INCREMENTAL, INFORMAL PROGRAM ACQUISITION[1]

Brian P. McCune

Advanced Information & Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, California 94040

Abstract. Program acquisition is the transformation of a program specification into an executable, but not necessarily efficient, program that meets the given specification. This paper presents a solution to one aspect of the program acquisition problem, the incremental construction of program models from informal descriptions [1], in the form of a framework 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.

1. Introduction

The paper 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. 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.

2. The Problem

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

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. The user thus has the most flexibility to provide new knowledge about any part of the program at any time. For example, a single fragment conveying a small number of pieces of information is the statement "A is a collection." This identifies an information structure called A and defines it as a collection of objects. However, the fragment says nothing about the type of objects, their number, etc. These details are provided in program fragments occurring *before* or *after* this one.

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. An incomplete program model part may be completed either by use of a default value, by inference by PMB, or from later fragments from the user.

[1] This paper describes research done at the Stanford Artificial Intelligence Laboratory and Systems Control, Inc., and supported by DARPA under Contracts MDA903-76-C-0206 and N00014-79-C-0127, monitored by ONR. Fellowship support was provided by NSF, IBM, and the De Karman Trust. The views and conclusions contained in this paper are those of the author.

Program model consistency is monitored at all times. PMB tries to resolve inconsistencies first; otherwise, it reports them to the user. For example, the membership test fragment $x \in A$ requires that either A have elements of the same type as x (whenever the types of A and x finally become known) or their types inferred to be the same.

Because a fragment may possess ambiguities, its interpretation depends upon the model context. So PMB specializes a generic operator into the appropriate primitive operation, based upon the information structure used. For example, *part_of*(x,A) (a Boolean operation that checks if information structure x is somehow contained within A) becomes $x \in A$, if A is a collection with elements of the same type as x, and an *is_component* if A is a plex (record structure).

PMB is capable of *canonization*, the transformation of equivalent information and procedural structures into concise, high level, canonical forms. 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.

3. Control Structure

The model building problem is to acquire knowledge in the form of a program model. The control structure of PMB is based upon the "recognition" paradigm [2], in which a system watches for new information, recognizes the information based upon knowledge of the domain and the current situation, and then integrates the new knowledge into its knowledge base. PMB has one key feature: 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 the input of new fragments or changes in the partial 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. 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 model has been filled in by response rules or perhaps other demons.

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. 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 modelling language's real power comes from its very high level operators for information structures such as sets, lists, mappings, records, and alternatives of these.

Below are English paraphrases of three rules that exemplify the major types of rules used in PMB. Rule 1 is a response rule for processing a new loop. Rule 2 is a demon that checks that the arguments of an *is_subset* operation are consistent. Rule 3 is a canonization demon that transforms a case into a test when appropriate.

[1] 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.

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

[3] If the statement is a case, the case has two condition/action pairs, and the first condition is the negation of the second condition, then change the case into a test.

Both response rules and simple demons are procedural. Compound demons (i.e., those whose antecedents test more than one object in the model) use declarative antecedent patterns that are expanded automatically into procedural form by a rule "compiler".

5. Example of PMB in Operation

The model building excerpt below displays (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 discusses three program fragments generated from an English dialog. Each fragment is followed by a description of how it was processed by PMB, a snapshot of the partial model at that point, and a list of the outstanding demons. A detailed trace for the first fragment shows PMB focusing on individual slots of a fragment, creating model templates, and creating subgoals. The other fragments emphasize the creation and triggering of demons.

Names preceding colons are unique template names that allow fragments to refer to different parts of the model. Missing parts of the partial model are denoted by "???". Newly added or changed lines are denoted by the character "|" at the right margin.

[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 exist.]

```
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 an optional partial ordering on the execution of its subparts. The response rule that processes the **composite** creates the following two subgoals, along with response rules to handle them (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 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 third fragment, which defines "input_concept" to be an *input* primitive operation, is omitted. Information structures from this fragment are not shown in the models below.

The fourth fragment defines the second step of the **composite**. This fragment might have come from "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". Since PMB doesn't know precisely where the test of "exit_condition" will be located, 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 attaches Demon 1 to the template for "exit_condition" to await this event. 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;
    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 fifth fragment defines the body of the loop, thus triggering the two demons set up previously. A possible 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
```

["Loop_body" is a composite with four named steps. 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, and succeeds. 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;
    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 of 32 fragments have been processed.]

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 model into an efficient program without human intervention. The model is acquired during the acquisition phase; the model is coded only when it is complete and consistent.

PMB may work 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. However, the operation of PMB is not predicated on the existence of other modules: all fragments to PMB could be produced by a straightforward deterministic parser for a surface language such as the one used to express fragments.

7. Conclusion

PMB has been used both as a module of the PSI program synthesis system [3] 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.

The most important topics for future work in this area include (1) extending and revising the knowledge base, (2) providing an efficient mechanism for testing alternate hypotheses and allowing program modification, and (3) providing a general mechanism for specifying where in the program model a program fragment is to go. The last problem has resulted in a proposed *program reference language* [1].

8. References

[1] Brian P. McCune, *Building Program Models Incrementally from Informal Descriptions*, Ph.D. thesis, AIM-333, STAN-CS-79-772, AI Lab., CS Dept., Stanford Univ., Stanford, CA, Oct. 1979.

[2] Daniel G. Bobrow and Terry Winograd, "An Overview of KRL, a Knowledge Representation Language", *Cognitive Science*, Vol. 1, No. 1, Jan. 1977, pp. 3-46.

[3] 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*, Vol. 1, CS Dept., Stanford Univ., Stanford, CA, Aug. 1979, pp. 342-344.