PM: A System to Support the Automatic Acquisition of Programming Knowledge

ROBERT G. REYNOLDS, MEMBER, IEEE, JONATHAN I. MALETIC, AND STEPHEN E. PORVIN, MEMBER, IEEE

Abstract—**The Partial Metrics (PM) system described in this paper utilizes chunking as a model for acquiring knowledge about program implementation. The chunking paradigm has three phases. The first phase partitions the object to be chunked into relatively independent parts called aggregates. The objects to be chunked in PM are code modules. Modules are separated into a collection of aggregates based on a model of stepwise refinement. A heuristic is given that generates a hierarchically structured collection of refinement steps that describes how the program could have been developed as a set of independent refinement decisions (object-oriented stepwise implementation). The second phase encodes (abstracts) each of the aggregates. Various techniques for symbolic learning can be applied to produce a frame-based encoding of information present in the code. This abstraction contains information about the aggregate's role in the refinement process as well as the code's functionality. The third phase inserts the chunked aggregate into a hierarhically structured library of cases based on the contents of its frame description. The storage of an aggregate enables its future use in problem solving activities. The paper describes an example of how this approach can be used to acquire knowledge from a sort module.**

Index Terms—**Automatic programming, chunking, knowledge acquisition, object-oriented stepwise refinement, reverse engineering, software quality metrics, software reuse.**

I. INTRODUCTION

HERBERT SIMON suggests that one of the major bottlenecks to the generation of automatic programming systems is the acquisition of programming knowledge for use in such systems [1]. For example, a number of researchers have demonstrated the importance of planning knowledge in the programming process [2]–[5]. Several automatic programming systems employ plan-like knowledge [6]–[8]. The library of plans associated with a given knowledge-based system is a major factor influencing that system's performance. However, acquisition of the planning knowledge that goes into such a library can involve an extensive amount of handcrafting on the part of the engineer. Such handcrafting is time consuming and impacts plan-based systems in three fundamental ways. First, it can narrow the scope of domain applications. Second, it can limit the number of applications within a given domain. Third, it can limit the number of target languages supported by the planning system.

Manuscript received September 30, 1989; revised April 8, 1990.

The authors are with the Department of Computer Science, Wayne State University, Detroit, MI 48202.

IEEE Log Number 9036985.

The goal of the Partial Metrics (PM) project is to investigate the development of hybrid learning environments which support the acquisition of such knowledge. It is felt that a suitable environment must support the following properties:

1) Storage and retrieval of heterogeneous knowledge and structures. Programming knowledge comes in many forms and the system should support the acquisition and utilization of each. For example, a professional programmer must possess knowledge of algorithms and data structures, knowledge about an application domain, and knowledge of how to plan and implement application software.

2) Task neutral knowledge structures. The knowledge organization does not bias its use toward only one type of problem solving activity or another. This property allows stored knowledge to be accessible to all phases of the software design process.

3) Layered learning strategy. This strategy allows knowledge to be acquired at complementary levels of abstraction. The system should be able to "remember" details of specific events and produce reasonable high-level generalizations about each event. There also must be mechanisms to support translation of acquired knowledge from one level of abstraction to another.

4) Knowledge acquisition at different granularity levels in programs. Stored knowledge may correspond to one line of code, several lines, or perhaps the entire module.

5) Knowledge structuring consistent with skilled programmer behavior. This property implies that the knowledge structure's design must be motivated by empirical studies whenever possible.

As motivated by the above properties, research on the PM project has taken the following directions. First, the prototype system is Case-Based. Each piece of programming knowledge is annotated with a description of the situation for which its use is appropriate. The piece of knowledge is stored in the network of cases based upon its annotation. The annotation is expressed using a frame structure with slots representing information relative to its use in the stepwise refinement process as well as its function in a program. The frame is therefore capable of characterizing the class of events in which its associated piece of knowledge can participate. Each category of knowledge object has its own frame template. For instance, the template for plans differs from those for code pieces.

Three principal knowledge categories have been defined so far; plans, fundamental implementation knowledge, and application implementation knowledge. Each category possesses its own hierarchical network of stored cases with the most generic events indexed at the top and the most specific at the bottom.

Current work on the project focuses on the acquisition of fundamental implementation knowledge. The project's immediate goal is to construct a system capable of organizing coding knowledge acquired from examining programs in introductory computer science texts. This paper demonstrates how the PM system acquires knowledge from the kind of code typically found in those texts. The particular example used is a sorting routine from Gilbert [9].

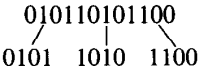
The framework for the knowledge acquisition process described is based on an empirical model of the stepwise refinement process for implementing code modules. This model describes stepwise refinement in terms of specifically developed metrics, *partial metrics* [10] and *refinement metrics* [11], which, respectively, assess the complexity of each refinement step in terms of a programming language's syntax and semantics. Use of the metrics has suggested a method for breaking down completed code modules into a collection of refinement objects based on a canonical model of the process by which they were implemented. This collection is structured into a hierarchy of aggregates (HAG). Each aggregate in the hierarchy is a potential object for acquisition into a case library; each can be viewed as a chunk of knowledge to be described and stored.

Section II presents a general model of the chunking process and describes how the PM system supports this paradigm. Section III presents a heuristic to generate the hierarchy of aggregates and applies the heuristic to the sort example. The HAG produced is a description of how the sort module can be implemented using refinement steps that satisfy a given set of metric constraints. Section IV discusses how the aggregates can be integrated into a hierarchically structured case library. As a result, the system is able to acquire and reuse programming knowledge from code modules.

II. The Role of Chunking in PM

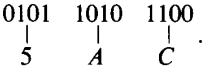
The need for humans to associate specific input with more general or abstract patterns was motivated by the work of Miller, a psychologist, on the size of a human's short-term memory [12]. His results suggest an average person has a limit of seven plus or minus two items. This limitation can be overcome when each of the objects in short-term memory is allowed to represent an encoding of a collection of more detailed objects. For example, few people would be able to remember the string of bits 010110101100. However, if the string is linearly decomposed into three 4-bit subsequences, (0101, 1010, and 1100), then each 4-bit sequence can be encoded in its hexadecimal equivalent. The resultant set of objects, 5 *AC*, is easy to remember. This result can then be decoded into the original, more detailed, representation if necessary.

The above example suggests that chunking consists of several distinct phases. The first phase decomposes input into aggregates of low-level objects. These aggregates should be relatively independent in order to simplify the encoding process. Simon observed [13] that many complex systems can be decomposed into a collection of hierarchically structured subsystems, such that the intrasystem interactions are relatively independent of intersystem ones. The decomposition of the binary sequence above can be expressed hierarchically as



If there is no dependence between the inter- and intrasubsystems, then the system is said to be *completely decomposable*, as is the case for the example above. *Nearly decomposable systems* exhibit a few interactions, but still permit efficient encoding to take place [13].

The second phase of the chunking process is encoding or abstraction. If the aggregates are independent then the encoding process is particularly straightforward since the abstraction mechanism can be applied independently to each. The presence of some interaction between components may cause the propagation of symbolic constraints produced by the chunking of one aggregate to others. In a nearly decomposable system this propagtion should be relatively local in nature. The encoding of the aggregates above can be expressed as



The third phase involves the storage of the chunked aggregates in a knowledge structure that enables its future use in a problem solving activity. It is assumed that the collection of chunked aggregates can be described in a hierarchical manner.

The chunking process described above has been employed in a variety of AI applications: for example, a hierarchical planning system for chess [14], a system for learning about problem solving procedures [15], and as part of a general model of human practice [16].

A. What is Being Chunked in PM?

Input to the PM system is target code from a specific programming language. The current prototype supports Ada, Pascal, C, OPS5, and Prolog. The prototype requires that target code have the following properties.

1) The code is well structured and maintainable. Esteva's Software Reusability Classification System [17] is used to determine this property for a candidate code module. His system utilizes inductive learning to generate a decision tree from examples of reusable and nonresuable code in terms of a set of software quality metrics.

2) The target code has been documented, well tested, and found to be reliable. Since the plan generated from

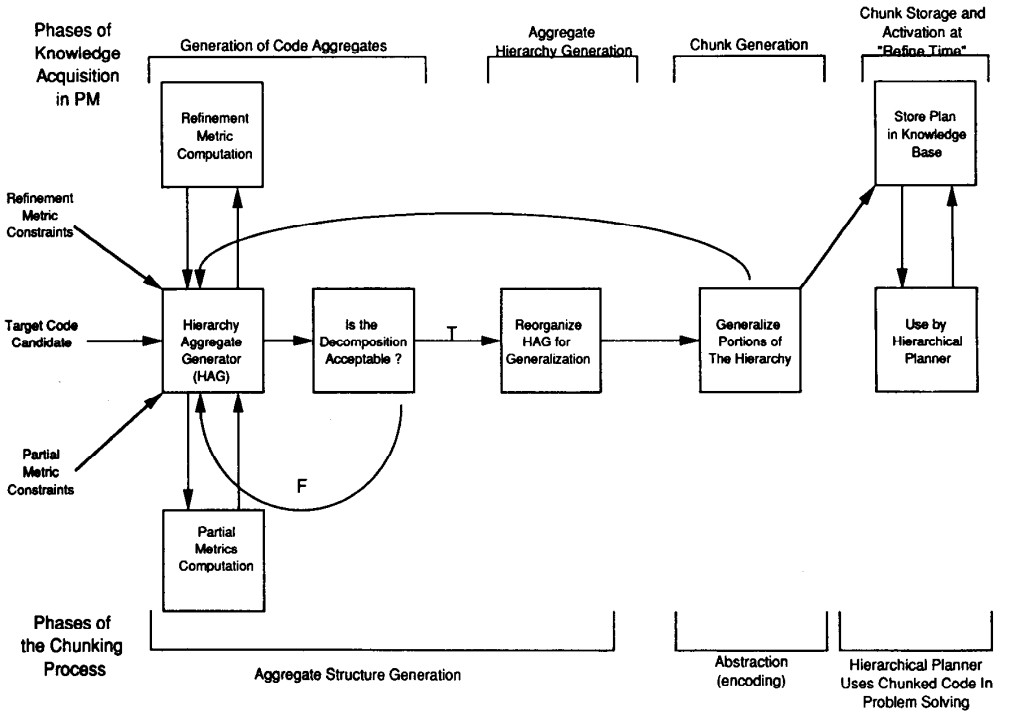


Fig. 1. Relationship between phases of knowledge acquisition in PM and chunking.

the code will be used for many other designs, this property prevents the propagation of errors to other applications that use the plan.

B. How the Chunking Process is Integrated into the Structure of PM

Fig. 1 relates the basic phases of the plan knowledge acquisition process in PM to the generic chunking paradigm. The initial phase attempts to partition the code into a hierarchy of code aggregates that are syntactically independent. The model used to generate this hierarchy is an attempt to describe the construction of the module as a set of relatively independent refinement steps.

The aggregation procedure is driven by a set of metric parameters which are input with the target code to be processed. These metrics, which are discussed in more detail in Section III, were selected because they successfully describe patterns in the structure of code segments added to pseudocode programs during code refinement [18], [19].

The relative independence of the set of aggregates produced is then assessed in metric terms. If the aggregates do not exhibit enough structural independence then the aggregation process can be repeated by adjusting the values of the metric parameters to produce a more appropriate grain size for the aggregates. The problem of finding a combination of metric values that produces aggregates with acceptable levels of independence is an example of the "granularity problem" described by Roy and Mostow [20].

The advantages of dealing with the granularity problem prior to the generalization process are twofold. First, it reduces the effort involved in generalization since fewer interactions between aggregates need to be handled. Second, it results in the generation of plans that are efficient to use. Tambe and Newell have shown that some chunks are more expensive to employ in the problem solving process than others [21]. One of the causes of this expense is the presence of a large number of condition elements in the chunk. By using metrics to force the relative independence of aggregates, the number of such conditions, and therefore the cost of using the aggregates, should be reduced. The following sections demonstrate how knowledge about a target program is acquired by the PM system.

III. The Generation of a Hierarchy of Aggregates

The principal AI paradigm used in this phase is search. Simon suggests that implementation of target code via stepwise refinement of pseudocode can be naturally expressed in terms of state space search [1]. The state space to be searched is the set of pseudocode programs in a target language. Given an initial abstract description of a desired program, a programmer searches for a sequence of refinement steps that will produce suitable target code. Fig. 2 presents a tree, termed a *refinement tree* by Barstow [7], which represents the possible paths from an initial pseudocode description to a final code module. Each

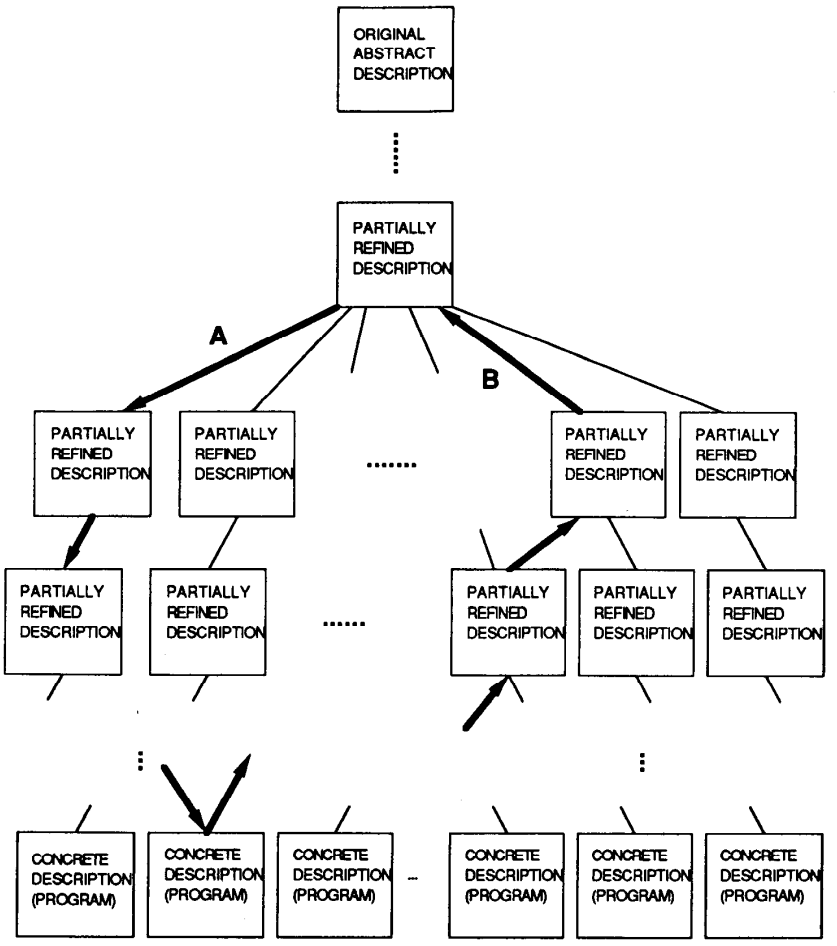


Fig. 2. Refinement tree representation of the search space for pseudocode refinement. Path *A* corresponds to the refinement steps used to generate the target code by the programmer. Path *B* corresponds to the reverse path generated by the PM system in producing the hierarchy of aggregates.

node in the tree corresponds to a pseudocode program. An arc from a node x to a node y represents the fact that y was produced, via refinement, from x.

For any given program implementation, each refinement in the programming sequence can be recorded to give the actual path taken through the tree by the programmer. Fig. 2 highlights the specific path (labeled *A*) taken by a programmer. Studies of numerous refinement sequences in the software engineering literature suggest a tendency of programmers to place constraints on the path taken through the tree in terms of the code complexity added at each step [10].

A set of metrics developed by Reynolds ([10], [11]) characterizes these constraints on the search process. These metrics are designed to assess the structural complexity of the code aggregates added during the refinement step. Several basic assumptions were made in the construction of these measures.

1) Each refinement step can be viewed as an aggregation of a number of constituent code generation decisions. A portion of each decision deals with the expression of the results in terms of a programmer's internalized model of the target language's syntax. Therefore, the syntactic structure of the target language's grammar affects the aggregation decisions made. This perspective has been suggested by others such as MacLennan [22]. The metrics selected to represent the impact of the language's grammar on the refinement process are called *refinement metrics* [11], [23].

2) The code aggregate added at each refinement step is also influenced by limitations on the programmer's short-term memory. It is assumed that these limitations can be estimated by counting the lexical symbols added to the code at each refinement step. The metrics developed to assess this aspect of the code aggregate are called *partial metrics* [10].

3) Both partial and refinement metrics should codify information used by a programmer in selecting a path of

refinement through the tree that satisfies certain limitations on the code complexity added at each step. In doing so, the metrics must be capable of predicting the complexity of the completed code that would be achieved if the programmer were to follow the current path to completion. If the prediction exceeds some acceptable limit the search systematically shifts to some other candidate path. It is important to ensure that the prediction consistently underestimates the possible value; otherwise, the fruitful path may be missed [24]. Both refinement and partial metrics have been designed to allow the prediction of properties along an entire path. In addition, these estimates are designed to consistently underpredict the complexity of the completed code associated with the current path.

The goal of the aggregation process is to traverse the refinement tree in reverse by systematically removing an aggregate of code from the target program and replacing it with a stub. The aggregate that is removed satisfies the set of metric constraints provided as input. This process continues to construct a path through the refinement tree using a simple breadth-first search from bottom to top. The search is guided by an evaluation function composed of one metric from each of the set of partial and refinement metrics. The result of the process is the production of a hierarchically nested sequence of code aggregates where each aggregate satisfies given metric constraints. Note that this path does not have to correspond to that used to construct the program in the first place. Fig. 2 reflects this distinction: the program is shown to be produced via one path *A* through the tree, and decomposition occurs in reverse following another path *B*.

A. The Metrics Used for Aggregation

Since the goal is to generate a reverse path through the refinement tree, it is desirable that the aggregates produced during the traversal should be considered in the same way as if they had been produced by an expert. If this is the case, the aggregate structure produced is more likely to be understandable to a human observer. In order to understand how each metric is able to characterize a particular aspect of the code aggregate added, the basic effects of a refinement decision on the code must be described.

The external effects of a refinement decision can be manifest in two basic ways:

1) a change in the systematic structure of the pseudocode program

2) a change in the implementation subgoals associated with the pseudocode program.

In order to measure the effect of a refinement decision it was necessary to produce measures that quantitatively assess the changes of each effect. Partial metrics were developed as a means to describe changes in a pseudocode program's structural complexity [25]. A partial metric is computed relative to a specific model of a pseudocode or partial program. In terms of this model, each program consists of a *projected* part and a *prescribed* part. The prescribed part corresponds to reserved words and symbols in the target language. The projected part corresponds to implementation tasks that remain to be carried out by the programmer. These tasks are represented in the code by stubs. The position of each stub in the code implicitly determines the syntactic class associated with that stub. The implied syntactic class is a characterization of the implementation task in syntactic terms.

Software engineers have developed a variety of metrics ([26]–[28]) which assess the structural complexity of completed code, programs that have only a prescribed component. A partial metric is one that is able to compute the contribution of both the prescribed and the projected part in order to produce an overall estimate of structural complexity. Reynolds [10], [25] demonstrated that a number of standard metrics, such as McCabe's, McClure's, and Halstead's, can be extended to measure the complexity of pseudocode programs. These extended metrics are especially good predictors of complexity when the code is less than 75% prescribed.

The specific partial metric used as the aggregation heuristic is *program volume*. Program volume is a prediction of the number of bits required to store the corresponding code segment in computer memory. Both prescribed and projected terms contribute to this value. The contribution of the projected terms is set to be the minimum contribution to the count of operands and operators for a nontrivial substitution of code for the projected term. As a result this prediction is a consistent underestimate of the actual value. The relationships necessary to compute program volume are given below:

n(program) =
number of unique operands in prescribed code
+ number of unique operators in prescribed code
+ number of unique projected terms

N(program) =
total number of operands in prescribed code
+ total number of operators in prescribed code
+ total number of operands associated with the projected terms
+ total number of operators associated with the projected terms

Volume(program) =
N(program) * $\log_2 n$(program).

Partial program volume reflects certain limits in the amount of pseudocode added at each refinement step [10]. Results from a study of numerous refinement examples taken from the software engineering literature suggested that the addition of a new pseudocode refinement added on the average 122 bits to the partial program volume with a standard deviation of 33. The example aggregates in this paper were produced by setting the partial program volume to the average 122 bits.

A second class of metrics was developed to measure the change in subgoal complexity as the result of a refinement. These metrics are termed refinement metrics and

measure language support for refinement decisions [11]. In particular, they are used to estimate the decision-making effort associated with the transformation of a program stub into completed code.

Associated with each term is its identifier class which corresponds to a particular class of nonterminals in the grammar for the target language. It is also possible to associate an index of syntactic complexity with each identifier class. This index is helpful in isolating trends in the selection process for projected terms.

The index used corresponds to the number of productions in the target language's grammar that are needed to express that identifier class in terms of the language's terminal symbols. The index, refinement depth, is defined as the maximum number of substitutions, without cycles, that are needed to replace a nonterminal on the right-hand side of a grammar's production rule that contains the identifier class on the left-hand side. If the nonterminal is on the left-hand side of more than one production then the maximum over the set of relevant productions is taken. The index is called refinement depth since it reflects the number of productions required to express the most complex subtask associated with the implementation of a stub of a given nonterminal class.

Refinement depth has been shown to be an important factor in predicting the structure of the code aggregate coded to a pseudocode program [18]. In particular, there appears to be a limitation on the extent to which a stub of a given nonterminal will be decomposed into subtasks. That is, the distance between the depth for the parent and each of its children seldom exceeds a certain value. This value is always much less than the maximum difference between nonterminal classes in the language. Constraints on the difference between the parent stub and its children reflect a gradualism in the problem solving process.

B. Generating the Hierarchy of Aggregates

The basic approach used in generating the HAG is to conduct a branch and bound search through possible paths from the target code to a root node. The search is constrained by the input values for refinement depth and partial program volume. Fig. 3 presents the result of partitioning a sort module, written in Ada and taken from Gilbert [9], into aggregates using a refinement depth of 23 and a partial program volume of 122. Each bracketed portion of code is indexed by the order in which it was produced.

The aggregation process is performed on the token stream generated from the target code. That is, the characters in the source code are grouped into collections that correspond to reserved words, symbolic names, and other code components. The token stream is a sequence of these tokens produced by scanning the code from top to bottom. Every symbolic name encountered in the scan, such as **no_items_interchanged,** is associated with a nonterminal class and its associated refinement depth. Initially all symbolic names in the token stream for the target code are associated with the *identifier* class.

```
if size ≤ 0 then                                              8
  put( error message );                                       7
else
  if size > 1 then                                            6
    loop
      no_items_interchanged := true;                          5
      for i in 1 .. size-1 loop                               4
        if table(i) > table(i+1)                              3 then
          temp        := table(i);                                2
          table(i)    := table(i+1);
          table(i+1)  := temp;
          no_items_interchanged := false;
        end if;
      end loop;
      exit when no_items_interchanged;                        1
    end loop;
  end if;
end if;
```

Fig. 3. Aggregation of Gilbert's sort example with partial volume of 122 and refinement depth of 23.

Once the token stream is produced the following aggregation procedure is executed:

REPEAT

1. Find the symbolic name associated with the lowest syntactic class. If there is more than one, select the one closest to the end of the token stream. This is the current "seed" for the aggregate.
2. WHILE
   [the partial program volume for the current aggregate is less than the given upper bound]
   AND
   [(the maximum refinement depth for the bracketed syntactic structures minus the minimum refinement depth for the bracketed syntactic structures) is less than or equal to the given upper bound for refinement depth]
   DO
   a. Adjust the top "bracket" by moving the scan to include the preceding token in the stream.
   b. IF [the token is a reserved word that requires another to be syntactically correct (such as **begin**) and the other term is currently not in the aggregate (e.g., **end**)]
      THEN
      [Adjust the bottom bracket downwards to include the additional term.]

   END WHILE
3. Remove the bracketed code segment and replace it with a new symbolic name. The syntactic class of that name is the nonterminal of highest refinement depth associated with a syntactic structure in the segment. This name corresponds to a stub representing the segment that was removed.

UNTIL (top of token stream reached).

The process can be illustrated using the example code. First, **no_items_interchanged** is determined to be the seed since it is the closest symbolic name to the end of the program. Next the scan moves the top bracket up to **when** without encountering a problem. The discovery of

when activates the extension of the bottom bracket to pick up the rest of the **when** action. At this point, the difference between the refinement depth values for the largest (*exit-statement*) and smallest (*identifier*) syntactic structures exceeds the refinement depth bound used of 23. As a result the bracketing process is terminated for the aggregate. The bracketed code is removed and replaced by a unique stub name associated with the aggregate. The process then continues until all aggregates are produced.

C. Preparing the Hierarchy of Aggregates for Chunking

Each aggregate, excluding the root, is referenced by a stub in another aggregate. These relationships are expressed by the tree in Fig. 4. Each aggregate corresponds to a node in the tree and is labeled by a unique stub name based on the general syntactic class for that code. The integer in the upper right hand corner represents the order in which the aggregate was generated. Our approach produces all the leaf nodes prior to interior nodes.

The number in the lower right hand corner indicates the aggregate's independence from other aggregates. This independence is computed in the following fashion. For each symbolic name in the aggregate's symbol table, divide the number of occurrences in the aggregate by the total number of occurrences in the program. The greater this ratio, the more localized the term's use is in relation to that aggregate. The average of these ratios over all symbolic names in the aggregate is an indication of syntactic independence. Note that the values tend to increase from the leaf aggregates to the root as a natural result of the fact that higher level modules contain more stubs, and all of these stubs are unique occurrences.

The success of the partitioning process described earlier in producing aggregates that exhibit sufficient syntactic independence to facilitate chunking is measured in terms of these values. Since the chunking process described in the next section typically begins at the leaf aggregates and propagates upward, it is important that the leaf aggregates possess sufficient syntactic independence to allow the process to begin. The rule of thumb used is that a leaf aggregate should have a ratio in excess of 0.75 in order to be a viable candidate for chunking. If this is not the case, an attempt is made to cluster leaf nodes (and their immediate parents) based on the extent to which the same variables are used in each. If the clustered nodes have a ratio less than 0.75 other leaf nodes are added until the ratio exceeds 0.75. In our example, the clustering combines aggregates 1 through 5. Since this new aggregate's ratio exceeds our lower bound of 0.75 it will be used in place of the 5 aggregates.

As a result of the clustering, each of the aggregates represents a relatively independent refinement step. This corresponds to the goal of making each refinement step an independent object. The canonical refinement sequence produced is referred to as an object-oriented refinement sequence. Note that the aggregation process concerned only the body of the code. This reflects another aspect of our canonical refinement sequence: variable declarations

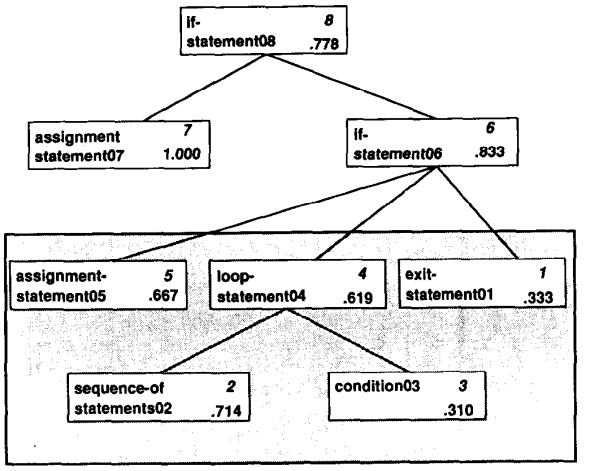


Fig. 4. Hierarchy of aggregates (HAG) for the sort example.

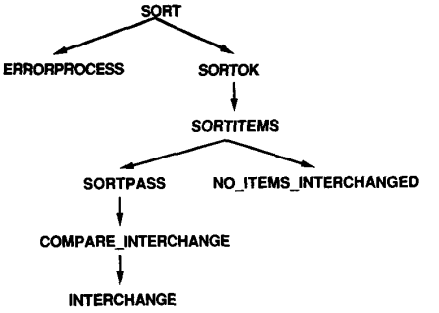


Fig. 5. Gilbert's actual refinement sequence.

are deferred until they are first used in the substituted code. Each refinement object is thus implicitly associated with the pertinent declaration. This opportunistic declaration of variables reinforces the view of refinement as a series of independent substitutions of refinement objects.

The canonical sequence of refinement objects produced will not necessarily correspond to the actual sequence used by a programmer in generating the code. However, if the programmer did use an object-oriented approach the PM generated sequence can resemble the original. For example, Fig. 5 displays the actual refinement sequence generated in Gilbert's sort routine. The labels for each node correspond to the name of the stub in the process. Notice that the tree structure closely resembles that produced by the PM system (see Fig. 4).

IV. Chunking the Hierarchy of Aggregates

In order to understand how the knowledge embodied in these aggregates is encoded and stored, one must consider the overall organization of knowledge in PM. Kolodner and Riesbeck [29] and Williams [30] among others have suggested a case-based approach to acquiring software engineering knowledge. In PM, each aggregate is considered a potential case. The chunking process concerns the production of an abstract description for each code aggre-

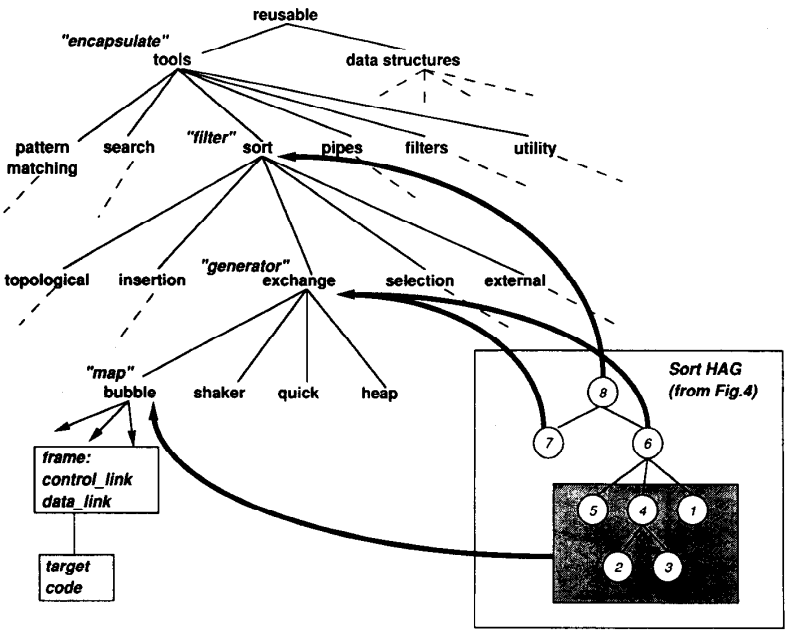


Fig. 6. Placement of the sort example aggregates into the hierarchically structured case framework.

gate. The description is made in terms of its role in both the refinement process and computation process. This description is then used to index the case into a hierarchically organized case-base.

Since PM's task is to acquire elementary programming knowledge, the overall organization of cases is chosen to reflect the way knowledge might be organized in a basic programming text. Specifically, PM uses a structure developed by Booch in his text on algorithms and data structures in Ada [31]. This organization was selected because it supports an object-oriented approach and because its target language, Ada, is supported by the PM prototype.

Fig. 6 presents a "slice" of Booch's hierarchical structure along with links to the sort example's HAG. The slice highlights that part of the tree which is relevant to the insertion of knowledge present in the sort example. Dotted lines represent extensions that are not needed to describe the acquisition of aggregates for the sort program.

Aggregates are stored in the structure based on their role in stepwise refinement and their computational role in performing a given function. Each of these aggregate's roles needs to be specified before the storage process can commence. The resultant description is represented as a frame with two parts: one part describes the role the aggregate played in the canonical refinement sequence from which it was taken; the other part describes its computational functionality.

The role played by an aggregate in the refinement process is expressed in terms of the data and control links between it and the other aggregates. There are two categories of links: explicit links and implicit links. In the stepwise refinement process, an aggregate replaces a stub in the pseudocode program. That stub had certain implied control and data links between it and the prescribed code in which it was embedded. The prescribed portion of the replacement aggregate makes some of these dependencies explicit in terms of target code. These explicitly instantiated control and data dependencies are said to be *activated* by the addition of the aggregates. The projected portion of the aggregate corresponds to implied dependencies that need to be resolved in future refinement steps. For example, the addition of code aggregate 7 into the current pseudocode provided by aggregate 8 makes the response of the program to a failed test explicit. Thus, aggregate 7 is explicitly tied to a failed branch.

The functionality of an aggregate is described in terms of generalizations about its component segments. Based on its code structure each aggregate is categorized as a subtype of three basic classes of generalizations: filters, generators, and maps [31]. If an aggregate's contents are composed of more than one generalization class the aggregate is given the category of the component class with highest priority. Maps are of highest priority followed by generators and filters. In the sort example, aggregate 8 is a filter while its child aggregates, 6 and 7, are generators. In fact, aggregate 6 performs both a filter and generator function and is thus classified as a generator.

The generalization activity which produces information about an aggregate's refinement and computation rules can be supported by a variety of symbolic learning approaches. Several different machine learning techniques have been investigated as vehicles for producing components of the frame descriptions. These include version spaces, genetic algorithms [32], among others.

The result of the generalization process is the production of a frame description for each aggregate. The aggregates are then added to the hierarchy (see Fig. 6) in order of their functionality. Aggregates classified as maps are added first because the insertion process is bottom up and map functions are generally found in leaf nodes. Each node in the hierarchy is labeled by a frame which describes the aggregate type that can be stored there. The frame structure, associated with a node, has the same format to the ones generated for each aggregate. The aggregate is inserted in the collection association with the frame that most closely resembles it. In Fig. 6 the map aggregate is attached to the *bubble sort* "map" node.

Other aggregates are added to the structure relative to the placement of the first aggregate. Since the generation invoked the map that defines the bubble sort activity, we move up the hierarchy along a path from that leaf node. Thus, the remaining aggregates are inserted at higher positions in the hierarchy in reverse order of their refinement. In the sort example, aggregate 6 is added to the collection of cases associated with the *exchange node.* A piece of the frame for that node is used to label it here. That piece states that aggregates stored there must be "generators." Likewise, to store the filter aggregate, we would move up the tree to the next node, *sort.*

The result of this process is the storage of aggregates in positions that reflect both their position in the refinement sequence and their functional differences. The insertion of knowledge in the form of aggregates is done bottom up; the use of these aggregates to produce desired code could take place in a top down fashion.

V. Conclusions

The chunking process discussed in this paper is based in large part on empirical observations of programmers. Thus, the actual distribution of knowledge in the tree structure produced by PM may be used to assess the ability of students to assimilate programming knowledge in terms of a particular set of programming examples. For example, if programs are not well-structured it may not be possible to successfully decompose them into independent aggregates. Therefore, the program, if it is to be stored at all, will be attached to a leaf node. If all the examples are of this type most of the information will be at the tree's bottom level. Given that the above situation represents a worst case scenario one expects a set of well-structured examples designed using an object-oriented refinement methodology will produce a more uniform distribution of information throughout the structure. These kinds of hypotheses are to be investigated in the future with this system.

References

[1] H. A. Simon, "Whether software engineering needs to be artificially intelligent," *IEEE Trans. Software Eng.*, vol. SE-12, no. 7, pp. 726–732, July 1986.

[2] E. Soloway and F. Detienne, "An empirically-derived control structure for the process of program understanding," Institut National de Recherche en Informatique et en Automatique Domaine de Voluerau Rocquencourt, Rapports de Recherche No. 886, B.P. 105, 78153 Le Chesnay Cedex, France, Aug. 1988.

[3] S. Letovsky, "Cognitive processes in program comprehension," in *Proc. First Workshop Empirical Studies Programmers*, E. Soloway and S. Iyengar, Eds. Ablex, 1986, pp. 58-79.

[4] R. S. Rist, "Plans in programming: Definition, demonstration, and development," in *Proc. First Workshop Empirical Studies Programmers*, E. Soloway and S. Iyengar, Eds. Ablex, 1986, pp. 28 47.

[5] E. Kant and D. R. Barstow, "The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis," *IEEE Trans. Software Eng.*, vol. SE-7, no. 5, pp. 458-471, Sept. 1981.

[6] C. Rich and R. C. Waters, "The programmer's apprentice: A research overview," *IEEE Comput. Mag.*, vol. 21, no. 11, pp. 10-25, Nov. 1988.

[7] D. R. Barstow, *Knowledge-based Program Construction.* New York: Elsevier-North Holland, 1979.

[8] W. L. Johnson and E. Soloway, "PROUST: An automatic debugger for Pascal programs," in *Artificial Intelligence and Instruction: Applications and Methods*, G. Kearsley, Ed. Reading, MA: Addison-Wesley, 1987, pp. 49-67.

[9] P. Gilbert, *Software Design and Development.* Chicago, IL: Science Research Associates Press, 1983.

[10] R. G. Reynolds, "The partial metrics system: Modeling the stepwise refinement process using partial metrics," *Commun. ACM*, vol. 30, no. 11, pp. 956-963, Nov. 1987.

[11] R. G. Reynolds and J. I. Maletic, "An introduction to refinement metrics: Assessing a programming language's support of the stepwise refinement process," in *Proc. 18th Annu. ACM Comput. Sci. Conf.*, Washington, DC, Feb. 1990.

[12] G. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychological Rev.*, vol. 63, no. 2, 1956.

[13] H. A. Simon, *The Sciences of the Artificial.* Cambridge, MA: M.I.T. Press, 1969.

[14] M. S. Campbell, "Chunking as an abstraction mechanism," Carnegie-Mellon Unv., Comput. Sci. Dep. Tech. Rep. CMU-CS-88-116, Feb. 1988.

[15] J. E. Laird, P. S. Rosenbloom, and A. Newell, "Chunking in SOAR—The anatomy of a general learning mechanism," *Machine Learning*, vol. 1, no. 1, pp. 11-46, 1986.

[16] P. S. Rosenbloom, "The chunking of goal hierarchies," Ph.D. dissertation, Comput. Sci. Dep., Carnegie-Mellon, Univ., 1983.

[17] J. C. Esteva, "Learning to recognize reusable software modules by induction," Ph.D. dissertation, Comput. Sci. Dep., Wayne State Univ., Detroit, MI, 1989.

[18] R. G. Reynolds, "A metrics-based model for reasoning about pseudocode design," *J. Inform. Software Technol.*, vol. 29, no. 9, pp. 497-502, Nov. 1987.

[19] ——, "PMS: An inference system to monitor the stepwise refinement of ADA pseudocode," *IEEE Expert*, vol. 2, no. 4, pp. 43-49, Winter 1986.

[20] S. Roy and J. Mostow, "Parsing to learn fine grained rules," Dep. Comput. Sci., AI/Design Project Working Paper 69, Rutgers Univ., June 1988.

[21] M. Tambe and A. Newell, "Why some chunks are expensive," Carnegie-Mellon Univ., Comput. Sci. Dep., Tech. Rep. CMU-CS-88-103, Jan. 1988.

[22] B. MacLennan, "Simple metrics for programming languages," *Inform. Processing Management*, vol. 20, no. 1/2, pp. 209-221, Jan. 1984.

[23] J. I. Maletic, "Refinement metrics: Assessing a programming language's support of the stepwise refinement process," M.S. Thesis, Comput. Sci. Dep., Wayne State Univ., Detroit, MI, 1989.

[24] P. H. Winston, *Artificial Intelligence.* Reading, MA: Addison-Wesley, 1984.

[25] R. G. Reynolds, "Metrics to measure the complexity of partial programs," *J. Syst. Software*, vol. 4, no. 1, pp. 75-91, Apr. 1984.

[26] M. H. Halstead, *Elements of Software Science.* New York: Elsevier-North Holland, 1977.

[27] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 308-320, Dec. 1976.

[28] C. McClure and J. Martin, *Software Maintenance: The Problem and Its Solution.* Englewood Cliffs, NJ: Prentice-Hall, 1983.

[29] J. Kolodner and C. Riesbeck, "Case-based reasoning: A tutorial," in *Proc. Eleventh Int. Joint Conf. Artificial Intell.*, Detroit, MI, Aug. 21, 1989. Los Altos, CA: Morgan Kaufmann, 1989.

[30] R. Williams, "Learning to program by examining and modifying cases," in *Proc. Case-Based Reasoning Workshop*, J. Kolodner, Ed., Clearwater, FL, May 10-13, 1988. Los Altos, CA: Morgan Kaufmann, 1988.
[31] G. Booch, *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin Cummings, 1987.
[32] R. G. Reynolds, "The control of genetic algorithms using version spaces," Wayne State Univ. Comput. Sci. Dep. Tech. Rep., Detroit, MI, 1990.

Robert G. Reynolds (M'80) received the Ph.D. degree in computer science from The University of Michigan in 1979 with a specialization in artificial intelligence (machine learning).

He joined the faculty at Wayne State University, Detroit, MI, in 1983 and is currently an Associate Professor. His main area of interest is automatic programming systems. At Wayne State University, he heads the PM project which deals with the use of machine learning tools in the acquisition of software engineering knowledge. He has authored or coauthored over 60 papers and a book.

Dr. Reynolds is a member of the IEEE Computer Society, the Association for Computing Machinery, and AAAI.

Jonathan I. Maletic received the M.S. degree in computer science from Wayne State University in 1989 and the B.S. degree from The University of Michigan, Flint, in 1985.

He is currently a Ph.D. candidate in computer science at Wayne State University. His research interests are in artificial intelligence, particularly in automatic programming, machine learning, and applications of AI to software engineering.

Mr. Maletic is a member of the Association for Computing Machinery and AAAI.

Stephen E. Porvin (S'89-M'89) received the B.A. degree from The University of Michigan in 1981 and the B.S. degree in computer science from Wayne State University in 1986.

He is currently finishing the M.S. degree in computer science at Wayne State University. As a Unisys employee he worked on BEACON, an expert system configurator and pricer. A member of the IEEE Computer Society, the Association for Computing Machinery, SIGART, and AAAI, his research interests include knowledge engineering, expert systems, and the application of artificial intelligence to software engineering.