The Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Working Paper 237 August 1982

Automated Program Description

D. Scott Cyphers

ABSTRACT

The *programmer's apprentice* (PA) is an automated program development tool. The PA depends upon a library of common algorithms (*cliches*) as the source of its knowledge about programming. The PA uses these cliches to understand how a program is implemented. This knowledge may also used to explain to a user of the PA how the program is implemented.

The problem with any explanation or description is knowing how much information to present, and how much information to hide. A set of simple heuristics for doing this can be used with the cliche representation of a program to produce reasonable descriptions of parts of programs. The system described combines "canned" phrases corresponding to cliche parts to form explanations. The process is fast and appears to be easily extensible to future versions of the PA and other domains.

Acknowledgments

This paper is dedicated to Dick Waters and Chuck Rich, without whom it would not exist. Additional thanks go to Reid Simmons who provided much guidance, to David Chapman, who got me started on the Lisp Machine, and to UROP, who provided the opportunity.

I would like to thank Francine Chen and Chuck Rich for their proofreading.

CONTENTS

1. Objects

In this paper, an object is considered to be anything which may be thought about. Computers, circuits, electron movement, programs, data structures and thoughts are all examples of objects in this sense. Often, objects are composed of other objects via relationships between them. The properties of the components interact, as dictated by the relationships, to give the object its overall properties.

Objects have both extrinsic and implementational characteristics. The extrinsic properties describe how the object will interact with the world around it. The implementational details describe how the object achieves its extrinsic properties.

There is a distinction between properties and objects. "Finding the minimum element in a list of integers" is a property of a whole class of objects which can have a wide variety of implementations. Thus, there is a many to one relationship from extrinsic properties to implementations.

Similarly, there is a many to one relationship from an object to extrinsic properties. A particular instance of an object which finds the minimum *integer* in a list may also find the minimum integer in a vector when given a vector, the minimum integer value of an equation when given an equation, etc. A second object, similar to this object, may return the minimum *value* in a list, vector, or equation. In the context of integers, these functions are identical, whereas they are different in the context of floating point numbers. The first would return 1 when given the list '(1 .5 6) while the second would return .5.

Rarely will an object be describable simply by its component objects. There is an additional factor, the organization of the objects. The organization provides a way for the extrinsic properties of the components to interact, so as to produce the extrinsic properties of the composed object.

Given objects which divide, count, and add, an object may be constructed which averages. To do this, a counting object is used to count the number of things which are being averaged. The adding object adds the things together. The divide object divides the sum by the number of objects.

The average object is constructed by relationships between the objects which compose it. The output of the count is the denominator input of the divide. The output of the sum is the numerator input of the divide. The implementations of the divide, count, and sum are not needed for one to be able to design the average object. Only their extrinsic properties have been used. The result is an object which has the implementation as described, and an extrinsic description of computing an average.

In a sense, the organization is a higher level concept than the objects. It is very important, but also very easily hidden. A machine language program for an unfamiliar machine is a good example of this. It is composed of ones and zeroes, the exact order of which is very important, but the meaning of the order is not visible without running the program.

2. Descriptions

The above properties of objects lead to three types of questions which may be asked about an object:

1) What are the extrinsic characteristics of the object?

2) How are the extrinsic properties implemented?

3) In what context is the object used?

The extrinsic properties are often referred to as "black box descriptions" of objects. They may be used to provide an abstraction of the internal details and implementation of the object, so as to make it easier to use and think about. For example, the implementation of the square root function is often irrelevant to the use of it, and simply knowing that it returns the square root of its argument is enough information.

However, the extrinsic properties are often approximations to the actual collective object. Some square root functions may take longer to run than others, or may provide different accuracies. When this type of information becomes important, the model of the object provided by its external description may no longer be valid.

The implementation of an object is also important in practice. An intelligent reasoning system must be able to adjust the extrinsic properties of the object to meet the needs of the system, or a variation of the object may be wanted. To do this, the implementation of the object must be available.

An important property of any description of an object is that it provide enough information about the object to serve the needs of the user of the description, while omitting enough information so as not to overwhelm the user of the description with superfluous detail. This is done by finding the important decompositions of an object that will provide the desired description of the object and using the right combination of extrinsic and intrinsic explanations.

3. Program Explanation

Finding the important compositions to use in a description is a hard problem. However, if they are provided, reasonable descriptions may be formed relatively easily through a simple set of rules for selecting extrinsic and implementational descriptions for objects.

The programmer's apprentice (PA) provides the composition through cliches. The cliches are the organization of a grouping of objects, along with some of its components, to provide for generalized classes of objects used in programming. A program analyzed in terms of cliches is easily described.

In a cliche definition, roles are used to mark where components will be used in an object. With each role definition is associated a description of what that object's purpose is in the cliche. An overall description of the organization of the cliche is also provided. Finally, an extrinsic description is provided.

Since roles often correspond to objects which are not a part of the cliche itself, the implementation and the extrinsic description of the cliche may have places in them in which descriptions of the objects to be filled in may be inserted. The types of descriptions used depend upon their structural relationship to the object being explained and the cliche being used to explain the object.

All objects which surround the object being described provide contextual or detailed descriptions, whereas all other objects provide extrinsic descriptions. When an extrinsic description is not available, the description of the use of the object is used instead. This is often the case when describing cliches which are not fully implemented, or cliches which have lost the link between some roles and objects.

The program explanation system knows how to generate extrinsic descriptions of certain of the primitives in a program, such as function calls and variables. Functions simply become a phrase like "the FOO of ARG1, ARG2, ..." and variable names are simply found.

4. Previous Work

Previous work in the area of program explanation has been concerned mainly with the generation of a description from a mathematical description of a program, such as developed by Rich [2], for example the work of Reid Simmons (verbal communication) or Claude Frank [1]

5. Detailed Description of Program Explanation

When an object is being described, an explanation handler is found which corresponds to the object and the type of description. Handlers consist of either a string or a list of strings and symbols, the symbols being part or role names. The strings and extrinsic descriptions of all of the parts and roles are concatenated to form the description of the object.

There are two varieties of implementational descriptions. If the object is a cliche, the implementational handler for that cliche is used. Otherwise, a description of the form "<contextual description of object> implemented as <summary description of object>" is generated.

When a contextual description is needed for an object, there are three candidates for an explanation handler. The first is the part assertion about the object, if one exists. Since this is the handler which has been most tailored to the particular cliche in which the object is used, it will be the most informative. Alternatively, if there is no part assertion or if no handler exists for that part assertion, a role assertion is tried. Should this also fail, an extrinsic explanation is generated for the object, unless the contextual explanation was being generated as a substitute for a summary explanation, in which case the process fails.

A similar set of candidates exists for finding handlers for an extrinsic description. If the object is a cliche, the cliche provides the handler. Otherwise, an attempt is made to generate a contextual description of the object, unless the summary description was being formed as a substitute for an initial contextual description, in which case the process fails. Finally, a generic handler is generated, based upon the type of the object.

The motivation behind this ordering is based upon the belief that the generic handlers will provide the least information about the object, and, thus, should be used as a last resort. Extrinsic and contextual description are fairly closely related to each other, and, thus, may substituted for one another, should one fail to be capable of providing an explanation.

6. Some Examples

Due to the dependence of the program description process on the cliche decomposition of the program, it is best to first look at the design of a program with the PA before looking at its explanation. These examples are from standard Knowledge Based Editor (KBE) [3] demonstration sequences. In actual use, code is produced after each step. However, in this paper, only the final code for the program will be shown. In addition, cliche definitions, which include the explanatory phrases used in program description, are shown when they are used.

6.1 Square root and cliches

The first example is for a square root program. The user of the KBE indicates that a program is to be written:

```
>Define a program MYSQRT with a parameter NUM .
```

This creates the basic form of a program, although the program contains no body. A cliche is used to begin the actual programming.

```
>Implement the program as a successive approximation.
```

Following is the definition of the cliche `successive approximation`:

```
(define-cliche successive-approximation
    (main "successive approximation"
     detail ("Successive approximation.  Finds a value such that"
             test "applied to the value succeeds.  If it does not,"
             "a new value is generated from the old by" approximation
             "and the process is repeated.  The first value tried is"
             initial-value "."))
    (initial-value test approximation)
 (prog (result)
       (setq result _initial-value_)
    lp (cond ((_test_ result) (return result)))
       (setq result (_approximation_ result))
       (go lp)))
```

In a cliche definition, items surrounded by _, for example, `_initial-value_`, represent roles. The list at the beginning of the cliche definition is the list of explanation handlers for the cliche. The `main` handler is used in producing extrinsic descriptions, and the detail handler produces the implementation descriptions. Handlers may also be provided for roles and parts. The default in this case is the name of the role or part with the hyphens removed. These handlers are used for contextual explanations.

If the explanation system were applied to the above cliche, the following procedure would be followed:

The detail handler is found. It is a list, so each element of the list is explained. The first element is a string, and simply returns itself:

```
"Successive approximation.  Finds a value such that"
```

The next element is test. There is no object filling the test part in the program (cliche), so an extrinsic explanation cannot be produced. A contextual handler is tried instead. There is none provided, so test is used. There is only one test, and no test is specified, so the article "a" is chosen. This is concatenated onto the string being produced:

```
"Successive approximation.  Finds a value such that a test"
```

The next two items are strings, and are concatenated onto the result:

```
"Successive approximation.  Finds a value such that a test applied to
the value succeeds.  If it does not, a new value is generated from the
old by"
```

As in the case with test, the value "an approximation" is found for approximation:

```
"Successive approximation.  Finds a value such that a test applied to
the value succeeds.  If it does not, a new value is generated from the
old by an approximation"
```

The remainder of the explanation process follows the same procedure, finally producing:

```
"Successive approximation.  Finds a value such that a test applied to
the value succeeds.  If it does not, a new value is generated from the
old by an approximation and the process is repeated.  The first value
tried is an initial value."
```

The programmer types:

```
>Implement the approximation as an average of 'RESULT
and '(// NUM RESULT).
```

Average is another cliche. Its definition is:

```
(define-cliche average
    (main ("average of" arg)
     detail ("Averages" arg "by summing them and dividing by 2.")
     arg "thing to be averaged")
    (arg arg)
  (// (+ _arg_ _arg_) 2.))
```

If this cliche were explained, the arg would be treated differently than the roles in successive approximation. Since there are two arg parts used in the cliche, an explanation would be formed for each of them. In this case, arg also has a handler, so the default, "arg", is not used in this case. The result would be:

```
"Averages a thing to be averaged and a thing to be averaged by
summing them and dividing by 2."
```

Unfortunately, the system does not make the aesthetic conversion to "Averages two things to be averaged by summing them and dividing by two".

When it was used in the program, the two roles were filled. If this use were explained, the process would be as follows:

First, the initial string goes into the description:

```
"Averages"
```

Then, all of the arg roles are found. They are filled, so an extrinsic description is made for each of them. The first is the RESULT. This is a variable, so it's name is used. The second is a function call. Since a function is not further analyzable, a default description is used, producing "a division of NUM and RESULT" in a rather obvious way. These are concatenated into a noun phrase "RESULT and a division of NUM and RESULT" and this is stuck into the arg position. The final string is then appended, producing:

```
"Averages RESULT and a division of NUM and RESULT by summing them and
dividing by 2."
```

The successive approximation cliche use now also has the approximation role filled, and an explanation of it would now proceed in a slightly different manner. The existence of the filled role means that an extrinsic description of it may be available. In this case, it is, and produces "an average of RESULT and a division of NUM and RESULT", which makes the entire description:

```
"Successive approximation.  Finds a value such that a test applied to
the value succeeds.  If it does not, a new value is generated from the
old by an average of RESULT and a division of NUM and RESULT and the
process is repeated.  The first value tried is an initial value."
```

Next, the programmer types:

```
>Implement the test as an equality within epsilon
of '(* RESULT RESULT) and 'NUM .
```

The definition of the `equality-within-epsilon` test is:

```
(define-cliche equality-within-epsilon
    (main ("equality within" epsilon "test")
     detail ("Succeeds when" arg "are within" epsilon "of each other.")
     arg "value being tested"
     epsilon "specified epsilon")
    (arg arg epsilon)
  (< (abs (- _arg_ _arg_)) _epsilon_))
```

Its description would be:

```
"Succeeds when a value being tested and a value being tested are within
the specified epsilon of each other."
```

As used in the program, the args are filled, while the epsilon is not. Thus, its description would be:

```
"Succeeds when the multiplication of RESULT and RESULT and NUM are
within the value epsilon of each other."
```

Continuing the program:

```
>Implement the epsilon as '0.0001 .
```

The description would now be:

```
"Succeeds when the multiplication of RESULT and RESULT and NUM are
within 0.0001 of each other."
```

The final program's explanation would be:

```
"Successive approximation.  Finds a value such that an equality within
0.0001 test applied to the value succeeds.  If it does not, a new value
is generated from the old by an average of RESULT and a division of NUM
and RESULT and the process is repeated.  The first value tried is an
initial value."
```

6.2 MEMQ and cliches

As a second example, consider a program for implementing the `MEMQ` function in Lisp. This may be implemented as a search through successive sublists of a list for an element whose car is eq to the item being searched for. When implemented with the KBE, this program would take the following form:

```
>Define a program MYMEMQ with a parameter ITEM.
>Add a parameter SEQUENCE.
>Implement the program as a search.
```

The definition of the `search` cliche is:

```
(define-cliche search
    (main "search"
     detail ("Enumerates" arg-of-enumerator "by" enumerator
             "in search of an element which satisfies" test ". "
             "If found," action ".  Otherwise, nil is returned.")
     arg-of-enumerator "thing being searched in"
     action "action taken if the test succeeds")
    (enumerator test action)
```

```
(prog (aggregate)
      (setq aggregate _arg-of-enumerator_)
   lp (cond ((_empty-test-of-enumerator_ aggregate) (return nil)))
      (cond ((_test_ aggregate) (return (_action_))))
      (setq aggregate (_step-of-enumerator_ aggregate))
      (go lp)))
```

An explanation of this cliche is relatively straight forward, using the same procedure as in earlier examples:

```
Enumerates a thing being searched in by an enumerator in search of an
element which satisfies a test.  If found, an action.  Otherwise, nil
is returned.
```

The programmer continues:

```
>Implement the enumerator as a sublist enumeration of 'SEQUENCE.
```

The definition of `sublist-enumeration` is:

```
(define-cliche sublist-enumeration
    (main ("enumeration of the successive CDRs of" list)
     detail ("Enumerates the successive CDRs of" list
             "by taking successive CDRs of it until the"
             "list is empty.")
    (list item)
  (prog (list)
        (setq list _list_)
     lp (cond ((null list) (return)))
        (setq _item_ list)
        (setq list (cdr list))
        (go lp)))
```

A description for this cliche would be:

```
"Enumerates the successive CDRs of a list by taking successive CDRs of
it until the list is empty."
```

As used in the program, the list role has been filled, so the description would be:

```
"Enumerates the successive CDRs of SEQUENCE by taking successive CDRs
of it until the list is empty."
```

The description of the program would now be:

```
Enumerates a thing being searched in by an enumeration of the
successive CDRs of SEQUENCE in search of an element which satisfies a
test.  If found, an action.  Otherwise, nil is returned.
```

The programmer continues:

```
>Implement the test as '(EQ (CAR SEQUENCE) ITEM).
```

The description again follows the same procedure. Upon reaching the test, the entered lisp expression must be described. This begins as:

```
"EQ of ... and ..."
```

The first thing is another funcall, and is described as:

```
"Car of SEQUENCE"
```

The second is a variable, and its name is its description. The resulting phrase for the test is thus:

```
"EQ of the CAR of SEQUENCE and ITEM."
```

Though this will generate a description for arbitrarily complex, the simple algorithm will not generate good descriptions. This is because the code contains none of the structure which the PA provides with cliches. This structure is extremely important in the generation of descriptions, since it provides the logical breakdown of the program, a task which the explanation system does not attempt to do, since it is done by other parts of the PA.

The overall description of the program is now:

```
Enumerates a thing being searched in by an enumeration of the
successive CDRs of SEQUENCE in search of an element which satisfies an
EQ of the CAR of SEQUENCE and ITEM.  If found, an action.  Otherwise,
nil is returned.
```

The programmer finishes the program:

```
>Implement the action as 'SEQUENCE.
```

The final description is:

```
Enumerates a thing being searched in by an enumeration of the
successive CDRs of SEQUENCE in search of an element which satisfies an
EQ of the CAR of SEQUENCE and ITEM.  If found, SEQUENCE.  Otherwise,
nil is returned.
```

7. Conclusion

When something has been broken up into what people consider to be logical units, it is fairly simple to describe it. The PA provides the mechanism to break programs up into logical units, since it needs them in this form to manipulate them (logical units have many uses), and, thus, the PA provides a representation for programs that is fairly easy to generate descriptions from.

Future versions of the PA will also include cliches for data structures and other programming objects. Few changes will probably be needed in the descriptive process to allow for these improvements, since the basic idea of what a cliche is remains the same.

One problem with the system is the need for phrases to be associated with each cliche. An improved, more complex, system may want to generate these descriptions from the cliche itself. This is probably a rather difficult problem, since it is difficult to even think of the phrases "by hand". Such a system would have to be able to do the entire job of generating phrases, since supplying partial phrases makes the phrases very hard to understand in the cliche itself.

8. Bibliography

[1] C. Frank, "A Step Towards Automatic Documentation", MIT/AI/WP-213, December, 1980.

[2] C. Rich, "Inspection Methods in Programming", MIT/AI/TR-604, (Ph.D. thesis), December, 1980.

[3] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 1, January 1982.