Abstract—This paper reports on the initial design and partial implementation of an interactive programming environment to be used by expert programmers. The system is based on three forms of program description: 1) definition of structured data objects, their parts, properties, and relations between them, 2) input-output specification of the behavior of program segments, and 3) a hierarchical representation of the internal structure of programs (plans). The plan representation is of major theoretical interest because it includes not only data flow and control flow relationships between subsegments of a program, but also goal-subgoal, prerequisite, and other logical dependencies between the specifications of the subsegments. Plans are utilized both for describing particular programs and in the compilation of a knowledge base of more abstract knowledge about programming, such as the concept of a loop and various specializations, such as enumeration loops and search loops. We also describe a deductive system which can verify the correctness of plans involving side effects on complex data with structure sharing.

Index Terms—Computer-aided design, data abstraction, Lisp, perturbation analysis, procedural abstraction, program verification, programer’s apprentice, programming methodology, side effects, software design, software development, software engineering.

INTRODUCTION

The first computers had limited computing ability and were very difficult to program. Since that time, hardware improvements have increased the computational power of the typical computer by several orders of magnitude. Some of this additional computing power has been used to make computers easier to program by developing assemblers, compilers, operating systems, and so on. Despite these tools, modern day computer programming seems to have encountered a complexity barrier. This complexity is not simply due to the size of programs, but also to the fact that as size increases, the number of interactions between modules grows much more quickly. This difficulty is felt particularly strongly in artificial intelligence research, where present day programs are already too large to be improved upon in their present form, and yet fall far short of the levels of performance to which the field aspires.

Many avenues are currently being investigated in the search for ways to overcome the current crisis in software engineering. Some seek to bring the experience and techniques of formal mathematics to bear on the problem. For example, Floyd [7] and Hoare [11]–[13] started a major branch of program verification research whose goal is to develop formal logical systems in which desired properties of a program can be proven as theorems. Others have followed past example by seeking to use the computer itself to help reduce the difficulty of programming. The design of new programming languages and language processors is currently a very active field, including Hewitt et al. [10], Liskov et al. [19], and Wulf et al. [28], to name a few. Useful code manipulation and bookkeeping tools, such as editors, indexers, and spelling correctors have also been implemented for existing languages. In the best cases, these tools have been integrated into coherent programming environments, as for example Interlisp [25] or the Programmer’s Workbench [6]. The ultimate form of using the computer itself to solve the software problem is automatic programming. The goal of this research is to create a system which will automatically generate correct and reasonably efficient code which satisfies given high-level, application-oriented specifications. Unfortunately it does not appear that this goal will be attainable in the near future [1].

Our approach to the software problem lies between language-oriented programming tools on the one hand, and automatic programming on the other. Language-oriented tools have essentially reached the limit of their potential without the addition of a major new kind of knowledge about the subject programs. Part of this knowledge is the same as required for automatic programming, i.e., an understanding of the application domain and the way that parts of a program, which exist in the abstract world of the computer memory, relate to concrete objects and operations in the real world. Another part of this knowledge is to a great extent independent of the application domain. This consists of the basic algorithms and data structuring techniques of programming such as those compiled in Knuth [18].

Given knowledge of basic programming technique and the ability to assimilate application domain concepts, it becomes possible for a computer system to understand a user’s program in a much deeper sense and therefore to cooperate with him more effectively in the design, implementation, and maintenance of the program. We call such a system a programmer’s apprentice. An apprentice need not be capable of programming by itself, but can aid the expert programmer by checking his work in various ways. We see this as a realistic interim solution to current software problems and as an evolutionary path towards the more ambitious goals of automatic programming. Similar solutions have been suggested by Floyd [8],
Winograd [27] and Hewitt and Smith [29], although there are methodological distinctions between their proposals and our work.

Work is in progress on this project in the form of three Ph.D. dissertations [21], [23], [26]. This paper reports on the theoretical framework we have worked out to support the construction of a programmer’s apprentice. A longer and more detailed progress report is available in [22]. The primary value of this framework thus far has been to identify the subproblems which are being pursued in the three dissertations. Each dissertation also includes an experimental implementation of one important component of the programmer’s apprentice. This phase of research will soon be followed by a unified implementation to demonstrate the feasibility of the kind of integrated system we describe here.

A Scenario

To explain our concept of a programmer’s apprentice in more detail, we first present an imagined scenario which traces the design, coding, and later modification of a hash table deletion program. The only major feature of the following scenario which is not currently part of our research goals is the use of free English dialogue. Although we feel certain that the system we are designing will be able to support a sophisticated natural language “front-end,” we have made no efforts in this direction.

[The scenario starts at a point when many design choices have been documented with the apprentice. In particular, the input-output specifications of lookup, insert, and delete have been given and the basic data structures have been chosen. The hash table has been implemented as an array of linked lists. Each list is called a “bucket.” Each entry in the table is a dotted-pair whose car is the key part.]

Prgmr: The plan for the delete routine has three main steps: first hash the input key to get an index; then fetch the bucket corresponding to that index from the table; and finally splice out all entries in the bucket whose key part is equal to the input key.

[The apprentice uses the programmer’s outline of the design to build an abstract representation of the program’s intended structure, called a “plan.” Notice the programmer assumes that the apprentice knows many basic programming techniques such as “splicing out” (deleting elements from a linked list).]

Apprentice: I’m sorry, but I can’t verify that this plan satisfies the specifications for delete. In particular, when the key of the first entry in a bucket is equal to the input key, I can’t prove that all entries with keys equal to the input key are not members of the table after the delete. In such cases the bucket after splicing out entries is not identical to the bucket before splicing out entries.

[The apprentice has symbolically evaluated the plan and discovered an error. In general, it is the responsibility of the programmer to supply corrections.]

Prgmr: OK, then modify the plan as follows. Compare the bucket after splicing with the bucket before splicing. If they are not identical, store the new bucket in the table under the index.

Apprentice: That seems to fix the bug.

Prgmr: Here’s the code.

(DEFINE DELETE (KEY)
 ; ; hash table delete routine.
(PROG (BKT1 BKT2)
 (SETQ BKT1 (TBL (HASH KEY)))
 (SETQ BKT2 (BKT-DELETE KEY BKT1))
 (COND ((NOT (EQ BKT1 BKT2))
 ; ; special case when first entry in bucket deleted.
 (STORE (TBL (HASH KEY)) BKT1)))))

Apprentice: I’m sorry but this code is not consistent with your plan. In the special case you are storing the bucket before splicing, rather than the bucket after splicing. Did you mean (STORE (TBL (HASH KEY)) BKT2)?

Prgmr: Yes, fix the code. Also, you can code BKT-DELETE as the standard list deletion algorithm.

Apprentice: Here it is.

(DEFINE BKT-DELETE (KEY BUCKET)
 ; ; list deletion routine coded by apprentice.
 (PROG (CURRENT PREVIOUS)
 (SETQ PREVIOUS NIL) ; ; initialize trailing ptr.
 (SETQ CURRENT BUCKET) ; ; initialize ptr to current entry.
 LP (COND ((NULL CURRENT)
 (RETURN BUCKET))
 ((EQUAL (CAAR CURRENT) KEY)
 ; ; splice out entry.
 (COND (PREVIOUS
 ; ; special case when first element deleted.
 (RPLACD PREVIOUS (CDR CURRENT)))
 (T (SETQ BUCKET (CDR BUCKET))))
 (SETQ CURRENT (CDR CURRENT))
 (GO LP))
 (T (SETQ CURRENT)
 (SETQ CURRENT (CDR CURRENT))
 (GO LP))))

[This is the implementation of a standard algorithm. Notice that the apprentice provides comments on generated code and attempts to use mnemonic variable names from the programmer’s design. In a cooperative programming environment, code written by the programmer and by the apprentice may be freely mixed; in fact each may edit the other’s work.]

[.. Some time later it is necessary to make a modification of the original design. The apprentice helps the (perhaps new) programmer maintain consistency.]

Prgmr: I want to make a change in the representation of buckets in the hashing programs. A bucket now has two parts: a count field and a list of entries as before. The count field will contain an integer equal to the length of the list of entries.
Let a bucket be a dotted-pair whose CAR is the count and whose CDR is the list of entries.

**Apprentice:** The proposed change will affect LOOKUP, INSERT, and DELETE.

**Prgrm:** What are the problems in DELETE?

**Apprentice:** After fetching the bucket you test the key of its first element. But in this new implementation, the CAR of a bucket is a number not an entry. Secondly, the input to BKT-DELETE is a bucket, but it requires a list of entries.

**Prgrm:** Explain the plan for BKT-DELETE please.

**Apprentice:** BKT-DELETE is implemented as the standard list deletion plan. Inputs are a key and a list of entries. The plan is a search loop using two pointers: a pointer to the current entry which is initialized to the input list, and a trailing pointer which is initialized to NIL. On each iteration it tests the key of the first element of the current list. If it is equal to the input key, it splices the current element out of the list by REPLACING the trailing pointer.

**Prgrm:** Change the specifications of this BKT-DELETE to expect as input a bucket of the new type, rather than a list of entries. Also change the initialization of the pointers to read:

```
(setq previous bucket)
(setq current (cdr bucket))
```

**Apprentice:** OK. Now the special case tests in BKT-DELETE and DELETE are not necessary since PREVIOUS is never NIL and therefore elements are always deleted by side effect.

**Prgrm:** Remove the unnecessary tests.

**Apprentice:** OK. By the way, there is currently no code to maintain the count field.

[... and so the session continues with the apprentice looking over the programmer's shoulder.]

**Overview of the Theory**

The major theoretical problem in building a programmer's apprentice is to develop a computer representation of programs and knowledge about programs which can facilitate smooth and natural interaction between the programmer and the apprentice. This kind of problem falls within the paradigm of artificial intelligence research.

Our system is based on three forms of program description:
1) definition of structured data objects, their parts, properties, and the relations between them, 2) input-output specifications of program behavior, and 3) a hierarchical representation of the internal structure of programs, called plans. The major new part of this work is the plan representation, which includes not only the data flow and control flow relationships between parts of a program, but also goal-subgoal, prerequisite, and other logical dependencies. Plans are utilized in the apprentice both for describing particular programs, and for the compilation of a knowledge base (plan library) of general knowledge about programming, such as the concept of a loop and its various specializations such as enumeration loops and search loops.

An important observation to be made from the scenario is that there are two different levels at which the apprentice needs to understand and describe the structure of a program. First, there is what we call the surface plan, a description of the control flow and data flow between the parts of the program. This information is explicitly stated in the program code. However, the apprentice also needs an understanding of the logical structure of the program, called the deep plan, which explains how and why the program works. The deep plan is not explicit in the code, but sometimes shows up in comments such as “special case when first element deleted.”

In the surface plan for a program, each operation or data structure is described by its universally true (or intrinsic) specifications. For example, the Lisp function CAR always returns the left half of the dotted-pair which is its argument. However, different instances of the CAR function may be used for different purposes. Deep plans assign a purpose (or extrinsic) description to each part of a program. For example, one instance of CAR in the scenario is used to extract the count field from a bucket; another is used to extract the key part of an entry. A major deficiency of current approaches to program description, such as the Floyd-Hoare method [7], [11]-[13], has been to concentrate on surface plans and intrinsic specifications because these are easily accessible from the code. Languages such as Simula [4], CLU [19], and Alphard [28] can make up for this deficiency to some extent by raising the abstraction level of the programming language so that purposes are more obvious in the source code.

Hierarchy is another major tool employed by programmers to help understand programs. Depending on the task at hand, the units of description (segments) can be very large or small. The plan for a program is hierarchical, so that a large segment can be described in more detail if necessary by expanding its internal plan. For example, at a high level of description the entire DELETE function is a single segment. Its internal plan has three subsegments: hashing, fetching the bucket, and list deletion. List deletion as a segment also has an internal plan, and so on. However our use of hierarchy is much less rigid than what is advocated by proponents of top-down programming or the virtual-machine method [3]. In the engineering of real-world programs, it is common for strict hierarchy to be violated, usually for the sake of efficiency. For example, the internal plans for two segments may overlap so that a single subsegment has two different purposes, one in each plan.

Closely related to hierarchy is the notion of abstraction. The apprentice's knowledge base of programming expertise must be expressed at a sufficient level of generality to be applicable to many different programs. Our representation system employs data abstractions of a type similar to Simula, CLU, or Alphard in which a data type is defined by its operations and properties. In the apprentice, however, we have combined this data abstraction technique with our plan representation, which abstracts the logical structure of procedures.

**Overview of the System**

Fig. 1 shows and names the main knowledge structures (denoted by rectangles) and processes (denoted by ovals) in...
our system. The left side of the figure shows the kinds of program description that are most directly involved in coding activity. The right side shows the deeper knowledge used in program design.

At the leftmost of the figure is a box labeled to indicate that Lisp code (with comments) is a main user input to the system. We feel it is important to have at the ground level actual code that can be run by a standard Lisp interpreter. The first level of abstraction above raw Lisp code is the surface plan, which is obtained from Lisp code by surface flow analysis. The surface flow analyzer is the only programming language dependent component in the system. Given Lisp code for a program, it generates a surface plan which represents the program’s control flow and data flow in a more convenient and abstract form.

On the design side of the diagram are the deep plans for particular programs under construction and the programming knowledge base, which contains very general deep plans and knowledge about how to verify them. More is said about these later and in [22].

For the apprentice to understand a particular program means that it has connected three levels of description of the program: code, surface plan, and deep plan. One way this can happen is for the programmer, as in the scenario, to initially specify and have the apprentice verify a deep plan. An important and novel feature of this approach is that the deductive system operates on plans rather than directly on Lisp code. This achieves a useful factorization of the verification problem. When the programmer then writes the actual code, the apprentice checks to see that it is consistent with the programmer’s intentions by first generating the surface plan and then recognizing the correspondence between segments in the surface plan and segments in the deep plan. The recognition component must be able to use general programming knowledge and specific programmer-supplied commentary to aid in establishing a plausible correspondence. The net result of this interaction with the apprentice is a complete description of the program at both the surface and deep levels, which will support explanation and aid in program maintenance.

Initial implementations of the deductive system and of the Lisp flow analysis component have been completed; the recognition component is under development. Our current implementation defines and operates on most of the important knowledge representations required to support the introductory scenario. However it is fragmentary and has none of the interactive facilities which would make it usable by a real programmer.

### The Scenario Revisited

We return now to the introductory scenario to describe the operation of our system in more detail. We will introduce the various representations and forms of processing we have developed as they would be invoked in the hash table example. The reader is referred to [22] for more details than given here.

[The scenario starts at a point when many design choices have already been documented with the apprentice. In particular, the input-output specifications of lookup, insert, and delete have been given . . .]

The apprentice’s basic unit of behavioral description is a segment. Data objects flow into a segment and new or side-effected data objects flow out. The point of time just before the segment executes is called its input situation; the point of time just after is its output situation. A segment is defined by its specs, which are a formal statement of input expectations (conditions on or relationships between input objects that are expected to hold in the input situation) and output assertions (conditions that will hold on the input and output objects in the output situation). In terms of code, a segment may correspond to a function definition, the body of a conditional, or several lines of open code. The degree of aggregation is flexible, allowing the programmer and the apprentice to work at the level of detail which is most convenient at the time. The specifications of the hash table deletion segment are as follows.

```lisp
(defspecs delete-segment
  (inputs: key1 table1)
  (outputs: table2)
  (expect: (hashtable table1)
            (hashkey key1))
  (assert: (hashtable table2)
           (side-effected table1)
           (id table1 table2)
           (forall (member table1 =entry)
                    (if (keypart =entry key1)
                        then (not (member table2 =entry))
                        else (member table2 =entry))))
)
```

These specifications state that delete-segment takes as inputs a hash key and a hash table. The net result of this segment (its output) is an updated hash table containing all the same entries as the input table except for those which have the input key. The side-effected clause above signals that table1 has been modified to produce the desired results, while the id clause states that table1 and table2 are two names which refer to the same object, the hash table, in different situations.

We make the convention that clauses which mention an output object refer to the output situation of the segment, while clauses which mention only input objects refer to the input situation. Since table1 is an input name while table2 is an
output name, we can conveniently talk about the state of the table before deletion by using the name TABLE1 and to the state of the table after deletion by using the name TABLE2. We may also state relationships between these two states of the table by using quantified statements in which certain of the clauses mention only input objects and certain mention output objects. Thus the first quantified statement in the above specs is equivalent to the following statement in which the situations have been made explicit.

\[
\text{(FORALL (member table1 = entry) IN *input-situation*)}
\text{IF (keypart = entry key1) IN *input-situation*}
\text{THEN (not (member table1 = entry)) IN *output-situation*}
\text{ELSE (member table1 = entry) IN *output-situation*)}
\]

\*INPUT-SITUATION* and \*OUTPUT-SITUATION* are special variables which, when reasoning about this segment, are bound to the appropriate input and output situations. Our system will accept specs written in this more explicit manner, although we have found the earlier abbreviated form more convenient. Notice that this quantified statement represents a side-effect to the hash table by specifying which facts of the input situation remain true in the output situation and which are to be updated.

...the basic data structures have been chosen. The hash table has been implemented as an array of linked lists. Each list is called a “bucket.” Each entry in the table is a dotted-pair whose CAR is the key part.

We want the apprentice to use a description of data structures which is close to the kind of explanations commonly given by programmers as above. One of the most common data structure notions is that there are object types which are characterized by their decomposition into parts and the relations that hold between these parts. Table I shows how we have taken this approach to describing the structure that is common to all hash tables.

The first four statements in Table I define four object types: HASHTABLE, HASHBUCKET, ENTRY, and HASHKEY. The next statement defines a part relationship and a type restriction on that part: ENTRY’s have a part called the KEYPART which is of type HASHKEY.

Some objects, for example arrays, have many parts which are identified by a numerical index rather than distinctive names such as KEYPART. This is represented by an INDEXED-PART statement. Table I defines hash tables to have an indexed part called BUCKETPART which is restricted to be of type HASHBUCKET.

Table I also defines certain properties, relations, and functions which are relevant to hash tables. A hash table has a SIZE, which is a natural number. There is a MEMBER relation between entries and the hash table, and between entries and hash buckets. Finally there is a HASH function which, given a hash key and a hash table, computes an index.

Notice that PROPERTY, RELATION and FUNCTION statements specify only the name of a particular relationship and restrictions on the types of the arguments. Many properties and relations between objects can be further defined in terms of the objects’ internal part structure, and are thus subject to change if a part is changed. For example, changes to the buckets of a table affect what is a member of the table. Our current system can reason about such side-effects, as will be seen later in the paper. The information required to do this kind of reasoning is expressed in RELATION-DEFINITION’S and PROPERTY-DEFINITION’S.

The RELATION-DEFINITION in Table I states that an entry is a member of the hash table if and only if it is a member of the bucket indexed by hashing the key part of the entry. The square brackets in the definition denote functional terms. Thus if (KEYPART ENTRY1 KEY1) is a predicate which means it is true that KEY1 is the key part of ENTRY1, then [KEYPART ENTRY1] is read the object which is the key part of ENTRY1.

The way the programmer has chosen to implement the
TABLE II

DESCRIPTION OF HASH TABLE IMPLEMENTATION

(IMPLEMENTATION-PART (implementing-array hashtable array))

(IMPLEMENTATION-DEFINITION
  (bucketpart hashtable index hashbucket) <=>
  (item [implementing-array hashtable] index hashbucket))

(IMPLEMENTATION-DEFINITION
  (size hashtable natural-number) <=>
  (upperbound [implementing-array hashtable] natural-number))

(IMPLEMENTATION-PART (implementing-list hashbucket list))

(IMPLEMENTATION-DEFINITION
  (member hashbucket entry) <=>
  (member [implementing-list hashbucket] entry))

(IMPLEMENTATION-PART (implementing-pair entry dotted-pair))

(IMPLEMENTATION-DEFINITION
  (keypart entry key) <=>
  (car [implementing-pair entry key])

The plan for the delete routine has three main steps:
first hash the input key to get an index; then fetch the bucket corresponding to that index from the table; and finally splice out all entries in the bucket whose key part is equal to the input key.

We view programs as being constructed of input-output segments connected by control and data flow. This is obviously not the only possible way to think about programs, but it is one which many practicing programmers find intuitive. Because the apprentice is intended to be an interactive system, this naturalness of representation has been an important design criterion throughout. We represent the control and data flow between segments in a plan graphically, as shown in Fig. 2 (the graphical representation is then straightforwardly encoded in an associative data base).

Subsegments in a plan have specifications which come from two sources. The programming knowledge base contains specifications and plans for many standard programming building blocks, such as splicing elements out of a linked list. A programmer can simply make use of these to build up a more complex plan for his particular application. He may also find it convenient to define new types of segments which help him organize his plan. These new specifications can then, if desired, be assimilated into the programming knowledge base so that they are available for use in designing other similar programs.

In this scenario we assume that HASH-SEGMENT, BUCKET-FETCH, and BUCKET-DELETE are common building blocks for hashing programs whose specifications, shown in Table III, have been entered before the scenario begins.

The specifications for BUCKET-DELETE make use of CASES to express conditional behavior. EXPECT and ASSERT clauses which are not nested within any case apply in all situations. Thus in any execution of BUCKET-DELETE a bucket will be returned which contains exactly those entries of the input bucket whose key part is not equal to the input key. Furthermore, this effect is achieved by side-effecting the input bucket. This part of the specs is followed by a case structure with WHEN clauses to specify the conditions of applicability of the particular cases, which are checked to make sure they are mutually exclusive. CASE1 says that when the first entry in the bucket has a different key than the input key, the output bucket will be identical to the input bucket. CASE2 states that this will not be true when the first entry has the same key as the input key. These specifications reflect the behavior of the
standard list deletion plan which deletes internal elements by re-routing the pointer from the previous element, and deletes leading elements by returning a pointer to a place in the list which is immediately behind all such deleted elements.

The Deductive System

[The apprentice symbolically evaluates the plan and discovers an error....]

The apprentice discovers errors using an innovative system for deductive reasoning about program behavior. In contrast to most other such systems [5, 14]-17 it is capable of reasoning about the behavior of programs involving side effects on complex data with structure sharing. (A more recent system by Suzuki [24] also is capable of dealing with such side effects in Pascal programs.) Other innovative features include the use of a situational data base to maintain a representation of intermediate states in a program’s execution and the use of anonymous objects to represent partial knowledge.

The basic action of the deductive system is a form of symbolic evaluation called specs application. The specs of a segment are applied to a set of input objects in a particular input situation resulting in a set of output objects (some of which may be input objects which have been subjected to a side effect), and a new output situation. The process consists of the following three steps.

1) Variables in the inputs clause are bound to objects in the input situation according to the data flow specified in the plan.

2) The expect clauses are verified in the input situation. If the expect’s cannot be satisfied, there is a bug in the plan.

3) A new output situation is created in which the assert clauses are asserted, substituting the appropriate input object names and creating new object names for newly referenced output objects. (Objects which are outputs by virtue of the fact that they have been subjected to a side effect continue to use their original names.)

This is similar to the way subroutines are handled by Hantler and King [9].

To verify an entire plan, an initial situation is created in which the expect clauses of the main segment are asserted. The subsegments of the plan are then symbolically evaluated, creating a tree of situations which follow from this initial situation. A tree rather than a simple sequence of situations arises when the specifications of some of the subsegments have cases. After all subsegments have been evaluated, an attempt is made to show that the assert clauses of the main segment hold in the final situation(s). If this final proof is successful then the plan is correct; if there is a bug in the plan then part of the proof will fail. The deductive machinery is structured in such a way that diagnostic messages can be constructed to describe the logical error in terms of the programmer’s plan.

Any segment for which specs are available can be used as a subsegment in a plan. In particular, a segment may be used as one of its own subsegments, forming a recursive program. Since loops may be represented as recursions with a single recursive call, we have no special mechanism for handling loops other than those used for recursive plans. The specification of the recursive segment serves both as a set of overall goals to be achieved and as the “loop invariant” as in the method of subgoal induction [20].

We will now see how these techniques enable the apprentice to detect the bug in the programmer’s initial plan for DELETE-SEGMENT.

First an initial situation is created in which the expect’s of delete-segment are asserted. Anonymous objects a-key and A-TABLE are created to represent its inputs. Anonymous objects are objects whose identity is uncertain. Given two objects at least one of which is anonymous, it is impossible know a priori whether or not they are identical. The situational data base starts out as follows:

   (hashkey a-key)  SITUATION-0
   (hashable a-table)

Following the programmer’s outline of the plan, hash-segment is now applied to A-KEY and A-TABLE. This segment is applicable in this situation since its expect’s are satisfied. A new situation is created to represent the state of knowledge about the data objects after hash-segment has executed. The assert clauses are asserted in this situation, including the creation of another anonymous object, AN-INDEXT, to represent the output of hash-segment.
(hashtable a-table)  SITUATION-0
(hashkey a-key)       SITUATION-0

(hash a-key a-table an-index)  SITUATION-1
(integer an-index)
(> an-index 0)

(not (> an-index a-size))
(size a-table a-size)

Notice that the apprentice has created another anonymous object a-size to represent the referent of [size table1] in the specifications of hash-segment. Had the size of a-table been known in the current (or any previous) situation, the bracketed expression would have been replaced by that object and no new object would have been created.

The apprentice now goes on to apply the specs of bucket-fetch and bucket-delete. The expect clauses of bucket-fetch are easily shown to be satisfied. However, bucket-delete has a case structure which requires special handling.

In symbolically evaluating such segments, the apprentice first attempts to prove all expect clauses which are not in any case. If this is successful, it then considers each case in turn.

If all the when's of a case can be proved, then the case is applicable and no further cases need be considered; the assert clauses of this case are asserted in the output situation. If any when of the current case can be shown to be false, then the case is inapplicable, and no assertions are added.

Finally, if there is a clause which can neither be proved nor disproved then the case has unknown applicability. Two output situations are created, one representing the possibility that the clause is applicable, the other representing the possibility that it is not. case1 and case2 of bucket-delete in this plan are of unknown applicability, and so two output situations, 3a and 3b, are created. Notice that in the snapshot of the situational data base below, we are not showing all assertions present in each situation, but only those we will need to talk about in the following discussion.

(Symbolic execution of the programmer's plan is now complete. This means that the apprentice should be able to prove the assert clauses of delete-segment in each of the final situations 3a and 3b. delete-segment has two quantified statements as output assertions: the first one states that all members of the input table will be members of the output table, except for those with the input key; the second one states that all members of the output table are also members of the input table. We will only demonstrate the proof of the first quantified statement here, starting with situation-3a.

The specific assertion to be proved is obtained from the specs by substituting the appropriate object names and situations. Notice that in specs the identical table in the input and output situations is distinguished by the use of different names (table1 and table2), whereas in the proof we have to be explicit about the situation in which we are describing the object a-table.

(forall (member a-table =entry)  in situation-0
  if (keypart =entry a-key)  in situation-0
  then (not (member a-table =entry))  in situation-3a
  else (member a-table =entry)  in situation-3a)

The deductive system uses only restricted universal quantification. Quantified statements are proved by creating an anonymous object about which nothing is known except that it satisfies the class restriction. Anything that can be shown to be true of this anonymous object must be true of all objects of the class. Thus to start the proof of the above quantified statement an anonymous object an-entry is created and (member a-table an-entry) is asserted in situation-0. The if-then-else part of the quantified statement causes two alternative assumptions to be made about an-entry. First it is assumed (keypart an-entry a-key). From these assumptions, together with facts already in the data base and the definition of membership in a table given earlier, it follows that (member an-entry a-bucket) in situation-2.

The quantified statement in situation-3, which represents one of the effects of bucket-delete, states that every member of a-bucket whose key is a-key in situation-2 will not be a member of a-new-bucket in situation-3. Thus an-entry is not a member of a-new-bucket in situation-3.

Furthermore, in situation-3a, a-new-bucket is assumed identical to a-bucket. It then follows that an-entry is not a member of a-table since it is not a member of the bucket hashed to by its key. Thus the then part of the quantified statement is proved.

The else part of the quantified statement says that an entry whose key is not a-key should remain a member of a-table. This is proved by creating a new situation for hypothetical reasoning in which it is assumed (not (keypart an-entry a-key)). Since the key part of an-entry is now unknown there are two subcases: an-entry may still happen to hash into the same bucket, a-bucket, as entries with a-key; or an-entry may be a member of some other bucket. If an-entry is a member of a-bucket, it follows from the specs of bucket-delete in situation-3 that an-entry is a member of a-new-bucket, and thus is still in the table. On the other hand, if an-entry is not member of a-bucket then it is a member of some other bucket which was not side-effected, so an-entry continues to be a member of that bucket and therefore of a-table.

The apprentice is now satisfied that the plan works correctly if case1 of bucket-delete is the applicable case. It must now consider how the plan would operate if case2 were applicable. The quantified statement to be verified in this case is the same as above except that now the terminal situation is situation-3b.)
The entries in the plan data base provide a logical chain of dependencies between specs clauses, showing how the ASSERT's of one group of subsegments interact to satisfy the EXPECT's of other subsegments or the ASSERT of the enclosing segment. Such logical dependencies are called purpose links: if they justify a subsegment's EXPECT clause they are called prequisite links; if they justify the ASSERT of the enclosing segment they are called achieve links. A deep plan is a pattern of purpose links explaining the logical structure of the program. The deep plan is a crucial representation in the apprentice because it allows the programmer and the apprentice to share an understanding of the purpose of every part of the program.

When the programmer attempts to modify a section of the code, it is the purpose links which explain what other segments will be affected and in what ways. ASSERT clauses which are not connected to any purpose link, for example, reflect aspects of a subsegment's behavior which are irrelevant to the task at hand, and which may be changed easily. In other cases, the purpose links will show what behaviors may be added without affecting the program's behavior.

CODING AND PLAN RECOGNITION

Pgrm: Here's the code...

Apprentice: I'm sorry, but this code is not consistent with your plan...

Eventually a programmer will refine his plan to the point where coding may begin. In order to assist the programmer further, the apprentice must recognize the correspondence between parts of the code and segments in some deep plan. Although the general problem of recognizing the plan of arbitrary code with no prior expectations is extremely difficult, we expect recognition to be quite practical given an interactive environment with strong expectations provided by the design phase and the programmer's comments. Thus discrepancies between the plan and the actual code discovered during recognition may be brought to the programmer's attention as potential bugs.

This raises the question of why have Lisp code at all, given deep plans which describe the programmer's intentions in a much more perspicuous form? There are two reasons for keeping code. The first reason is the serious shortcoming of current specification languages which might be used to describe segments in the deep plan. Specification techniques have not yet been developed which capture all the important design criteria used by practicing programmers. Furthermore, our deep plan representation is intended to be a level of abstraction which ignores many efficiency issues. Until we have a theoretical basis for dealing with space-time tradeoffs and the like, we cannot give the programmer any better way of expressing his efficiency-determined design choices than letting him actually write the crucial code, as long as it is compatible with the deep plan. We feel this is a realistic approach to building a usable programmer's apprentice system in the near future.
en between symbolic interpretation and real interpretation is that it is not possible in general to decide which branch of a conditional to take on the basis of a symbolic value. A symbolic interpreter must split control flow and follow both paths leading to an eventual join.

Program loops are handled in our symbolic interpreter by proceeding forward normally until a jump is made to a point in the code that has already been evaluated. When this occurs, a join in control flow is constructed and a special analysis of the data flow is performed based on the following observation: the only thing wrong with the way the loop body has already evaluated is that potential data flow between the outputs of the loop body and inputs of the loop body have been missed. This is remedied by first grouping the segments in the loop body, which has the effect of calculating the net inputs and outputs of the group of segments, and then adding extra data flow links for any free variable outputs that are also inputs. Notice that this approach to analyzing loops results in a plan that is essentially a recursion, i.e., the outputs of the segment feed back into the inputs.

In the uniform syntax of Lisp, control flow and data flow primitives such as PROG, COND, SETQ, and RETURN appear as function calls. However these special forms do not give rise to segments in the surface plan. These forms are viewed as connective tissue between the segments of code in the program that actually do something (i.e., have I/O specifications). This leads to a surface plan which has almost no hierarchy. There are only two kinds of aggregation that are assumed at this level of analysis: the segments in a LAMBDA body are grouped, and the body of each loop is grouped into a single segment. However, the symbolic interpretation of certain special forms such as COND does leave behind suggestions for likely groupings. Later in the recognition process these suggestions, together with deeper knowledge of programming and plans, are used to impose further structure on the initial flat plan, producing greater correspondence with the more hierarchical deep plan.

The second phase of plan recognition consists mostly of grouping the surface plan and assigning more extrinsic specifications. Grouping is simply the operation of drawing a segment boundary around a number of segments at the same level in the surface plan, thereby creating a new segment, and calculating the net data flow and control flow between the new segment, its subsegments, and other segments now at the same level.

As grouping proceeds, an attempt is made to identify each group with a segment of the deep plan. Identification of a surface plan segment with a deep plan segment is possible only if the data and control flow links surrounding the surface plan segment are consistent with the data flow and purpose links surrounding the deep plan segment. If so, the identification suggests a more extrinsic description of the segment than is apparent from the code alone. If the proposed identification is valid, then this extrinsic specification must be deducible from the intrinsic specifications of the grouped subsegments. If all the segments in the surface plan can be grouped and identified with segments in some deep plan, then the program has been recognized.

Failure to recognize a plan leads to either of two courses of action. Possibly the surface plan can be regrouped to identify in a different way with segments of a deep plan. However, if the program is sufficiently constrained so that no such regrouping is possible, the failure is reported to the programmer as a coding bug, as in the scenario:

**Apprentice**: I'm sorry but this code is not consistent with your plan. In the special case you are storing the bucket before splicing, rather than the bucket after splicing. Did you mean (STORE (TBL (HASH KEY)) BKT2)?

The apprentice has determined that the STORE instruction in the code corresponds to the deep plan segment which inserts the updated bucket into the table. However the data flow link in the surface plan fails to correspond to that of the deep plan; the input to STORE should be the output of BUCKET-DELETE (i.e., the updated bucket), instead it is BUCKET-FETCH's output. The apprentice therefore reports this as a coding bug, using deep plan concepts to frame the explanation.

**A Library of Plans**

In order to be useful, the apprentice must have inherent knowledge of many common programming techniques. Plans are a way of representing not only the structure of particular programs, but also of capturing the structure common to many programs. For example, the search loop plan captures the essential similarity between programs which search arrays, lists, and any other data structures which can be linearly enumerated. This deep plan represents not only what is common between these data structures, but also the typical procedural steps that are used to search them: an initialization, a test for exhaustion of the enumeration, a test on the current element to see if it is the one being searched for, and a bump step to the next element in the enumeration.

The structure of the plan library is a major area of research which we have only begun to attack. However, several important criteria have been established. The library should be structured to capture the significant generalizations among plans. For example, search loops with early exits should be representable as incremental generalizations of the general search loop. It is also crucial that the structure of the plan library allow a smooth interaction between plan specialization and the selection of data structures. Thus once it is decided that the linear structure to be searched is an array, it should be easy to transform the general search loop plan into an array searching plan while maintaining a representation of those parts of the logical structure which still apply.

These features suggest a plan library organized around a hierarchy with the topmost distinctions made on the basis of the loop and recursion structure of the plans, e.g., iterative loops, single recursions, double recursions, and so on. Loops, for example, can then be subcategorized into search loops, counting loops, approximation loops, and so on. We are currently investigating how to make these ideas more precise. A similar library of programming knowledge has been constructed by Barstow [2]. However, his library is implemented as a rule-based system oriented towards program synthesis and is weak in its representation of the logical structure of programs.
Program Maintenance

What has been achieved in the apprentice environment is an important factorization of the software design problem. The programmer and the apprentice first work at the plan level developing a consistent structure of interdependent segment specifications. Only then does coding begin, with the apprentice looking over the shoulder of the programmer to make sure the code correctly implements the plan. Finally, once a program has been completely designed and coded in this manner, the apprentice will have built up a very rich description which forms the basis for an interactive documentation facility. Furthermore, the descriptions built up by the apprentice can help the programmer to maintain a consistent design as new requirements force evolutionary changes to be made.

Prgmr: I want to make a change in the representation of buckets in the hashing programs. A bucket now has two parts: a count field and a list of entries as before. The count field will contain an integer equal to the length of the list of entries. Let a bucket be a dotted-pair whose CAR is the count and whose cdr is the list of entries.

Apprentice: The proposed change will affect LOOKUP, INSERT, and DELETE.

Prgmr: What are the problems in DELETE?

Apprentice: After fetching the bucket you test the key of its first element. But in this new implementation, the CAR of a bucket is a number not an entry. Secondly, the input to BKT-DELETE is a bucket but it requires a list of entries.

In verifying the plans for LOOKUP, INSERT, and DELETE the apprentice recorded the dependence of parts of the proofs on design decisions such as the implementation of buckets as lists of entries. The programmer here is proposing to change this IMPLEMENTATION-DEFINITION. The apprentice therefore checks to see what segments in the plans depend on this definition for their correct operation. It finds, for example, that in the plan for BUCKET-DELETE the key extracting segment has an EXPECT clause requiring that its input be an entry. In the verification this EXPECT depended on the data flow in the plan, the implementation of buckets as lists, and a type restriction requiring every member of a bucket to be an entry. The fact that this purpose link will no longer hold if the IMPLEMENTATION-DEFINITION is changed is reported to the programmer as above.

Prgmr: Explain the plan for BKT-DELETE please.

Apprentice: BKT-DELETE is implemented as the standard list deletion plan. Inputs are a key and a list of entries. The plan is a search loop using two pointers: a pointer to the current entry which is initialized to the input list, and a trailing pointer which is initialized to NIL. On each iteration it tests the key of the first element of the current list. If it is equal to the input key, it splices the current element out of the list by REPLACING the previous pointer.

Prgmr: Change the specifications of this BKT-DELETE to expect as input a bucket of the new type, rather than a list of entries. Also change the initialization of the pointers to read...

The apprentice has recognized the plan for the bkt-delete program as a specialization of the general search loop plan in the plan library. This particular search loop uses two pointers and operates by splicing out the appropriate entries. Thus the plan library provides standard programming concepts which the apprentice uses to give explanations which are easily understood by the programmer, and which the programmer can use to instruct the apprentice. This also demonstrates how the apprentice can help maintain not only the program but its documentation as well.

Further Work

Work is continuing on this project in the form of three Ph.D. dissertations currently in progress. Shrobe [23] is continuing to develop the theory and implementation of the deductive system. Rich [21] is implementing the plan recognition component together with a plan library which will codify many of the basic techniques of non-numerical programming in Lisp. Finally, Waters [26] is taking a plan-based approach to the task of building a system to understand Fortran programs in the IBM Scientific Subroutine Package.

References

[1] R. Balzer, "Automatic Programming," Univ. Southern California/Inf. Sci. Inst., Los Angeles, CA, Tech. Memo, 1973.
[2] D. R. Barstow, "Automatic construction of algorithms and data structures using a knowledge base of programming rules," Stanford Artificial Intelligence Lab. Memo AIM-308, Nov. 1977.
[3] O. J. Dahl, E. Dijkstra, and C. A. R. Hoare, Structured Programming. New York: Academic, 1972.
[4] O. J. Dahl and K. Nygaard, "SIMULA—An ALGOL-based simulation language," Commun. Ass. Comput. Mach., vol. 9, pp. 671-678, Sept. 1966.
[5] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Univ. California, Berkeley, June 1973.
[6] T. A. Dolotta and J. R. Mashey, "An introduction to the programmer's workbench," in Proc. 2nd Int. Conf. Software Eng., San Francisco, CA, Oct. 1976, pp. 164-168.
[7] R. W. Floyd, "Assigning meaning to programs," in Mathematical Aspects of Computer Science, J. T. Schwartz, Ed., vol. 19. Providence, RI: Amer. Math. Soc., 1967, pp. 19-32.
[8] "—, "Toward interactive design of correct programs," in IFIP Proc., 1971.
[9] S. L. Hantler and J. C. King, "An introduction to proving the correctness of programs," ACM Computing Surveys, vol. 8, no. 3, Sept. 1976.
[10] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in Proc. 3rd Int. Joint Conf. Artificial Intelligence, Stanford Univ., Aug. 1973, pp. 235-245.
[11] C. A. R. Hoare, "An axiomatic basis for computer programing," Commun. Ass. Comput. Mach., vol. 12, pp. 567-580, 583, Oct. 1969.
[12] "—, "Proof of a program: Find," Commun. Ass. Comput. Mach., pp. 39-45, Jan. 1971.
[13] "—, "Proof of correctness of data representations," Acta Informatica, vol. 1, no. 4, pp. 271-281, 1972.
[14] S. Igarashi, R. London, and D. Luckham, "Automatic program verification I: A logical basis and its implementation," Stanford Artificial Intelligence Memo 200, May 1973.
[15] J. C. King, "A Program Verifier," Ph.D. dissertation, Carnegie Mellon Univ., 1969.
[16] "—, "Proving programs to be correct," IEEE Trans. Comput., vol. C-20, no. 11, Nov. 1971.
[17] "—, "Symbolic execution and program testing," Commun. Ass. Comput. Mach., vol. 19, no. 7, July 1976.
[18] E. Knuth, The Art of Computer Programming, vols. 1-3. Reading, MA: Addison-Wesley, 1968, 1969, 1973.
[19] B. Liskov et al., "Abstraction mechanisms in CLU," Commun. Ass. Comput. Mach., vol. 20, no. 8, Aug. 1977.
[20] J. H. Morris and B. Wegbreit, "Subgoal induction," Xerox Palo Alto Res. Center, CSL-75-6, July 1976.
Abstract—The use of capabilities to control the access of component programs to resources in an operating system is an attractive means by which to provide a uniform protection mechanism. In this paper, a capability is defined as an abstract encapsulation of the data needed to define access to a protected object. We do not assume that capability checking is necessarily concentrated in a protection kernel, nor that capabilities to different types of objects are all of the same degree of complexity. We explore a language-based capability mechanism in which protection environments are declared by declaration, enforcement protocols are automatically produced by a compiler, and access control policy is clearly placed in the hands of the system designer. The basic mechanism introduced is a program component called a capability manager that is an extension of the monitor concept. It can be used to realize most of the facilities associated with kernel-based capabilities, including preemptive revocation.

Index Terms—Access control, capability, exception handling, manager, monitor, protection, resource allocation, revocation.

I. INTRODUCTION

The invention of the monitor concept [1], [2] and its inclusion in the programming language Concurrent Pascal [3] illustrate the use of a standardized language-based mechanism to achieve synchronization of processes accessing a shared data base, to encapsulate abstract resource types, and to provide more flexible, dynamic access control. An obvious

Capability Managers

RICHARD B. KIEBURTZ, MEMBER, IEEE, AND ABRAHAM SILBERSCHATZ

Charles Rich (S'75–S'78) received the B.A.Sc. degree (honors) in engineering science from the University of Toronto, Toronto, Ont., Canada, in 1973, and the S.M. degree from the Massachusetts Institute of Technology, Cambridge, in 1975, where he completed a joint master's thesis on the programmer's apprentice with H. E. Shrobe.

He is currently at the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, completing his Ph.D. dissertation which is concerned with the representation of programming knowledge and its application to the construction of a plan library for the programmer's apprentice. His other research interests include engineering problem solving, knowledge representation, and natural language understanding. Mr. Rich is a member of the Association for Computing Machinery, SIGART, SIGSOFT, and the IEEE Computer Society. Over the last four years he has been the recipient of a Postgraduate Scholarship from the National Research Council of Canada.

Howard E. Shrobe received the B.A. degree (magna cum laude) in mathematics from Yale University, New Haven, CT, in 1968, and the S.M. and Ph.D. degrees from the Massachusetts Institute of Technology, Cambridge, in 1975 and 1978, respectively.

From 1968 to 1973 he worked as a Systems Engineer for the General Electric Company, and Honeywell Inc., maintaining and designing large interactive systems. In 1973 he entered graduate school at the Massachusetts Institute of Technology where he completed a joint master's thesis on the programmer's apprentice with C. Rich in 1975. Since that time he has been working on deductive reasoning techniques for the programmer's apprentice at the Artificial Intelligence Laboratory, M.I.T. He completed a Ph.D. dissertation on these topics in August, 1978. His other interests include computer-aided design, software engineering, and programming language design.

Dr. Shrobe is a member of Phi Beta Kappa.