Knowledge-Based Program Analysis

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. 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.

The implementation-level view abstracts away a program's language- and implementation-specific features. To understand a program at this level, you need...
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.34

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 Understannder recognizes higher level events that represent more function-oriented concepts. The Understannder adds these newly recognized concepts to the abstract program model, allowing questions to be answered.

---

**Figure 1.** Diagram of forward program development and backward program abstraction.
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. 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 \( [\pi, \sigma, \delta] \) causes the recognition of a new set of events \( \eta \), where \( \pi \) is derived from \( \pi_1, \pi_2 \), and \( \sigma \) is derived from \( \sigma_1, \sigma_3 \), and \( \delta \):

1. \( \pi_1 \) is a simple event.
2. \( \pi_2 \) is a simple event.
3. \( \sigma_3 \) is a simple event.
4. \( \delta \) is a composite event based on 1 and 2.
5. \( \eta \) is a composite event based on 1, 3, and 4.

The Paraphraser translates these explanations into natural-language descriptions.

The Debugger translates 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 [(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 tempvar, are variables in the swap operation. Because the value of tempvar should not affect the external behavior of simple-swap, it does not appear in the external-form attribute. Each data object has a sub

```plaintext
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);
```

**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-precede | c-enclose | c-interleave | c-overlap
- l-operator ::= l-precede | l-enclose | l-interleave | l-overlap

```

**Figure 5.** Plan-definition syntax.

script that indicates its declaring block. For example, B2 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.

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 $top
  path (c-precede (assign var-defined $var value-used $init)
    (c-enclose (enumerator $loop-cond
      (key (assign var-defined $var value-used ($top $var $val)))))

In this plan, $ denotes an attribute and ?
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 \( \hat{\text{init}} \) 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 \( V1 \) specified in \( P \), there is an attribute \( A \) in \( E \) with value \( V2 \) such that \( V1 \) and \( V2 \) 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 return 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.
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 provides help verify the correctness of the conclusions and reveals 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 relays 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 *kand f 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 *f in the second For loop at line 232 should have been from 2 to *K, not from 1 to *K. Finally, the two indices of *A could be substituted by [*f, *f + 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 it 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 Understand 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 derivations of new events; Figure 9 shows its final structure, annotated with the names of recognized events. Using the text information contained in the recognized
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. A 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 9. JTMS model of the program in Figure 6.

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 Understaner 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 expresion).

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], [OffsetSet −1], and [OffsetSet 0]. If the miss part includes the rules

If Index runs increasingly
then The first round of Index values must range
from LowBound − OffsetSet to
UpBound − OffsetSet;
The last Index value must be
LowBound − OffsetSet;
else The first round of Index values must range
from UpBound − OffsetSet to
LowBound − OffsetSet;
The last Index value must be
UpBound − OffsetSet.

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 − OffsetSet1) to N (UpBound − OffsetSet2). Besides, because j runs increasings, the last j value (when k takes its final value 1) should be 2 (LowBound − OffsetSet1), 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.

Acknowledgment
This work was supported in part by IBM.

References