventually create a default format. Eventually see if the *output* can be transformed into an *inform_user* (done by OUTPUT-TO-INFORM-USER).

Primitive Operations That Alter the Flow of Control

The first rule below was used near the beginning of the CLASSIFY example in Chapter 4.

RULE149: We have the condition slot of an *assert_exit_condition*. First make sure it's a legal name. Eventually check every loop that the *assert_exit_condition* is in to see that condition is the name of an exit block in (at least) one of them (done by RULE194). Then store the condition.

RULE238: We have the instance-of slot of a *procedure_instance*. If the procedure named as the slot value doesn't exist, then create it and store cross-references between it and the instance. Eventually store in the *procedure_instance* the type returned by the procedure.

RULE243: We have the list of bindings of a *procedure_instance*. If the list isn't empty, then process each element of the list, store the bindings list in the *procedure_instance*, eventually make sure there are the same number of bindings as parameters in the procedure (done by PARAMETER-#-CONSISTENCY), and eventually guarantee that the types of bindings and parameters agree.

B.2 Completeness by Inference

RULE191: An exit condition of a loop is now known. Create an *assert_exit_condition* with the name of the exit block as its label slot. Create a test with the exit condition as its condition slot and the *assert_exit_condition* as its true action.

INSERT-POP: If we have a POP below the current template, then store the name of the POP in the template.

INSERT-SELECT-ALTERNATIVE: If an *is_of_type* POP is in an exit condition of a loop and the alternative referred to by the *is_of_type* only has two possibilities, then insert a *select_alternative* after the exit condition.

B.3 Completeness by Generating Cross-References

RULE051: We have a new reference to an instance. Update which ACU has this instance in its list of relevant AIU instances. This ACU should be the least global ACU that contains both the current reference to the instance and the current ACU listed in the ACU scope slot of the instance. Also update the ACU scope slot in the instance template to point to this newly computed ACU. Quantified POPs are considered to be ACUs when their referent instances are being handled. There is an error if the new reference to the instance implies that the instance is used both locally and globally to a procedure or quantified POP.

B.4 Consistency Checking

SAME-PRIMITIVE: Check that the two AIUs are primitives with the same specifier subtype (e.g., both strings).

RULE194: Check that the exit condition of a loop is contained within that loop. When the current template knows what operational unit is above it, see if it is the loop we are looking for. If not, move up eventually to the next higher operational unit and repeat the check. There is an error if the program model template is reached.

EXIT-CONDITION-NOT-TEST-CONDITION: Ensure that the exit condition of a loop isn't also the condition of a test (since RULE191 puts the exit condition in a new test).

IS-ELEMENT-CONSISTENCY: In an *is_element* POP, make sure that the element slot is the same type as the prototypic element of the collection slot.

IS-SUBSET-CONSISTENCY (discussed in Section 8.1.3): Make sure that the subcollection and collection slots of an *is_subset* POP have the same prototypic elements.

RULE256: An AIU instance that is used as the quantifier in a POP such as a *true_for_all* has been changed. Make sure that it is still remembered only once.

RULE257: An AIU instance that is used as the quantifier in a POP has been changed. Make sure that it hasn't been forgotten or modified.

IS-OF-TYPE-CONSISTENCY: In an *is_of_type* POP, make sure that the specified AIU prototype is somewhere in the alternative's subtree.

RULE103: We are expecting a template that returns a collection. If the template exists and doesn't return a collection, then there is an error.

PARAMETER-#-CONSISTENCY: Ensure that a *procedure_instance* has the same number of parameters as the procedure declaration.

B.5 Inconsistency Resolution

Both of the following rules deal with the resolution of prototype-instance ambiguity.

RULE267: An AIU instance is referenced. If a template with that name already exists and is an AIU prototype, then create a new instance template for the prototype and insert a *remembered_value* POP between the reference to the instance and the instance itself.

RULE276: An AIU prototype is referenced. However, the template referenced is currently an AIU instance that doesn't have a prototype defined. Change the template into an AIU prototype and ask questions about its type. Create a new instance template, make it the primary instance of the prototype, copy information from the old instance to the new instance, update all pointers to the old instance to point to the new one, and make all demons that were active in the old instance active in the new instance instead.

B.6 Specialization of Generic Operators

RULE300: If the first argument of an *are_equal* generic operator is an alternative, then the second argument should be a subtype of the alternative. Change the *are_equal* into an *is_of_type*. Finally, try to insert a *select_alternative* in the appropriate place after the *is_of_type*.

B.7 Canonization

TRUE-FOR-ALL-TO-IS-SUBSET: The current template is a *true_for_all* test. If it has the form (*true_for_all x*) ((*x is_element a*) *implies* (*x is_element b*)), for arbitrary expressions *a* and *b*, then transform the entire *true_for_all* expression into (*a is_subset b*).

OUTPUT-TO-INFORM-USER (discussed in Section 8.1.3): If the *output* POP is merely outputting a string constant to the user, then transform it into an *inform_user*.