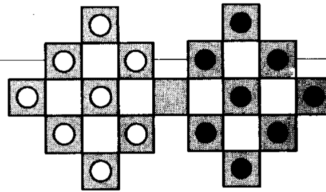


Knowledge-Based Program Analysis

Mehdi T. Harandi and **Jim Q. Ning**, *University of Illinois*

PAT provides high-level support for program maintenance. It uses an object-oriented framework of programming concepts and a heuristic-based concept-recognition mechanism to understand programs.

Without an adequate understanding of a program's meaning, it is impossible to maintain it effectively. This is especially true for large, complex programs. To modify a program, a programmer usually develops a mental model of its intended function. He then uses this model as a basis when he modifies the intended function or corrects the encoded implementation of the function. However, it is very difficult to construct such a mental model. Without automated support, a large part of the maintenance time is spent trying to understand what is to be maintained.

In this article, automatic program analysis is both the mechanized process of understanding high-level concepts from program text and the use of those concepts to guide program maintenance. The understanding element constructively derives a program's underlying meaning by statically examining its source code without using any specification or execution information.[1,2] Maintenance support offers high-level assistance to the maintainer in documentation, correction, enhancement, and other maintenance activities. While high-level support for program maintenance is the goal, program understanding is the means to achieve this goal.

We have realized this notion of automated program analysis in our knowledge-based Program Analysis Tool. PAT uses an object-oriented framework to represent programming concepts and a heuristic-based concept-recognition mechanism to derive high-level functional concepts from the source code.

Program views

Conceptually, you can view a program from different levels of detail. Program understanding transforms a program from a more detailed view into a more abstract view. Based on the abstraction level, we classify program views into four broad categories: implementation-level, structure-level, function-level, and domain-level views.[3,4]

The implementation-level view abstracts away a program's language- and implementation-specific features. To understand a program at this level, you need

0740-7459/90/0100/0074/$01.00 © 1990 IEEE

knowledge of the language's syntax and semantics and, possibly, some knowledge of the implementation. Typically, an implementation-level view is represented as an abstract syntax tree and a symbol table of program tokens.

The structure-level view further abstracts a program's language-dependent details to reveal its structure from different perspectives. The result is an explicit representation of the dependencies among program components. Examples of structure-level views are dataflow and control-flow graphs, data and control dependency graphs, interprocedural calling relations, ripple-effect graphs, petri nets, structure charts, and other intermediate-to-low-end design graphs. Recently, some effort has been made to generalize these representations to capture all the interesting structural features of programs in a unified representation.[5-7]

The function-level view relates pieces of the program to their functions to reveal the logical (as opposed to the syntactical or structural) relations among them. Each component of a function-level view is an abstract representation of a class of functionally equivalent, but structurally different, implementations.

The domain-level view further abstracts the function-level view by replacing its algorithmic nature with concepts specific to the application domain. For example, in the context of student-record keeping, a program functionally understood as "computing average by summing its inputs divided by the number of inputs" is interpreted as a "grade-point-average computation" routine.

Figure 1 shows how these abstraction categories roughly correspond to the information used in different stages of the development life cycle. While this article's focus is on the function-level view, you can easily extend the methods and tools presented here to deal with domain-level understanding.

Expert's model

Observations show that human experts have a better problem-solving model than previous automatic-program-analysis systems: They can usually comprehend a program efficiently without using a formal method of proof. To understand a program, experts do not exhaustively apply all their knowledge about programming to repeatedly transform the program. Nor do they extract all the information about dataflow and control flow to make abstractions.

An expert views a program not only as a text file of sequenced characters but also as a set of interrelated concepts. He understands a program by learning abstract concepts from it. Initially, he may understand a program only syntactically. Then, discrete, otherwise unrelated low-level concepts may help him recognize higher level concepts until he can comprehend the whole program as a single functional unit.

He uses his programming knowledge to recognize high-level concepts. Typically, this knowledge includes stereotyped code patterns of common programming strategies, data structures, and algorithms. When he sees a concept's stereotyped pattern, he looks for evidence that suggests its existence. This concept-recognition process results in a plausible conclusion, rather than a rigorous proof.

Using this heuristic-based knowledge, he skips trivial parts and looks only for things he deems important. He can relate concepts that are not adjacent because all the concepts in his concept base — simple and complex — are equally visible at all times. In the end, he forms a functional model of the program, usually a hierarchical structure that relates all the concepts recognized and rooted in the original concepts. He then uses this model to guide maintenance.

PAT overview

The PAT system, illustrated in Figure 2, is based on the human expert's analysis model. PAT tries to help maintainers answer three questions:

- What does this program do (what high-level concepts does it implement)?
- How are the concepts encoded, in terms of low-level concepts?
- Are the recognized concepts implemented incorrectly?

To do this, the Program Parser first rewrites the program into a set of language-independent objects, called events, and puts them in the Event Base. Using this event set, the Understander recognizes higher level events that represent more function-oriented concepts. The Understander adds these newly recognized

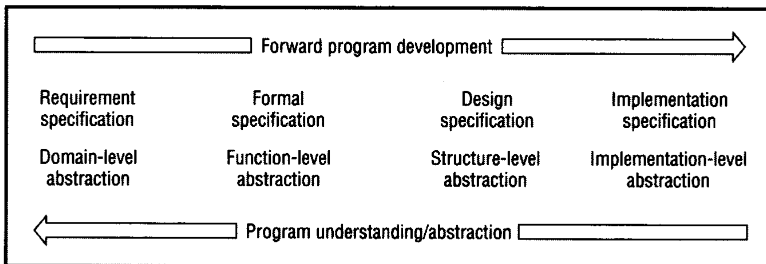


Figure 1. Diagram of forward program development and backward program abstraction.

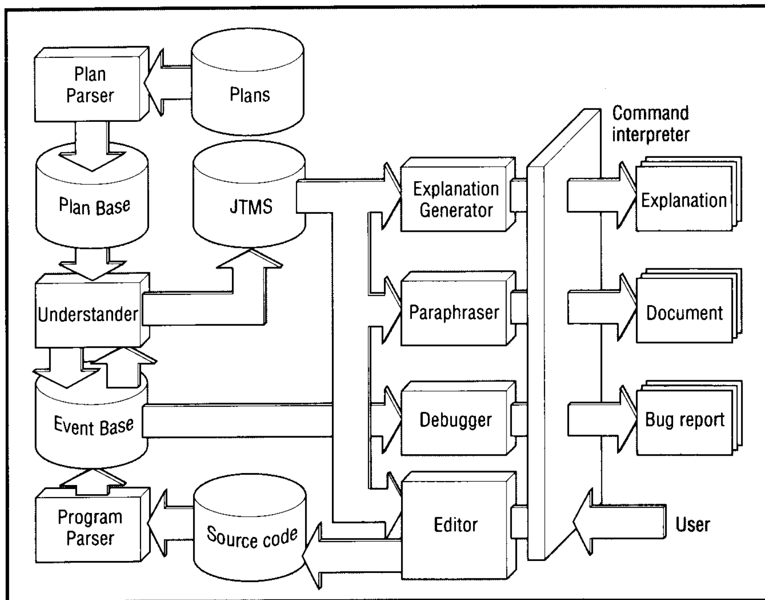


Figure 2. PAT's architecture.

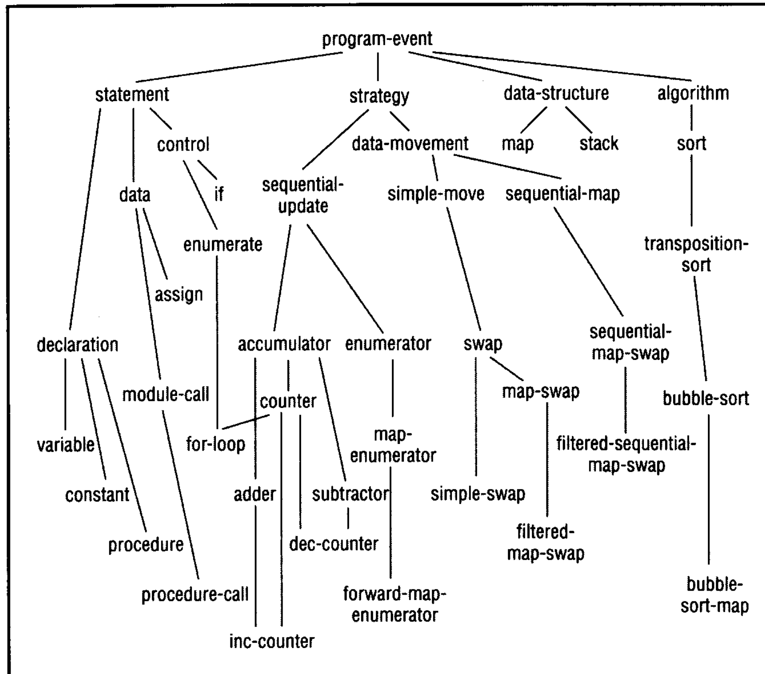


Figure 3. Part of the program event classification hierarchy.

events to the event set and repeats the process until it recognizes no more high-level events. The final event set, presented to the maintainer, answers the first question.

The Understander's main component is a deductive-inference-rule engine. It uses a library of program plans, stored in a plan base, as inference rules to derive new, high-level events. The program plans, which have been parsed by the Plan Parser, contain understanding, paraphrasing, and debugging knowledge. When the Understander generates a new event, it may trigger other rules to fire, causing the derivation of more events.

Discovering new events is of little use without the ability to explain the logical connections among them. To do this, PAT maintains a justification-based truth-maintenance system to model the understanding process.[8] When the Understander identifies a new event, the JTMS records the result *and* its justifications. The Explanation Generator uses the JTMS to show how high-level events are derived from the low-level events, thus answering the second question.

For example, the Explanation Generator gives the following explanation when an original set of events {*s*1, *s*2, *s*3} causes the recognition of a new set of events {*e*1, *e*2}, where *e*1 is derived from *s*1, *s*2, and *e*2 is derived from *s*1, *s*3, and *e*1:

1. *s*1 is a simple event.
2. *s*2 is a simple event.
3. *s*3 is a simple event.
4. *e*1 is a composite event based on 1 and 2.
5. *e*2 is a composite event based on 1, 3, and 4.

The Paraphraser translates these explanations into natural-language descriptions.

The Debugger examines the final set of recognized events to answer the third question. Each program plan contains knowledge on near-miss implementation patterns that are commonly associated with events that are recognizable by that plan. The Debugger uses this information to identify a possible misimplementation.

Finally, the Editor lets you interactively modify the program; such changes may trigger more inferences, the results of which are updated in the JTMS automatically.

Knowledge representation

PAT represents two types of knowledge explicitly: program knowledge and analysis knowledge. Program knowledge is represented by programming concepts contained in program text. Analysis knowledge embodies information necessary for program analysis and is represented by information contained in program plans.

Program knowledge. In our paradigm, each syntactic or semantic concept contained in a program is expressed in an object-oriented abstract representation, called a program event. Program events are organized in a hierarchy. At the lowest level — the source level — are events representing language constructs like statements and declarations. At a higher level

are events corresponding to common programming patterns and strategies like structure enumerators, accumulating a sequence of values, and counting. Events can also represent data structures or designs like stacks, queues, trees, and their corresponding operations. At an even higher level, events can represent standard algorithms for common problems like mathematical computation algorithms, searching, and sorting.

Whatever it represents, each event is an instance of an event class. Figure 3 shows a partial hierarchy of event classes.

All events have attributes. Each event has an *interval*, which comprises two parts: a control interval and a lexical interval. The control interval determines where the event is in the control path when the code is executed. The lexical interval determines where the event is in the nested hierarchy of the program text. An event also has an external form, for presentation.

Events have an event-class attribute, which denotes their class. We define common attributes in the top-level class (the program-event), and they are inherited by all classes. In addition to inherited attributes, an event may have its own attributes, as Figure 4 shows.

Figure 4 shows a program segment and the event representation of the concept contained in the segment, simple-swap. In Figure 4, the control-interval, [0 (120 43 44 45)], says that this simple-swap event comprises three subevents at locations 43, 44, and 45 and that they are in a module (usually a procedure) that is invoked from location 120. The event at location 120 is part of the main program module. We always assume that the control to the main program is transferred from some imaginary location 0.

The lexical-interval [0 (4 43 44 45)] says that the module that lexically encloses the simple-swap event is numbered 4 (its block number), which is globally declared. The main program module has block number 0.

The local attributes, var1, var2, and temp-var, are variables in the swap operation. Because the value of temp-var should not affect the external behavior of simple-swap, it does not appear in the external-form attribute. Each data object has a subscript that indicates its declaring block. For example, B_0 says that the data object B is declared globally. We subscript data objects to distinguish multiply declared identifiers in different lexical environments. A language parser and a simple control-flow analyzer determine the attributes of source-level events; the attributes of higher level events are computed from their composing events.

```
        ...
    38  procedure swap ( var X, Y : integer);
    40  var T : integer;
    43  assign X to T;
    44  assign Y to X;
    45  assign T to Y;
    47  end-procedure;
        ...
   120  proc-call swap (A, B);
        ...
```

event-class:	SIMPLE-SWAP
interval:	([0 (120 43 44 45)] [0 (4 43 44 45)])
external-form:	((43 44 45) SIMPLE-SWAP A B)
var1:	A_0
var2:	B_0
temp-var:	T_4

Figure 4. Event representation of the simple-swap concept.

plan *event*
- **path** *event-path-expression*
- **test** *binding-constraints*
- **text** *documentation-information*
- **miss** *near-miss-expression*

where the *event-path-expression* is defined as:

event-path-expression	::= *event-specifier* | *interval-operator* {*event-path-expression*} *
event-specifier	::= { **key** } *event*
interval-operator	::= *c-operator* | *l-operator*
c-operator	::= **c-precede | c-enclose | c-interleave | c-overlap c-contain | c-meet | c-sequential | c-parallel | ...**
l-operator	::= **l-precede | l-enclose | l-interleave | l-overlap | l-contain | l-meet | l-sequential | l-parallel | ...**

Figure 5. Plan-definition syntax.

Analysis knowledge. In PAT, knowledge about program understanding, documentation, and debugging is represented as a program plan.

Figure 5 shows the syntax of a plan definition. Understanding knowledge is encoded in the plan's path and test sections. An event-path expression in the path part specifies the lexical and control sequence requirements of a subset of the plan's event patterns. A pattern might match a source-level event, such as an assignment, or it might match a high-level concept, such as an enumerate. An event set is an instance of a plan if it meets the path expression of the plan and any constraints expressed in the test part.

Knowledge to generate documentation is stored in the text part and knowledge to perform near-miss debugging is stored in the miss part.

To understand event-path expressions and interval operators (logical operators we use to define lexical and control sequencing requirements), examine the event-path-expression part of an accumulator plan:

```
plan (accumulator :update-var ?var
  :init-value ?init :update-value ?val
        :update-cond ?cond
          :accumulator-op ?op)
   path ( c-precede (assign :var-defined ?var
     :value-used ?init)
    ( c-enclose (enumerator :loop-cond
      ?cond)
      ( key (assign :var-defined ?var
        :value-used (?op ?var ?val)))))
```

In this plan, : denotes an attribute and ?

```
...  ...
203 const          N  :  integer = 100;
205 var            A  :  array(1..N, Elem);
...  ...
225 procedure     unknown ();
228     var S  :  integer;
230     for-loop  K  from  N–1  to  1  do
232         for-loop  J  from  1  to  K  do
233             if  A(J–1) > A(J)  then
234                 assign  A(J)  to  S;
235                 assign  S  to  A(J–1);
236                 assign  A(J–1)  to  A(J)
238             end-if
240         end-for-loop
245     end-for-loop
250 end-procedure;
...  ...
400 proc-call unknown();
...  ...
```

Figure 6. Bubble-sort program.

```
E205:    event-class:   VAR
         interval:      ([0 205] [0 205])
         name:          A0
         var-type:      array(1..N0,Elem0)
E232:    event-class:   FOR-LOOP
         interval:      ([0 (400 232 240)] [0 (45 232 240)])
         update-var:    J0
         init-value:    1
         final-value:   K0
         direction:     UP
         step:          1
E233:    event-class:   IF
         interval:      ([0 (400 233 238)] [0 (45 233 238)])
         if-cond:       A0(J0–1) >A0(J0)
E235:    event-class:   ASSIGN
         interval:      ([0 (400 235)] [0 (45 235)])
         var-defined:   A0(J0–1)
         value-used:    S45
```

Figure 7. Source-level events generated from the program in Figure 6.

denotes a pattern variable. The path expression specifies two assignment events, one enumerator event (a loop construct), and their variable bindings — a possible component set for an accumulator event.

The path part also requires that ?var be initialized to some value ?init before the loop and the loop-carried assignment events are reached. So an accumulator event is identified only if the initial assignment event precedes the loop event on the control path (c-precede) which, in turn, encloses the second assignment event (c-enclose).

Key events identify important plan components. In this example, the second assignment event is a key; it must be identified first to recognize the accumulator event. Identifying key events first helps reduce the search space.

Event-path expressions heuristically categorize classes of equivalent event sequences, which may not be lexically adjacent. As long as their relative positions meet the lexical and control requirements expressed by the path, they are recognized as components of a higher level event.

Program analysis

PAT's understanding power comes from a pattern-directed inference engine that uses a plan library. Plans are represented as inference rules that are stored in the plan base.

Plan rules are triggered by events defined in the plan's event-path expression. The control and lexical requirements, extracted from the event-path expression, combined with a plan's binding constraints, govern the firing of a rule. The rule body is always an assertion that declares a new event when the trigger patterns and test conditions are satisfied.

Program understanding is automated as an inference by which new events are inferred from existing ones using the plan rules. Event *E* matches a trigger pattern *P* of plan rule *R* if

- either *E* and *P* are in the same event class or *P* is in a superclass of *E*; and
- for any attribute *A* with value *V*1 specified in *P*, there is an attribute *A* in *E* with value *V*2 such that *V*1 and *V*2 are unifiable, given pattern-unification bindings.

The first condition says that a trigger pattern can match with events that are more specific than it is. For example, an array-search pattern could match not only an array-search event but also a linear-array-search event or a binary-array-search event.

The second condition says that *P* matches *E* as long as the information in *P* is subsumed by (not necessarily equal to) the information in *E*. In defining the plan pattern array-search, for example, you need specify only the attributes containing the array name and the value of the search target. A binary-array-search event will match this pattern, although it may have extra attributes such as the array index pointers.

These conditions guarantee that when a new event is asserted into the event base the inference engine must rerun only those plan rules that are in the same class or in superclasses of the new event. Similarly, when a new rule is added to the plan base, the inference engine will apply the new rule only to events in the same class or subclasses of the rule's class. The JTMS records the results of the inference process.

The reasoning procedure in PAT is less formal than that used in other deduction, transformation, parsing, or graph-matching approaches. A PAT analysis cannot rigorously prove anything because it is a selective inspection, not a total reduction of a program. Our intention is to capture human experts' behavior in program understanding so we can handle programs

P_{50}: If there exists a decremental FOR-LOOP event
then there exists a DEC-COUNTER event.

P_{51}: If there exists an incremental FOR-LOOP event
then there exists an INC-COUNTER event.

P_{52}: If there exists an ASSIGN event from ?Var1 to ?Temp which precedes an ASSIGN ?Temp to ?Var2 event and another ASSIGN ?Var2 to ?Var1 event on a control path (c-precede)
then there exists a SIMPLE-SWAP(?Var1, ?Var2) event.

P_{53}: If there exists a VAR event ?A of type array(?L..?U,?Type)
then there exists a MAP(?A,?L..?U,?Type) event.

P_{54}: If there exists a MAP(?A,?L..?U,?Type) and the definition of which lexically precedes (l-precede) an INC-COUNTER event indexing through ?A
then there exists a FORWARD-MAP-ENUMERATOR event on ?A.

P_{55}: If there exists a SIMPLE-SWAP ?Var1 and ?Var2 event in which ?Var1 and ?Var2 access a MAP(?A,?L..?U,?Type) event
then there exists a MAP-SWAP event.

P_{56}: If an IF event lexically encloses c-enclose an MAP-SWAP event
then there exists a GUARDED-MAP-EVENT event.

P_{57}: If a FORWARD-MAP-ENUMERATOR event c-encloses an GUARDED-MAP-SWAP event
then there exists a FILTERED-SEQUENTIAL-MAP-SWAP event.

P_{58}: If a DEC-COUNTER event c-encloses an FILTERED-SEQUENTIAL-MAP-SWAP event
then there exists a BUBBLE-SORT-MAP event.

Figure 8. Plans used in understanding the program in Figure 6.

with missing, extra, or buggy parts, and avoid the combinatorial barriers in the analysis of large programs.

Maintenance support

Based on the final structure of the JTMS net, PAT's Explanation Generator can informally show how the high-level events are derived from the low-level ones. The explanation it provides helps verify the correctness of the conclusions and reveal the functional and logical relations among the program components represented by the recognized events.

Each plan has a text slot that identifies the intended function of the event it is supposed to recognize. This text can be natural-language statements or a formal specification. By tracing the JTMS's net from the top, the PAT Paraphraser generates program documentation using the information in each event's text slot and explanations of how each event is composed of subevents. This documentation helps maintainers find discrepancies between intended and implemented functions.

The miss part of a plan definition contains heuristic knowledge for diagnosing common coding errors in the plan's target event. Typically, a plan's event-path expression is intentionally relaxed so it will recognize correct and buggy patterns as plan instances. The buggy part will be examined only following a successful recognition pass. Allowing near-miss recognition of events may help find very deep bugs that are otherwise very difficult to detect.

The JTMS maintains a network structure that connects the current set of believed events about the program. When a user modifies the program, those changes will be automatically reflected in the JTMS, which relates a modification's effect on the implemented functions directly, making the modification easier to follow.

Example

Figure 6 is a segment of a much larger program written in a Pascal-like language. The maintainer wants to understand only this segment; he does not want to analyze the entire program, so the information provided to PAT is incomplete. The definitions of variables *K* and *J* are invisible, we have no idea what the initial value of array *A* is, and we know nothing about its component's type.

Furthermore, this portion is buggy: To exchange the contents of *A*(*J*) and *A*(*J* −1) correctly, the assignment at line 236 should occur before the assignment at line 235. Also, the range of looping variable *J* in the second For loop at line 232 should have been from 2 to *K*+1, not from 1 to *K*. Finally, the two indices of *A* could be substituted by [*J*, *J*+1]. This segment also includes noise in the lines indicated by ellipses.

The information we have about this segment is not sufficient to prove formally that the program does a bubble sort, but we can reach such an understanding based on our knowledge of typical bubble-sort implementation patterns. A human expert would assume that array *A* has been initialized somewhere else before the program control reaches this segment, unless he sees an explicit contradiction.

PAT first parses the segment, recognizing a set of events that represent the program's source-level concepts: events representing each variable and constant definition, an event representing the procedure, six events corresponding to statements in the procedure, and an event representing the call at line 400.

Figure 7 shows four of these events. The interval definitions in Figure 7 indicate that the procedure unknown and the variables *J*, *K*, and Elem are declared at block 0 (the global environment) and that the procedure's block number is 45.

After the events are loaded into the event base, the Understander calls the plans from the plan base and tests if their trigger events match the input events. Figure 8 shows the plans used in this example, which are expressed in English to aid comprehension.

Plan P50 is triggered by event E230 (the For loop at line 230) and generates a new event E1001 (dec-counter). Event E232 triggers plan P51, generating E1002. The combination of events E234, E235, and E236 trigger P52 to generate E1003 (simple-swap). Next, E205 triggers P53 to generate event E1004. The combination of E1004 and E1002 triggers P54 to generate E1005. E1003 triggers P55 to generate E1006. P56 is triggered by E233 and E1006 to generate E1007. E1005 and E1007 trigger P57 to generate E1008. Finally, E1001 and E1008 trigger P58 to generate E1009, which is the bubble-sort algorithm.

The JTMS keeps track of the derivation of new events; Figure 9 shows its final structure, annotated with the names of recognized events. Using the text information contained in the recognized

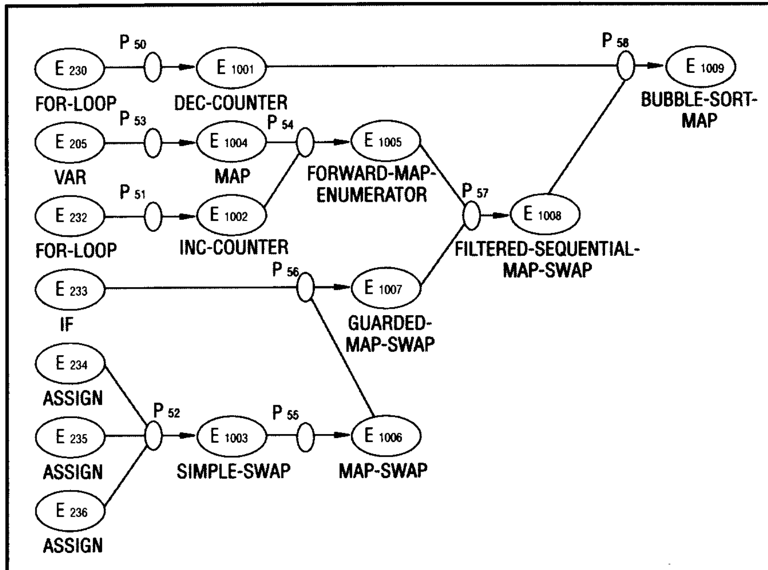


Figure 9. JTMS model of the program in Figure 6.

This program implements a BUBBLE-SORT-MAP event at lines
(205 230 232 233 234 235 236 238 240 245)
which sorts the map A using a bubble sort algorithm.
It consists of
1. A DEC-COUNTER event at lines (230 245)
   which decrementally changes the value in K from N – 1 to 1.
   It consists of
   1.1 A FOR-LOOP event at lines (230 245).
2. A FILTERED-SEQUENTIAL-MAP-SWAP event at lines
   (205 232 233 234 235 236 238 240)
   which sequentially switches the adjacent elements in a map A
   if A(J–1) > A(J), indexed by J from 1 to K.
   It consists of
   2.1. A FORWARD-MAP-ENUMERATOR event at lines (232 240)
   which incrementally enumerates the elements in map A indexed by J
   from 1 to K. It consists of
   2.1.1. A MAP event named A at line (205)
   which is a mapping from 1..N to Elem.
   It consists of
   2.1.1.1. A VAR event named A at line (205).
   2.1.2. An INC-COUNTER event at (232 240)
   which incrementally changes the value in J from 1 to K.
   It consists of
   2.1.2.1. An FOR-LOOP event at lines (232 240).
   2.2. A GUARDED-MAP-SWAP event at lines (233 234 235 236 238)
   which switches the values in A(J–1) and A(J) if A(J–1) > A(J).
   It consists of
   2.2.1. An IF event at lines (233 238).
   2.2.2. A MAP-SWAP event at lines (234 235 236).
   It consists of
   2.2.2.1. A SIMPLE-SWAP event at lines (234 235 236)
   which switches the values in A(J–1) and A(J).
   It consists of
   2.2.2.1.1. An ASSIGN event at line (234).
   2.2.2.1.2. An ASSIGN event at line (235).
   2.2.2.1.3. An ASSIGN event at line (236).

Figure 10. Paraphrase of the example program.

events, PAT can explain its understanding, as the paraphrase in Figure 10 shows.

As for the two bugs in this segment, without a function-level understanding of the program, we can only point out that the initial value of $A(J-1)$ is not used in each iteration in the inner loop and that $J-1$ is less than the lower bound of A when J equals 1.

In our paradigm, event-specific debugging information is encoded in the miss part of a plan. For example, in the plan for recognizing the simple-swap event, we intentionally relax the sequencing requirement in its event-path expression, encoding it instead in its miss part. When the Understander recognizes the three assignment events as components of a simple-swap event, it reexamines their control sequence. PAT then determines that

- the three events are intended to accomplish a simple-swap event (they meet the event-path expression) and
- the positions of the last two should be switched (they meet the near-miss expression).

Similarly, when PAT recognizes the bubble-sort-map event, it unifies the 1, N, $J-1$, and J expressions with the plan's variables, producing the bindings [LowBound 1], [UpBound N], [Index J], [OffSet1 –1], and [OffSet2 0]. If the miss part includes the rules

```
If Index runs increasingly
then The first round of Index values must
  range
    from LowBound – OffSet1 to
      UpBound – OffSet2;
  The last Index value must be
    LowBound – OffSet1.
else The first round of Index values must
  range
    from UpBound – OffSet2 to
      LowBound – OffSet1;
  The last Index value must be
    UpBound – OffSet2.
```

PAT can determine that A is indexed incorrectly by J because the first round of J values (when K takes its first value $N-1$) ranges from 1 to $N-1$, not from 2 (LowBound – OffSet1) to N (UpBound – OffSet2). Besides, because J runs increasingly, the last J value (when K takes its final value 1) should be 2 (LowBound – OffSet1), not 1.

Experiments with PAT have affirmed our initial expectations. PAT now includes about 100 program-event classes that represent language constructs, coding heuristics, data-structure definitions and operations, and functional coding patterns.

PAT's plan base contains a few dozen plan rules covering value accumulation, structure enumeration, simple mathematical computations, counting, sequential search of ordered and unordered structures, different types of searching, tree traversals, and sorting. For practical applications, we believe PAT will need at least several hundred event classes and plans.

PAT could be improved in several ways. It should be able to incrementally acquire program knowledge by asking an expert for help when it cannot understand a program and by saving the generalized solution for future use. Also, when PAT comes up with a conclusion that an expert rejects, it should know how to modify its knowledge to account for the failure.

PAT is not intended to replace maintenance done by people. Instead, it is a high-level assistant that provides plausible predictions, suggestions, and explanations about a program's function — information that is not easily derived with traditional maintenance tools. ❖

Acknowledgment

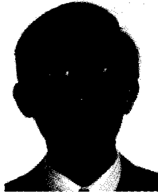
This work was supported in part by IBM.

References

1. M.T. Harandi and J.Q. Ning, "PAT: A Knowledge-Based Program-Analysis Tool," *Proc. Conf. Software Maintenance*, CS Press, Los Alamitos, Calif., 1988, pp. 312-318.
2. J.Q. Ning, *A Knowledge-Based Approach to Automatic Program Analysis*, doctoral dissertation, University of Illinois at Urbana-Champaign, Urbana, Ill., 1989.
3. J.Q. Ning and M.T. Harandi, "An Experiment in Automating Code Analysis," *Proc. AAAI Symp. Artificial Intelligence and Software Engineering*, AAAI Press, Stanford, Calif., 1989, pp. 51-53.
4. W. Kozaczynski and J.Q. Ning, "SRE: A Knowledge-Based Environment for Large-Scale Software Reengineering Activities," *Proc. Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif, 1989, pp. 113-122.
5. J.M. Bieman and N.C. Debnath, "An Analysis of Software Structure Using a Generalized Program Graph," *Proc. Compsac*, CS Press, Los Alamitos, Calif., 1985, pp. 254-259.
6. N. Wilde, R. Ogando, and E. Edge, "Specification for Prototype Dependency Analysis Tools," Tech. Report SERC-TR-13-F, Software Engineering Research Center, University of Florida, Tallahassee, Fla., 1987.
7. S. Yau and P.C. Grabow, "A Model for Representing Programs Using Hierarchical Graphs," *IEEE Trans. Software Eng.*, Nov. 1981, pp. 556-574.
8. J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, Vol. 12, 1979, pp. 231-272.

Mehdi T. Harandi is an associate professor of computer science at the University of Illinois at Urbana-Champaign and is director of the university's knowledge-based progamming assistant project. His research interests include knowledge-based systems, software specification and design, and AI applications to software development.

Harandi received a PhD in computer science from the University of Manchester, England. He is editor-in-chief of *International Journal of Expert Systems: Research and Applications* and a member of ACM and the IEEE Computer Society.



Jim Q. Ning is an associate scientist at Arthur Andersen and Company's Strategic Technology Research Center. His research interests are forward and reverse engineering, object-oriented methods, knowledge-based systems, and programming knowledge representation.

Ning received a BS in electronics from Beijing Normal University and an MS and PhD in computer science from the University of Illinois, where he did the work reported here.

Address questions about this article to Harandi at Computer Science Dept., University of Illinois, 1304 W. Springfield Ave., Urbana, IL 61801.

Books You Can't Put Down

The technical literature is ever expanding. Which books should you read? Which should you avoid?

IEEE Software reviews books with your needs in mind, whether you are a practitioner or educator. Each issue, Mike Lutz selects recent books spanning software's diverse theories, practices, and philosophies and has experts in the field evaluate them. Our Book Reviews department has an eclectic mix to keep you abreast.

We're what a technical magazine should be: Practical. Authoritative. Lucid. Direct.

For subscription information, write *IEEE Software*, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264; call (714) 821-8380; or use the reader-service card.

IEEE Software

The state of the art about the state of the practice.

Critical Examination

You depend heavily on your software tools. But how do you choose the right ones?

IEEE Software removes the hype with in-depth reviews of widely used, industrial-strength software. Each issue, Paul Oman selects knowledgeable reviewers to examine a class of product and evaluate them in an at-work environment. Our Software Test Lab department gives you the technical evaluation you need to make a choice.

We're what a technical magazine should be: Practical. Authoritative. Lucid. Direct.

For subscription information, write *IEEE Software*, 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1264; call (714) 821-8380; or use the reader-service card.

IEEE Software

The state of the art about the state of the practice.