DOMAIN SPECIFIC DEBUGGING AIDS FOR NOVICE PROGRAMMERS

Joachim Laubsch and Marc Eisenstadt

The Open University
Milton Keynes, ENGLAND*

ABSTRACT

We have been developing a debugging aid tailored to novice programmers learning to use a simple asaertional database language. Based in part on existing programmer's apprentice and debugging projects, our system makes several novel contributions: it is oriented towards a large audience (several hundred per year) of computer-naive users; the system deals with argument passing, recursion, and side-effects. Symbolic evaluation of the user's code produces an effect-description which is compared with an idealised effect-description (derived from a library plan). The library plan uses domain-specific knowledge to allow the user a great deal of freedom to invent alternate approaches to the programming task at hand. Based on effect-description mismatches, a variety of hints for correcting the code can be given to the student.

I INTRODUCTION

This paper describes recent progress in the design and implementation of a debugging assistant for novice programmers. The novices are Open University psychology students learning about problems of modelling and representation using SOLO, a LOGO-like language oriented towards the manipulation of an assertional data base [2]. To give the reader a quick feel for SOLO, here is a short SOLO program and its MacLisp counterpart:

```
SOLO:   TO INFECT /X/
        1 NOTE /X/ HAS FLU
        2 CHECK /X/ KISSES ?Y
         2A If present: INFECT *Y; EXIT
         2B If  absent: PRINT "ADIOS"; EXIT

Lisp:   (defun infect (x &aux y)
          (assert `(,x has flu))
          (cond ((present `(,x kisses ?y))
                 (infect y))
                (t  (print "ADIOS")))
          nil)
```

Fig. 1

* This research is supported by grants from the U.K. Social Science Research Council (HR7372/1) and the German Academic Exchange Service.

Extensive user aids ensure that students enter lexically and syntactically correct programs. Even so, the programs typically fail to perform as intended. While realising that such failures can be of great pedagogical value, our aim is to develop a technique for helping users circumvent hurdles which they might find too tedious to deal with or which may prevent them from attacking pedagogically more valuable tasks. Since our students normally work on their own (via dial-up lines to one of our DECsystem-20's), the provision of intelligent debugging aids is of great practical benefit.

Students write programs on one of several suggested topics. Schema-matching, simple inferencing, and children's subtraction skills are among the favourites. The subtraction task is particularly interesting, because SOLO has no numerical primitives, and thus students must think very carefully about how to represent numbers in a database, what operations to allow, etc. We can provide our debugging assistant with a fair amount of knowledge about subtraction, which makes it a good testbed for our ideas. It is important to note, however, that we do not expect the students to have to work towards implementing a particular algorithm. On the contrary, one of the novel aspects of our approach is that we encourage the students to come up with their own representations and algorithms. The data representation they choose will of course influence the algorithm they use, and vice versa. Our debugging assistant is provided with detailed knowledge of a wide range of possible data structures which could be used to represent subtraction tables, along with an abstract description of effects to be achieved by a subtraction program. Only after examining the student's representation does the assistant decide what the 'idealised effect description' for that student's code should be— i.e. what changes to the assertional data base ought to occur as a result of a successful run of his program.

The debugging assistant deals with parameterised subroutines, conditionals, and side effects, thus extending the pioneering work of Adam and Laurent [1], Goldstein [5], Miller and Goldstein [6], and Ruth [8]. It relies extensively on a language-independent 'plan diagram' representation developed by Rich and Shrobe [7], and Waters [10]. The latter is the basis for computing the effects of programs and program segments, which in turn gives the debugging assistant a way to understand what the program

achieves, and a way to compare this with what the program should have achieved. In the case of small mismatches, it is possible to recommend a particular patch to the student. More importantly, however (since we don't want to write the entire program for the student) is the generation of tailor-made counterexamples to provoke the student into thinking up his or her own solution.

When a student gets stuck, he or she can invoke the debugging assistant by typing HELP DEBUG. The assistant investigates the student's code and data base in several phases, as outlined below:

* TRANSLATION: User programs are translated (as they are entered) into a language independent 'plan diagram* notation (PDN) [7, 9, 10] which encodes the control flow, the data flow, and a declarative description of each program's overall effect. During translation, three kinds of 'irrational code' can be detected (cf. [5]): 0) unreached code; (2) an unbound CHECK variable used following the 'absent' branch of a CHECK statement; (3) useless code, which includes variable binding without later reference, assertion and deletion of the same data base triple, and superfluous CHECKs. The translation phase is described in [3].

* PLAN RECOGNITION: Certain sequences of code (some of which are specific to SOLO, others of which are specific to the subtraction task) can be recognized in their PDN form as manifestations of various 'standard library plans'. This recognition is performed by referring to a plan library, which knows not only how to recognize plans, but also what their effect descriptions are. We have identified over a dozen standard library plans which are manifest in the code of actual SOLO users. A typical plan is 'simple net traversal'. In terms of the plan library of Shrobe et. al. [9] this is a 'generator' plan, which recursively fetches all the nodes along a chain of a particular relation, e.g. nodes n1 , n2 and n3 in the network below:

n1---REL--->n2---REL--->n3

This in turn can be used as part of a more complicated plan, such as 'generate and side-effect each'. This plan is the basis for the INFECT program shown in Fig. 1— INFECT generates a sequence of nodes by simple net traversal (steps 2 and 2A), and unconditionally side-effects every one of them (step 1). The role of such plans is discussed in [4].

* SYMBOLIC EVALUATION: An analysis is made of all possible pathways which the program might have explored, and an effect description is generated, providing a declarative account of what the program achieves in terms of changes to the data base. This is especially important for those pieces of the program which did not appear to match any known plans in the plan library. This phase is described in section II below.

* EFFECT DESCRIPTION MATCHING: The effect description of the student's program i3 now compared with an idealised effect description to find mismatches. The idealised effect description is computed by expanding a domain specific library plan in a top-down fashion to mesh with the student's own chosen data base representation (which we analyse bottom-up). The debugging assistant has criteria for detecting near misses, and heuristics for suggesting how to repair a program. If this fails, the effect descriptions can still provide the basis for generating useful counterexamples which may help the student to focus more clearly on the underlying cause of the bug. Section III discusses effect description matching in some detail, and gives examples of the kinds of bugs which our system can currently deal with.

All of the examples shown below run in our MacLisp implementation, although the debugging assistant is not yet robust enough for use in a 'live' student environment.

II DERIVING EFFECT DESCRIPTIONS BY SYMBOLIC EVALUATION

A. Symbolic evaluation describes a program's effect by stepping through all cases

Symbolic evaluation serves as a tool for deriving a canonical description of the effect that a user program will have, when run in some environment. This description can be compared with the effect description of known (library) plans. Such a comparison will show that the program either behaves as intended by the tutor, deviates "slightly" (in which case we can suggest a patch), or deviates "grossly" (in which case we can derive counter-examples).

SOLO programs typically modify a global data base by side-effects. A procedure receives arguments and implicitly an input data base (db-in), and produces an output data base (db-out). The effect of a program is described by a tree which at its root has db-in and at its leaves has db-out for each possible condition.

The data base consists of a set of triples, each having the form

triple = [<source> <link> <target>]

We represent a data base as an association list of bindings, where each binding is

binding = [bind (<source> <link>) <target>]

The original data base, after the student has used a problem-specific SETUP procedure (and possibly after a series of top-level assertions) can then be represented as

dbO - ([bind (<source1> <link1>) <target1>]...|

By "the binding of (a b) in e", abbreviated "(get a b)", we refer to the target of the first binding whose left part is (a b) in a data base e. Alternative environments are represented as a cases

statement of the form:

```
(cases [<predicate1> <environment1>] ... )
```

The predicate can refer to node identity, e.g. (- (get a b) c)), or to the presence of a binding, e.g. (present <source> <link>). We interpret the latter predicate as true in an environment e if (<source> <link>) is bound in e.

B. Effect descriptions for SOLO primitives

Each primitive step in a plan diagram is an instance of some class which has associated with itself the knowledge of how it affects the data base. Symbolic evaluation produces the following: (1) augmentations to db-in (for FORGET and NOTE steps); (2) a local binding environment; (3) a cases statement (for CHECK steps).

CHECK steps come in two varieties: with an open or a closed variable as the target. A CHECK with an open variable produces a local binding to that variable. The local binding environment is not shown in the effect description since it does not change the data base. Instead, the binding environment is used for finding the substitution of a variable in terms of access to the data base through the procedure's input parameters.

Here is an example taken from a definition used in a student's subtraction program. It looks up the difference between two numbers /A/ and /B/ by consulting some table T (pointed to by the relation 'SUBTBL' emanating from node /A/). The result, R, is stored on the node /ANS/. The student could have chosen to represent subtraction tables in the form of simple relations, e.g. 5——4——>1, 5——3——>2, etc., but has instead opted for a more flexible representation of the form 5—SUBTBL——>5SUB, 5SUB——3—>2. This allows addition and multiplication tables to be stored in the data base as well. The use of indirect accessing of tables gives the procedure its name, 'Diff-ind':

```
TO DIFF-IND /A/ /B/ /ANS/
1 CHECK /A/ SUBTBL ?T
  1A If present: CONTINUE
  1B If absent:  EXIT
2 CHECK *T /B/ ?R
  2A If present: CONTINUE
  2B If absent:  EXIT
3 NOTE /ANS/, VAL *R
```

Fig. 2

For the purposes of symbolic evaluation, here are the key events which occur in the above procedure:

* At step 1A, T is bound to (get /A/ SUBTBL).

* At step 2A, R is bound to (get (get /A/ SUBTBL) B).

* At step 3, db-in is augmented by [bind (/ANS/ VAL) (get (get /A/ SUBTBL) /B/)].

A CHECK-step produces two possible environments, which are represented as alternative branches of a cases statement. This yields the following effect description for DIFF-IND in environment e:

```
1 diff-ind(/a/ /b/ /ans/) -
2  (cases [(present /a/ SUBTBL)
3          (cases [(present (get /a/ SUBTBL) /b/)
4                  ([bind (/ans/ VAL)
5                         (get (get /a/ SUBTBL)
6                              /b/)] . e ) ]
7                  [t .e)
8          [t •e)
```

Fig. 3

It is possible to avoid the split into alternative cases if the (<source> <link>) pair of the CHECK is known to be present or absent in the current environment. This occurs if a case predicate further up in the tree (possibly at dbO) implies this. For example, in the case of an invocation of DIFF-IND in an environment where both case predicates are known to be true (i.e. /A/'s SUBTBL link and that table's /B/ link are both known to be present), the above effect description would be simplified to:

```
{[bind (/ans/ VAL)
       (get (get /a/ SUBTBL) /b/)] . e )
```

A CHECK with a closed variable or constant as target is translated as follows:

```
(CHECK (a b C) <present-code> <absent-code>)  ->

      (cases [(- (get a b) C) <present-code>']
             [t  <absent-code>'])
```

where <form>' is the translation of <form>.

A step containing a call to a user-defined procedure in some environment db-in is translated as:

```
(<user-proc-name> <arglist>) =>
     (db-out (<user-proc-name> <arglist> db-in))
```

Here (db-out <form>) denotes the output environment produced by the symbolic evaluation of <form> which will be expanded in-line.

A useful rule for simplification of user procedure calls is:

```
(db-out  (<user-proc-name>
          <arglist>
          |<binding> . <env>)))    ->

(<binding>
 (db-out  (<user-proc-name>
           <arglist>
           <env>))|
```

which is applied whenever it can be shown that the user procedure does not depend on <binding>.

111 WATCHING THE EFFECT DESCRIPTION WITH A DOMAIN-SPECIFIC LIBRARY PLAN (DSLP)

The typical output of symbolic evaluation will be a highly nested cases statement with get-compositions embedded in the predicates and environment. Unless we can assign "meaning" to these get-compositions, we cannot interpret this as a solution to the student's task, since the student may have chosen from a variety of data representations, each requiring special access and data base change operations. To take a rather simple example, the student may have asserted triples like [9 COMPL 1], [8 COMPL 2], etc. If we knew that these triples represented a 10-complement function, we could parse (get x COMPL) as (10-complement x) for any x. Our assistant has a repertoire of low level DSLPs which extract this knowledge from the student's data base for each possibly useful function or predicate in the subtraction domain (e.g. add1, sub 1, val-of, geq).

Once these low level DSLPs are recognized, we can do a top-down analysis similar to that of Ruth [8], starting from a high level DSLP which is predefined by us for each particular task. We will now focus on a typical task undertaken by our students, namely writing a SOLO program to perform two column subtraction. The student is given definitions of SETUP and ANSWER procedures which impose a layout of the form:

```
A        B        |
                  |        SETUP
C        D        |
-----------------
E        F        |        ANSWER
```

The DSLP for this task is like a grammar that can be used to recognize all possible variations in the actual effect description of the user's code. For 2 column subtraction the DSLP has the following definition (omitting db-out for brevity):

```
1 2-COL-SUB (A B C D E F db-in) =
2  (cases [(geq (val-of B) (val-of D))
3          (cases
4           [(geq (val-of A) (val-of C))
5            {[bind (val-of E)
6                   (diff (val-of A) (val-of C))]
7             [bind (val-of F)
8              (diff (val-of B) (val-of D))]
9             . db-in}]
10           [t (fail 2-COL-SUB)])]
11         [t {[bind (val-of E)
12              (borrowed-from
13                (val-of A) (val-of C))]
14             [bind (val-of F)
15              (borrowed-by (val-of B) (val-of D))]
16             . db-in}])
```

Fig. 4

The student may have written the following program for two column subtraction:

```
TO SUBTRACT
1  DIFF-IND B D F
2  CHECK BORROW-FLAG IS SET
   2A If present: DECREMENT A; CONTINUE
   2B If absent: CONTINUE
3  DIFF-IND A C E
4  ANSWER
```

Fig. 5

The assistant recognizes this as a function at the top of the definition hierarchy and symbolic evaluation expands it by inserting the effect descriptions of the called functions (and so on recursively for these). The resulting effect description is a large composition of gets, which ideally should be matched by 2-COL-SUB.

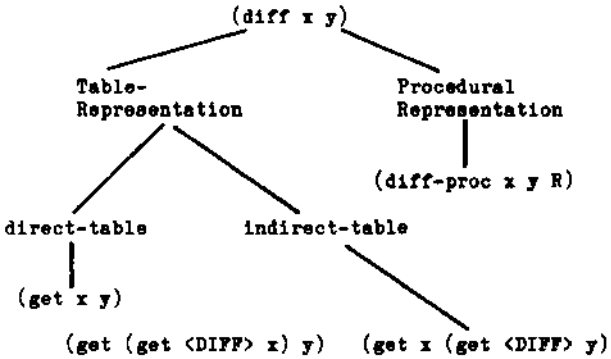
Recognition now proceeds top-down, involving the sub-plans val-of, geq, diff, borrowed-from and borrowed-by. Each of these sub-plans is itself rewritten to conform with the particular representation adopted by the student. E.g. the sub-plan for dealing with a borrowed-from column containing the digits x and y can be re-written in two ways:

```
borrowed-from (x y) = (diff (sub1 x) y) ¦
                      (diff x (add1 y))
```

Similarly, the diff plan (which implements x - y for x > y) must be rewritten in terms of primitives which the student may have elected to use. The difficulty here is that the student may have chosen to implement 'diff in terms of tables in the data base or a recursive counting procedure. We will consider both representations:

1. The student may have invented a 2-digit subtraction table

He may have chosen among a wide variety of table representations. We represent these as a class hierarchy, which for each class has the associated rule for rewriting the user's effect description into diff. <DIFF> is an arbitrary symbol invented by the user.



If we take account of argument reversals, there are 10 possible ways of organizing subtraction tables. Each indirect-table subclass has two triples as parts which are mutually constrained (the outer and the inner get) by the difference predicate. We hypothesize a particular table-class,

and seek confirmation until we can decide on the actual class used by the student. Then we look for all the entries of the table and create an object DIFF which has associated with it the code to recognize (diff a b), e.g. as (get (get a SUB) b). Data base analysis reveals the frequent error-type of missing or inconsistent entries in the data base. Triples which belong to no pre-defined class, but "partially" match some class (i.e. because they contain a known link-name) are categorized as inconsistent and the user is informed about them.

2. The student may have represented 'diff' procedurally

Using add1 and sub 1, he may have written a recursive procedure to implement 'diff. Since a SOLO procedure can only have an effect by side-effecting the data base, the DSLP for a procedural implementation has to change slightly. Any function f, which is represented procedurally as f-proc will be transformed in the library plan as follows:

```
([bind (val-of x) (f <arglist>)] . e) ->
      (f-proc <arglist> N e))
```

where N is the node to be side-effected.

The following example shows a typical student implementation of 'diff together with the effect description produced by the debugging assistant:

```
TO DIFF-REC /X/ /Y/ /ANS/
1 CHECK EQUALS /X/ /Y/
 1A If Present: NOTE /ANS/ VAL 0; EXIT
 1B If  absent: CONTINUE
2 CHECK /X/ SUB1 ?x1
 2A If present: DIFF-REC *x1 /Y/ /ANS/; CONTINUE
 2B If  absent: EXIT
3 CHECK /ANS/ VAL ?v
 3A If present: CONTINUE
 3B If  absent: ERROR; EXIT
4 CHECK  *v ADD1 ?v1
 4A If present: NOTE /ANS/ VAL *v1
 4B If  absent: ERROR; EXIT
```

Symbolic evaluation first produces:

```
diff-rec (x y ans db-in) -
 (cases [(= (get EQUALS x) y)
         ([bind (ans VAL) O] . db-in}]
        [(present (get x SUB1))
         ([bind (ans VAL)
                (get (get ans VAL) ADD1))1 .
          (self (get x SUB1) y ans db-in))])
```

This matches a recursion pattern for which there is a DSLP. The binding being affected in the terminating case (ans VAL) is identical to that of the post-recursive step, which means that (ans VAL) is successively rebound. The DSLP simplifies this to an instance of a prototypical data base entry:

```
[db-entry value   [bind (ans VAL) g007]
          init    ((g007 . 0))
          rec-rel ((x . (get x SUB1))
                   (g007. (get g007 ADD1)))
          term-if (= (get EQUALS x) y)]
```

It says that the value is an (ans VAL) binding whose target can be computed by substituting for g007 the init value and then stepping through the rec-rel a-list substituting for g007 until term-if is true. This representation has the advantage that other recursive implementations (i.e. a tail-recursive one) can be mapped to an equivalent db-entry. The DSLP for diff-proc will match this db-entry instance modulo renaming and substituting (subl x) for (get x SUB1).

IV COMPARING EFFECT DESCRIPTIONS OF USER PLANS WITH LIBRARY PLANS

A variety of user programs for a given task can be recognized as having the same effect description as a pre-stored plan. The reduction to a canonical form allows the user to (1) invent arbitrary names, (2) order steps arbitrarily, (3) break up procedures arbitrarily into sub-procedures, (4) pass values through intermediate nodes of the data base, and (5) invent his own data representation.

What happens if the user's program does not match a library plan?

Our goal is to give the student as much help as possible in detecting the underlying cause of an error. Here, the effect descriptions are useful too: By using the library plan to generate the get-compositions corresponding to his data-representation, we can compare the terminal nodes of both his and the DSLP's effect description for each condition. (The assistant takes care of situations in which nested cases must be reordered for both trees to match.)

A very frequent error which can be detected by comparing effects at terminal nodes is 'name-substitution' (e.g. the student writes w and means v). The assistant compares the bindings added or deleted at each terminal branch of DSLP (goal bindings, gi) with those of the user program (actual bindings, ai). Assume it finds a mismatch of two bindings at some terminal branch:

```
goal-binding:   g = [bind (g1  g2) g3]
actual-binding: a = [bind (a1  a2) a3]
```

For example if in line 3 of DIFF-IND (Fig. 2) the code were NOTE /ANS/ VAL *T, then lines 4-6 of its effect description (Fig. 3) would be:

```
([bind (ans VAL) (a SUBTBL)] . e)                | a
```

instead of

```
([bind (ans VAL) (get (get a SUBTBL) b)] . e| } g
```

as suggested by 2-COL-SUB (lines 5-8 of Fig. 4).

We go up in the symbolic evaluation tree to the place where a was achieved using variable w (i.e. line 3 of DIFF-IND using variable T). If the mismatch was in pair (ai,gi), and in the local environment some other variable was bound to gi, we can propose substituting it for the one used in the

code. In our example the mismatch was in |a3,g3) with a3 being T, and g3 being R, so a substitution of R for T at line 3 is suggested to the student.

If gi is not bound locally, we look at the CHECK (or procedure call) where w was bound and recursively search for the goal of binding w to gi. In case of success, we ask the student whether he wants to accept the proposed renaming.

If this fails, we try step insertion, because another typical source of errors in SOLO is students forgetting a level of indirectness (i.e. using the source of a triple, instead of the target along some link). Lines 1 and 3 in SUBTRACT (Fig. 5) manifest this error, given the implementation of DIFF-IND (Fig. 2).

A CHECK step can be inserted to correct that, if by substituting (get s K) for some subexpression s in ai (where K is some constant link), ai can be made equal to gi. In SUBTRACT the actual effect is:

```
(diff-ind B D F)                              ! a
```

whereas the goal in 2-COL-SUB (lines 7-9 of Fig. 4) gives

```
(diff-ind (get B VAL) (get D VAL) F)          ] g
```

and thus K - VAL. The insertion of two CHECK statements of the form CHECK B VAL ?B-VAL and modification of the calls of DIFF-IND in lines 1 and 3 (Fig. 5) is suggested.

If step insertion fails also, we look at the possibility of reversed present and absent branches (cf. [8]). This manifests itself in the effect description such that the correct data base change has occurred, but at the wrong branch of the tree. In such cases, the user is queried about whether the clauses of the CHECK statement should be reversed.

If this fails also, our last resort is to look for a branch of the tree where the conditions are correct, but there is a disagreement in the effect. The assistant chooses a counterexample that would run through this branch of the tree, and asks the student to try that example on his program. Consider line 2B in DIFF-IND (Fig. 2). The student did not consider the case where (geq a b) does not hold. The effect description for SUBTRACT (Fig. 5) has no effect where it should match lines 11-16 of 2-COL-SUB. We collect the case predicates that lead to this leaf of the symbolic evaluation tree, and see that under the condition

```
(and (present (B SUBTBL))
     (not (geq (val-of B) (val-of D))))
```

SUBTRACT would fail. Since the assistant knows that (val-of B) and (val-of D) are digits, it generates values which fulfill this condition and asks the student to try his program with those.

V CONCLUSIONS AND FURTHER WORK

Our work demonstrates how a combination of domain-specific knowledge and symbolic evaluation techniques can be applied to building a help facility for novice programmers. Writing programs in an elementary assertional data base language like SOLO is experienced by our students as a simple and natural entry point into programming. However, automatically coping with the sorts of bugs which arise through programming by side-effect requires extensions to earlier work on program understanding and debugging. The techniques we have employed give the students a fair amount of freedom in choosing their own representations, a fact which reflects one of the main goals of the course they are taking, i.e. understanding computer modelling and knowledge representation.

We are currently extending the plan library in accordance with empirical observations, in order for the debugging assistant to be capable of handling a wider range of programming tasks.

REFERENCES

[I] Adam, A. and Laurent, J.P. "LAURA, a system to debug student programs." Artificial Intelligence, 15, 1980, 75-122.

[2] Eisenstadt, M. "Artificial intelligence project." Units 3/4 of Cognitive Psychology: A Third Level Course. Milton Keynes, England: Open University Press, 1978.

[3] Eisenstadt, M. and Laubsch, J. "Towards an automated debugging assistant for novice programmers." Proc. AISB-80, Amsterdam, 1980.

[4] Eisenstadt, M., Laubsch, J., and Kahney, J.H. "Creating pleasant programming environments for cognitive science students." Technical report no. 16, CAL Research Group, The Open University, Milton Keynes, England, 1981.

[5] Goldstein, I. "Understanding simple picture programs." Technical Report 294, MIT AI Lab, Cambridge, MA, 1974-

[6] Miller, M. and Goldstein, I. "Structured planning and debugging.** Proc. IJCAI-77. Cambridge, MA, 1977, 773-779-

[7] Rich, C. and Shrobe, H. "Initial report on a LISP programmer's apprentice." IEEE Trans. Soft. Eng., SE-4:6, 1978, 456-466.

[8] Ruth, G. "Intelligent program analysis." Artificial Intelligence, 7, 1976, 65-68.

[9] Shrobe, H., Waters, R., and Sussman, G. "A hypothetical monologue illustrating the knowledge underlying program analysis." AI Memo 507, MIT AI Laboratory, Cambridge, MA, 1979.

[10].Waters, R. "A system for understanding mathematical FORTRAN programs." AI Memo 368, MIT AI Lab, Cambridge, MA, 1976.