

Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures

Howard Elliot Shrobe
Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

Abstract

REASON is the deductive component of a larger program understanding system called the Programmer's Apprentice [1]. REASON's key features are : (1) Its use of the Truth Maintenance System [2] to record and manipulate the justifications for its deductions, (2) A discipline of explicitly recording control information in a form which may be manipulated by the reasoning system itself and (3) Its goal of producing a "common sense" explanation of how a program functions, as opposed to the formal proofs common in verification. One particularly difficult problem which such a system must face is the analysis of program which allow destructive modifications of potentially shared data structures. Our observation is that experienced programmers tend to make simplifying assumptions when analyzing such programs; even if these assumptions are found to be false they can be useful in guiding a more accurate analysis. This paper shows how REASON mimics this behavior of expert programmers.

Introduction

REASON is the deductive component of the programmer's apprentice system, a "program understander" which is not mainly concerned with providing a rigorous proof of a program's correctness, but rather with giving a "common sense" explanation of how the intended behavior of the whole program results from the known (or assumed) behaviors of its parts. REASON is able to analyze the effect of a proposed change to a program, viewing the modification as a perturbation rather than as a whole new program requiring a new proof.

One particularly difficult problem which REASON faces is the analysis of programs which allow destructive modifications to complex and shared data structures. Apparently simple operations can produce non-trivial results; a modification of one part of a structure can change properties of the whole structure. Thus, in reasoning about the results of a particular action, a program analyzer must assess what properties besides those explicitly stated will also change. Suzuki [3] attacked the problem of verifying PASCAL programs which allow such operations; however, the formal approach of his work is not easily adapted to the interactive, and common sense context of the programmer's apprentice.

This research was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract number N00014-75-C-0643, and in part by NSF grant MCS77-04828.

This general problem of knowing what changes and what remains the same has been termed the "frame problem" in [4]. In the case of common sense reasoning, one has to assume that most things don't change unless there is reason to believe otherwise. When such assumptions lead to trouble, a belief revision process is necessary. In the case of program understanding, similar techniques can also be applied. Consider the following procedure which deletes all elements of a list which are also members of a hash table.

```
(defun remove (list-1 hashtable)
  (mapc
    '(lambda (x)
      (and (lookup x hashtable)
           (setq list-1 (delete x list-1))))
    list-1))
```

This program has a bug which few programmers (even experts) spot when examining the code. (The reader might try to figure out what the problem is now before proceeding). Suppose that the hashtable is implemented as an array of lists (the lists are called buckets) as is typically done in LISP. Suppose also that the parameter list-1 is bound to a list which shares some structure with one of the buckets of the table. In this case the above program will have the unintended effect of removing elements from the hashtable as well. This is because DELETE operates by side effect.

Most readers will feel cheated by this example. It doesn't seem fair to bring in the issue of structure sharing out of the blue. Quite so. This illustrates a difference between formal verification and common sense reasoning: In a verification system one must consider the most general case of a program's behavior; this is typically done by considering all input

parameters of the program to be objects which *might* share structure. This forces the verifier to consider all cases of possible identity. However, the very fact that the example above seems like a trick indicates that this is not the way programmers routinely think about their code. Quite the contrary, the *tacit calculus* of expert programmers assumes that, unless otherwise stated, sharing of structure does not take place.

There is a good reason for this tacit calculus; in common sense reasoning and also in engineering disciplines we find that there is a trade off between ease of comprehension and accuracy. Simplifying assumptions are made to reduce the cognitive complexity of a problem to the point where a solution can be developed. In general the simplifying assumptions are chosen so that the solution reached is "close enough". In those rare cases where the solution is not close enough, it is debugged; however, it still plays a guiding role in the formation of the improved solution.

Our observations above indicate that programmers use more than one strategy for analyzing the effect of an action. In the more reckless strategy, one assumes that things are not changed unless there is reason to believe they do. This has the advantage of being right most of the time, requiring less effort, and allowing the programmer to form a "first order" theory of what the code does. In the more careful strategy, one does the opposite, assuming things are effected unless evidence to the contrary exists. This, has the advantage of never allowing a false conclusion to be drawn, but the disadvantage of requiring much greater effort, to the extent that it can prevent one from forming a "first order" understanding.

REASON is designed to conduct both of these forms of reasoning. More significantly, however, it is a dependency based system which can use information gathered during the "common sense" analysis to guide the "formal" analysis. Frequently, all one desires from the formal analysis is the statement that there are "screw cases" in which the behavior inferred during the informal analysis no longer obtains.

The Dependency Based Formalism

REASON is implemented in a variant of the language AMORD [5], a formalism which records both facts and the reasons for believing them. All goal states and the reasons for their existence are recorded within the same formalism. *Justifications* are a crucial form of information in REASON. When an assertion is entered into REASON's data base, it is always accompanied by a justification explaining why the new assertion is believed. To make this convenient, each assertion is assigned a unique "fact-name". For example:

Assertion	System-Supplied-Fact-Name
($< X Y$)	F-1
($< Y Z$)	F-2

The transitivity rule can be applied to F-1 and F-2 and a newly deduced fact entered into the system using the ASSERT function:

```
(Assert '( $< X Z$ ) '(Transitivity F-1 F-2))
```

ASSERT takes two arguments: the new assertion to be added to the data base and the justification for the assertion; a justification is a list whose first element is a justification name and whose remaining elements are the fact-names upon which the new assertion depends.

REASON makes deductions using *rules* which consist of two elements: a *trigger set* and a *body*. The trigger set is a list of patterns each of which has two parts: a fact-name variable and an assertion pattern. The body is a LISP expression which is evaluated in an environment in which the variables of the patterns are bound to the objects which they matched. The following is a fairly typical REASON rule:

```
(rule ((:f (Rest :list-1 :list-2))
      (:g (Member :list-2 :obj-1)))
      (assert '(member :list-1 :obj-1)
              '(List-Membership :f :g)))
```

Variables are indicated by a leading colon (:).

In these triggers, the leading single variable (:f or :g) is the fact-name variable, the remaining part of each trigger (Rest :list-1 :list-2) or (Member :list-2 :obj-1)) is the assertion pattern. A rule is applicable if all its patterns are matched by facts which have currently valid justifications (see below). The body is then executed in the environment of the match.

As each trigger is matched to an assertion, the fact-name variable of that trigger is bound to the fact-name of the matched assertion. This allows the body of the rule to refer to its triggering facts. In particular, assert statements in the body of the rule may include a justification mentioning these facts.

Assertions have one of two statuses in REASON, they are either *in* or *out*. A fact which is *in* is believed to be true. An assertion whose negation is *in* is believed to be false. If both an assertion and its negation are *in* then the data base is contradictory and corrective action is required. If neither the assertion nor its negation is *in*, then the fact is simply unknown.

The meaning of justifications such as the transitivity justification shown above is that whenever F-1 and F-2 are *in* then F-3 should also be *in*. If for some reason either F-1 or F-2 became *out*, then F-3 would lack support and would also become *out*. The *ining* and *outing* of facts is managed by the Truth Maintenance System [2] which ensures that no fact is *in* unless it has well founded support.

It is frequently necessary to *assume* that some fact holds even though no reason exists for believing it. This is often done in hypothetical reasoning as when one proves that A implies B by assuming A and deriving B. One assumes a fact because there is

no apparent reason not to believe it; thus, the assumed fact is justified by making it depend on the *outness* of its negation. This is done by the function ASSUME:

```
(Assume '(Made-of The-Moon Green-Cheese) '(Bill F-23))
```

which states that the system will believe that the moon is made of green cheese as long as it has no reason to believe that the moon is not made of green cheese and as long as it believes fact F-23. The function ASSUME builds a justification which has two parts: A list of assertions upon whose *inness* the fact depends and a list of assertions upon whose *outness* the fact depends. When the above ASSUME form is invoked it creates the assertion:

```
(Not (Made-of The-Moon Green-Cheese)) F-1001
```

and then justifies its argument assertion by stating that it depends on F-1001's being *out* and on F-23's being *in*.

```
F-1002 (Made-of The-Moon Green-Cheese) (Bill (F-23) (F-1001))
```

where the first list in the justification is the list of facts whose *inness* supports F-2 and the second list is the *out* list. Whenever an assertion changes its status, the Truth Maintenance system propagates the change until only assertions with well-founded support remained *in*. It is, therefore, always safe to make an assumption since if this introduces an inconsistency, the TMS will cause the improperly assumed fact and all other facts which depend upon it to go *out*.

Reasoning About Simple Side-effects

A program segment is thought of as forming a *transition* between its input and output *situations*. Situations are representations of the state of affairs which obtain at a particular point of the computation. When appropriate, assertions may be situationally tagged to indicate at what points of the computation they are true. For example, to state that the first element of List-1 is Obj-22 when Segment-41 is entered we would assert:

```
(Input-Situation Segment-41 Situation-1)
((First List-1 Obj-22) Situation-1)
```

A side-effect is indicated as follows:

```
(Side-effect <object> <in-sit> <out-sit> <new-fact>)
```

Which says that the <object> has been subjected to a side-effect during the transition from the <in-sit> to the <out-sit> with the result that <new-fact> is made true in the <out-sit>. The new fact can be any relation (primitive or not). Side-effect processing consists of deducing which properties can safely be moved along the transition from <in-sit> to <out-sit> safely; I refer to this process as *transition analysis*. My general approach of explicitly recording dependencies dictates that REASON should provide a justification whenever it decides to move a fact across a transition. Similarly, if it decides not to move a fact it should justify this decision as well. These recorded dependencies allow REASON to make an initial decision based

on a cursory analysis of the circumstances, while still reserving for itself the option of reconsidering in more detail at a later time. If it should reconsider, the recorded dependencies can rapidly indicate which properties depend on the simplifying assumptions of the cursory analysis.

The basic protocol is as follows: An assertion in the input situation of a transition can be moved to the output situation of the transition only if there is an explicit assertion declaring it *safe* to do so. In phase one of the analysis, simple rules are run to find reasons for not moving a fact. If such cause is found, these rules assert that it is *unsafe* to move the fact. At the end of this process each assertion is asserted to be safe to move; however, the justification for this safe assertion depends on the *outness* of the corresponding unsafe assertion. Thus, if any reason for considering the assertion unsafe had been found, the unsafe assertion will be *in*, causing the safe assertion to be *out*. If at the end of this process a safe assertion is *in* for a particular fact, then the fact will be asserted in the output situation with its justification pointing to the safe assertion. The following rules carry out these operations:

The Default Assumer for the Careful Protocol

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact))
      (:g (:old-fact :in-sit))
      (:h (transition :in-sit :out-sit)))
      (assume '(not (safe :g))
              '(safety-first :f :g :h)))
```

The Default Assumer for the Fast and Dirty Protocol

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact))
      (:g (:old-fact :in-sit))
      (:h (transition :in-sit :out-sit)))
      (assume '(safe :g :in-sit :out-sit)
              '(reckless-abandon :f :g :h)))
```

The Safe Fact Mover

```
(rule ((:f (safe :old-fact :in-sit :out-sit))
      (:old-fact (:fact :in-it))
      (:g (transition :in-sit :out-sit)))
      (Assert '(:fact :out-sit)
              '(safe-from-side-effect :f :g)))
```

Make The Side-effect Appear In The New Situation

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact)))
      (assert '(:new-fact :out-sit)
              '(side-effects-happen :f)))
```

where the last of these merely asserts that the relation stated in the side-effect assertion is true in the output situation; this is, of course, independent of the safe assertions.

REASON has a set of rules for determining whether a particular fact is affected by a particular side effect. If the fact is not affected then the rules assert that the fact is *safe-from* this particular side effect (and conversely). If a fact is safe-from all side effects at a particular transition then it is *safe*. The simplest such rule is the rule of direct negation which says that a fact which is explicitly negated by a side-effect is *unsafe*:

```
(rule ((:f (side-effect :obj :in-sit :out-sit (not :fact)))
      (:g (:fact :in-sit)))
      (Assert '(not (safe-from :f :g))
              '(direct-negation :f :g)))
```

Another simple side-effect rule concerns objects (like records) which are composite structures with several distinct fields, called *parts*. Suppose that we modify a particular record to change one of its parts to a new value. It follows, that assertions stating the old value of the affected part of the record are not safe from this side effect. However assertions mentioning other fields of the structure are safe.

```
(rule
  ((:f1 (side-effect :object-1 :s-1 :s-2
                    (:new-part-name :object-1 :new-part)))
   (:f2 ((:old-part-name :object-2 :old-part) :s-1)))
  (cond
    ((eq :old-part-name :new-part-name))
    (t
     (assert '(safe-from :f1 :f2)
             '(diff-part-side-effect :f1 :f2)))))
```

The next rule is for side-effects to objects such as arrays which have indexed-parts. Suppose we have an object with an indexed structure (for example, an ARRAY, a HASH-TABLE, or a record structure including an ARRAY) and that this object is modified to change the part indexed by INDEX-0. Also suppose that we have an assertion saying what is the part indexed by INDEX-1. Then, the side-effect should make the assertion unsafe if the two indices are identical and leave it unaffected otherwise, as expressed in the following rule:

```
(rule
  ((:f (side-effect :object :in-sit :out-sit
                    (:ip :object :new-index :new-value)))
   (:i ((:ip :object :old-index :old-value) :in-sit)))
  (assert '(if ((equal :old-index :new-index) :input-sit)
            then (not (safe-from :f :i))
            else (safe-from :f :i))
          '(indexed-part-side-effect :f :i)))
```

Notice that it is altogether possible that neither the premise of the IF-THEN-ELSE, nor its negation are present in the data base and thus, that neither a *safe-from* assertion nor its negation will be created. In the first pass, "fast and dirty" analysis, this lack of an unsafe assertion will be taken as grounds for assuming that the assertion is safe; it will, therefore, be moved across the transition. This will not be logically incorrect, since the justifications for the assumption are explicit and can be withdrawn; it is, however, not a very useful strategy. Even for a fast and dirty pass this strategy is a little too dirty. We would rather say that if it is *possible* that the two indices are equal, then we should not consider the assertion *SAFE*, but should rather do a case analysis, considering separately the two possibilities of equality and non-equality.

Since the conclusion that the assertion is not *SAFE* is based on the *possibility* that the indices are equal, we need a way of

asserting that this possibility exists; this *possibility assertion* can then be included in the justification. I have developed some syntactic mechanisms to facilitate the use of the concept. The starting point is the observation that an assertion is possible as long as its negation is *out*; of course, if the assertion is *in* it is also possible. Thus, the following support structure captures the notion of possibility (the wavy line indicates that the dependency is on the *oumess* of the assertion, this is called non-monotonic support).

F-3 (Possible ((Equal Index-0 Index-1) S-1))

F-2 (Not ((Equal Index-0 Index-1) S-1))

F-1 ((Equal Index-0 Index-1) S-1)

As a syntactic convenience, I have added an IF-POSSIBLE-THEN-ELSE construct, which is invoked as follows:

```
(If-possible (:f fact-1)
  then body-1
  else body-2)
```

This expression is interpreted in two stages. First the support structure shown above is created for an assertion which states the possibility of FACT-1. If this assertion stays *in*, then BODY-1 will be executed in a binding environment where :F is bound to the possibility assertion for F-1. If the negation of FACT-1 is *in* (i.e. FACT-1 is impossible) then BODY-2 is executed in a binding environment in which :F is bound to the negation of F-1.

Given this, the appropriate body for the rule dealing with side effects to indexed-part structures is:

```
(if-possible
  (:f (equal :old-index :new-index) :input-sit)
  then (assert '(not (safe-from :f :i))
              '(careful-indexed-part :f))
  else (assert '(safe-from :f :i)
              '(careful-indexed-part :f)))
```

This says that if it is at all possible that the indices are equal, the system will decide that the indexed-part assertion is not *SAFE*. If REASON decides that moving this assertion is important it can try backward chaining rules on the IF-THEN-ELSE assertion to create a case analysis.

More Complicated Effects

So far the analysis of side-effects has been quite simple considering only assertions about PARTS and INDEXED-PARTS. These are the most primitive notions in the system. However, there are a host of more complex defined notions which have been developed to allow programs to be thought of in more high level terms.

The relations which are used in describing programs are complex in the sense that they are a logical combination of assertions which ultimately depend on the PART structure of the objects implementing the more abstract notion. For example, in

hashing systems there is a notion of membership in the table which is defined in terms of membership in one (or more) of the INDEXED-PARTS (BUCKETS) of the table. Similarly, since we frequently implement these BUCKETS as LISTS, membership of an object in the BUCKET reduces to whether the object is the FIRST part or a MEMBER of the REST part of the list. Thus, simple side-effects to the part structure of an object can result in side-effects to derived properties of the object. For example, side-effecting a TABLE to set one BUCKET to the EMPTY-LIST will (potentially) delete several members of the table. The processes handling side-effects, therefore, must examine the way in which facts in the input situation of a transition depend on one another and use this as a guide to the transition analysis.

For example, let us define LIST membership in the standard way: an object is a MEMBER of a LIST if it is either the FIRST of the LIST or a MEMBER of the REST of the LIST.

```
(Member list object) <=> (Or (First List Object)
                             (Member [Rest List] Object))
```

The brackets indicate a *reference expression*; they are read "the object which is ...". REASON extracts the component clauses of the right hand side of this definition, building a network of *potential dependency* assertions, which links assertions to those side-effects which might make them UNSAFE. For example, from the definition for LIST membership we can get the following potential dependency assertions:

```
(potential-dependency (member :list :object-1)
                     (first :list :object-2))
(potential-dependency (member :list :object-1)
                     (rest :list-1 :list-2))
(potential-dependency (member :list :object)
                     (not (member :list-1 :object)))
```

Potential dependency assertions are the information used to determine that there *might* be a logical connection between a fact and a side-effect. These say that if (1) There is an assertion in the input situation which matches the first pattern and (2) There is a side-effect on the transition which matches the second pattern, then it is *possible* that the assertion is rendered UNSAFE by the side-effect. Notice that in the case of dependencies on many-to-many relations (such as MEMBER) the dependency is on the negation of the relation. If functional relationships (such as PART or INDEXED-PART) assertions are involved, a side-effect asserting a new value for the relation, such as (First List Foo), implicitly negates any previous value of the property, such as (First List Bar); for these relations the dependency pattern is not negated. The network of potential dependency assertions is completed by using a transitivity rule.

The information in the potential dependency assertions is used by a rule which monitors transitions in which facts might be made UNSAFE by a side-effect. If such situations are noticed, the rule asserts that the fact is *possibly-unsafe*. Any fact which is POSSIBLY-UNSAFE is expanded, i.e. the right-hand side of its definition is added to the data base.

```
(rule ((:f (potential-dependency :a :b))
      (:g (side-effect :object :s-in :s-out :b))
      (:h (:a :s-in)))
      (Assert '(possible (not (safe-from :g :h)))
              '(pd :f :g :h)))

(Rule ((:h (Possible (not (safe-from :f :g)))))
      (Assert '(Expand :g) '(pd-expand :h)))
```

Two points about these rules for determining potential dependency should be noted. First, these rules only signal the *possibility* that an assertion is affected by a side-effect; it is for other more thorough rules to explore whether or not the assertion actually is SAFE or not. This allows a many layered control structure in which one set of rules notices candidates for examination, and other sets of rules chose to examine these candidates at a level of detail deemed appropriate.

The second point to be made here, is that the potential dependency rules shown so far are actually of the "fast and dirty" variety. Remember that our earlier example showed that different local variable names might, in fact, name the identical object. Usually people rule out this possibility of "aliasing" to facilitate their analysis. However, to be completely accurate one must examine all possibilities.

The careful version of a rule for LIST membership, for example, is:

```
(rule ((:f (side-effect :obj :s-in :s-out (first :obj :new-first)))
      (:g ((member :obj-2 :old-first) :s-in)))
      (If-possible (:h (id :obj :obj-2))
                   then (Assert '(possible (not (safe :g from :h)))
                               '(poss-or-se-careful :f :g :h))))
```

Once it has been determined that an assertion is possibly affected by a side-effect it remains to be determined whether the assertion is SAFE or UNSAFE. When a defined relation is possibly unsafe it is expanded into its definition; thus, the only remaining analysis is that associated with the logical connectives. Rules for these connectives are given in [6].

Once an assertion has been determined to be UNSAFE, it is then treated as a side-effect itself. This initiates a consideration of derived side-effects, propagating the side-effect through the various levels of definition.

Reducing Complexity in Side Effect Analysis

Much of the complexity in reasoning about side effects occurs in recursively defined structures which share substructure. Suppose that we know of the existence of two lists, and one of these is modified; given what we have developed so far, we must consider the possibility that this modification will also change some properties of the second list as well. However, if we knew that the two structures were *disjoint*, then this possibility would be eliminated, reducing the complexity considerably. Burstall [7], introduces some techniques for reasoning about side-effects which use this notion of disjointness to advantage. I will extend that notion in this section.

Suppose that we know that LIST-1 and LIST-2 do not share any structure and that one of these lists has an element spliced into it; since neither list can be a sub-list of the other it is not necessary to conduct a thorough investigation as outlined in the preceding sections. It is immediately obvious that the modification to the first list cannot affect the second.

The notion of list-structure can be generalized to that of recursive data structures; having made this generalization we can create a hierarchy of classification for side-effects, properties and the degree of sharing exhibited by recursive structures. It will follow that the level of sharing can limit the degree to which side-effects to one structure can effect properties of the other. Similarly, the classification of properties into levels isolates those at a higher level from less powerful side-effects.

For our purposes a recursive structure can be thought of as a labelled digraph with two distinct types of arcs: those labelled as value arcs and those labelled as structure arcs. The set of objects connected to a node by outgoing structure arcs are called the *immediate children* of the node; the set of objects reachable by outgoing value arcs are called the value set of the node. All immediate children of any node of a recursive data structure must be either terminal or non-terminals of the recursive data type. Value nodes may be objects of any type. Terminal nodes may not have outgoing structure arcs.

Actually, we are not as interested in sharing as in the lack of it: disjointness. We may identify three types of disjointness which have some utility. We have previously defined *structure sharing* as having a node in common. *Structurally disjoint* structures are those which do not share structure. For lists this means that no sublist (the transitive closure of CDR) of the two lists is shared.

Often we will have non-recursive structures such as HASH-TABLES whose parts are recursive structures. For such objects, we define structure sharing in the obvious way: namely two objects share structure if there is a part of the first and a part of the second which share structure. Thus, a HASH-TABLE and a LIST share structure if one of the table's BUCKETS shares structure with the LIST. They are structurally disjoint if there is no BUCKET of the TABLE which shares structure with the LIST.

The next type of sharing is termed *value sharing*. Recall that the nodes of a recursive structure can have parts other than those which represent the immediate children. A list is a recursive structure which has a value at each node called the FIRST. When there is an object which is a value of two recursive structures, we say that there is *value sharing*; conversely, if there is no such object we say that the structures are *value disjoint*. It follows that if two structures are value disjoint, they are also structurally disjoint.

Two objects are *totally disjoint* if (1) The objects are both structurally and value disjoint and (2) All objects pointed to by each node are totally disjoint. It follows that if two RECURSIVE-STRUCTURES are totally disjoint, then side-effects to the one can not effect the other.

Now let us classify side-effects in a manner similar to that used for structure sharing. We call side-effects *strictly structural* if they only affect the immediate children property of some node of the structure. RPLACD is the simplest such side-effect, although LIST-INSERT and LIST-DELETE, etc. are also strictly structural side-effects. Notice that a strictly structural side-effect to one structure will not affect any property of another object from which it is structurally disjoint.

We may also identify *strictly value* side-effects such as RPLACA which only change value properties. If two structures are structurally disjoint, then a strictly value side-effect to one will not affect properties of the other. A *structural side effect* is one which consists only of strictly structural and strictly value side-effects. A SORT program which works by changing both CAR and CDR pointers in a list is an example of a structural side-effect. If two objects are structurally disjoint, a structural side-effect to one will not change any property of the second.

Finally, an *indirect side effect* is one which only changes properties of VALUES of the RECURSIVE-STRUCTURE. For example, a marking graph traversal procedure which sets the MARK property of the VALUE of each node is such an INDIRECT side-effect procedure. If two objects are VALUE DISJOINT, then INDIRECT side-effects to one will leave properties of the other unchanged.

Finally we come to a classification of properties. We can notice that some properties such as SUBLIST or LENGTH only depend on the recursive structure of the object, and not on the VALUES at each node. We call these *strictly structural properties*. More commonly, properties such as MEMQ, depend both on the recursive structure and on the identity of the various VALUES present at each node, but not on any property or sub-structure of these values; these are called *value dependent* properties. Finally, there are properties which depend both on the structure and on mutable properties of the objects present at each node. MEMBER (as opposed to MEMQ), for example, depends on the structure of the list as well as on the structure of the objects pointed to by the list.

We note that a side-effect at one level can not effect a property of a lower level. For example, a VALUE side effect cannot affect a STRICTLY STRUCTURAL property. An INDIRECT side-effect cannot affect a STRUCTURAL property.

These observations can be summarized by several simple rules of the following form:

```
(rule ((:f1 (Side-effect :obj-1 :s-in :s-out :se))
      (:f2 (Structural-Side-Effect :se))
      (:f3 (Structurally-disjoint :obj-1 :obj-2))
      (:f4 ((:fact :obj-2) :s-in))))
  (Assert (Safe-from :f1 :f4)
    (Structure-disjoint :f1 :f2 :f3 :f4)))
```

```
(rule ((:f1 (Side-effect :obj-1 :s-in :s-out:se))
      (:f2 (Structural-Side-Effect :se))
      (:f3 (Structurally-isolated :obj-2))
      (:f4 ((:fact :obj-2) :s-in)))
      (Assert (Safe-from :f1 :f4)
              (Structure-isolation :f1 :f2 :f3 :f4)))
```

Notice that since these rules assert that a particular property is SAFE, they remove the need to engage in the more complicated analysis shown earlier. A large percentage of side effects have very limited range of effect precisely because there is a strong limit on structure sharing. Thus, most of the time one of the above rules will fire and REASON's work will be done. In some rare cases, the more complex and thorough analysis will be required.

Given these ideas we can extend our fast and dirty protocol for side effect analysis even further. It appears that most programmers assume by default that the inputs to a program segment are structurally disjoint. In our first pass analysis we will assume that all recursive structures are structurally disjoint unless otherwise indicated. This is coded by making rules of the following general form:

```
(rule
  ((:f (side-effect :list-1 :sin :sout :se))
   (:g ((P :list-2) :sin)))
  (if-possible (:h (structurally-disjoint :list-1 :list-2))
    then (assert '(safe-from :f :g))
          '(loose-living :f :g :h))
    else (assert '(possible (not (safe-from :f :g)))
                '(careful :f :g :h))))
```

Sussman [8] argues for the importance of "almost correct theories" and debugging as a means of reducing the complexity of problem solving tasks to levels within human limits. My approach to analyzing side effects follows in this tradition. REASON's first pass analysis is not rigorous, but it is useful. Much of the time it just happens to give the right answers (for incorrect reasons). The dependencies recorded during the first pass analysis can help guide an accurate and complete investigation in a second pass analysis if the problem is considered to be worth the effort. Often, however, the first pass analysis is adequate to allow the apprentice to help analyze the consequences of proposed program modifications and to generate high level common sense explanations of the code.

References

- [1] Rich, C. and Shrobe, H. [1978], "Initial Report on a LISP Programmer's Apprentice", IEEE Trans. on Soft. Eng., V4 #6, November 1978, pp. 456-467.
- [2] Doyle, Jon *Truth Maintenance Systems for Problem Solving*, M.I.T. Artificial Intelligence Laboratory Technical Report 419, January 1978.
- [3] Suzuki, N. "Automatic Verification of Programs with Complex Data Structures", Stanford AIM-279, February 1976.
- [4] McCarthy, J. and Hayes, P. "Some Philosophical Problems from the Standpoint of Artificial Intelligence", Machine Intelligence 4, American Elsevier, N.Y.
- [5] deKleer, J., Doyle, J., Steele, G. & Sussman, G.J. "AMORD: Explicit Control of Reasoning", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, August 1977.
- [6] Shrobe, Howard *Logic and Reasoning for Complex Program Understanding*, PhD Thesis, MIT, August 1978.
- [7] Burstall, R. M. "Some Techniques for Proving Properties of Programs Which Alter Data Structures", Machine Intelligence 7, Edinburgh University Press.
- [8] Sussman, G.J. *A Computational Model of Skill Acquisition*, M.I.T. Department of Mathematics Ph.D. Thesis; M.I.T. Artificial Intelligence Laboratory Technical Report 297, August 1973; *A Computer Model of Skill Acquisition*, New York, American Elsevier 1975.