HEURISTIC AND FORMAL METHODS IN AUTOMATIC PROGRAM DEBUGGING

William R. Murray

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-10 June 1985

This work was sponsored in part by NSF CER grant MCS-8122039 and Army Research Office grant DAAG29-84-K-0060.

ABSTRACT

TALUS is an automatic program debugging system that both detects and corrects nonsyntactic bugs in student programs written to solve small but nontrivial tasks in pure LISP. TALUS permits significant variability in student solutions by using heuristic methods to recognize different algorithms and formal methods to reason about computational equivalence of program fragments. A theorem prover intensionally represents an infinite database of rewrite rules, thus allowing for unanticipated implementations. TALUS detects bugs using formal methods in both symbolic evaluation and case analysis. Heuristic methods conjecture the exact location of bugs and alterations necessary to correct the bugs. Finally, formal methods establish or disprove these heuristic conjectures, reflecting a generate and test methodology.

I. Introduction

TALUS acts as the domain expert of an intelligent tutoring system to teach LISP. A complete intelligent tutoring system would include a student model, a dialog manager, courseware, and additional domain expertise.

Input to TALUS consists of one or more student functions intended to solve an assigned task. Typical tasks include REVERSE, MEMBER, UNION, INTERSECTION, FACTORIAL, and FLATTEN. Output consists of the debugged student functions and bug annotations.

While TALUS will not recognize unanticipated *algorithms*, unanticipated *implementations* are permitted. Associated with each task are representations of algorithms that solve the task. Heuristics match the student's buggy solution with the algorithm most similar to it. Formal methods detect bugs in the student's functions by comparing them with stored functions. Heuristic methods suggest minimal alterations to the student's functions to remove the bugs; formal methods accept or reject the proposed alterations.

II. Previous Approaches to Automatic Program Debugging

Heuristic approaches to automatic program debugging parse student programs into an abstract representation and match that representation against either stored plan templates from a library [John84; Solo82], or a model program [Adam80]. Bugs appear as unaccountable differences between stored correct plans and the parsed program. Plan transforms increase the range of programs that can be accepted by statically representing common implementation variants. However, some correct student implementations will require inductive proofs or unanticipated transforms to establish their equivalence to stored plans. These implementations are rejected as buggy or unanalyzable.

A formal approach to automatic program debugging, by Katz and Manna [Katz76], extends the logical analysis of programs to include program incorrectness and a means of correcting incorrect programs. Program statements are related to synthesized inductive invariants. When these invariants are insufficient to establish a proof of correctness, program statements are altered so the necessary inductive invariants are derived. Synthesizing these inductive invariants and determining what program statements to alter is difficult; no implementation of their design exists to date.

Shapiro [Shap83] traces the execution of pure PROLOG programs to isolate the presence of bugs in procedures whose traces are incorrect. The user supplies information about examples to try, the correctness of program traces and violations of well founded relationships. Bugs are corrected by synthesizing correct clauses or by searching among

perturbations of buggy clauses. This method can be applied to other functional programming languages, but requires user query and only debugs the program with respect to the examples provided.

Other approaches include analysis by synthesis [Mill82], analysis of program execution [Shap81], plan parsing [Mill79; Ruth73], and analysis of program output [Gold74].

III. Automatic Debugging in TALUS

The debugging approach of TALUS, described in this section, attempts to increase the acceptable variability in student solutions and the robustness of the debugging process, while not relying on the student to assist in the debugging or to understand formal verification techniques. Debugging occurs in three stages: algorithm recognition, bug detection, and bug correction.

A. Algorithm Recognition

All functions, stored or student, are parsed into E-frames. Algorithms and solutions are collections of functions. E-Frame slots represent abstract properties of recursive functions that (partially) enumerate the elements of a recursively defined data structure. A function's E-frame has slots representing its recursion type (tree, list, or number), recursive calls, terminations, variable updates, and task role (main, constructor, or predicate). The E-frame representation facilitates a robust algorithm recognition process by allowing partial matching to occur on the semantic features of abstract enumerations and the role of functions in solving tasks, rather than on code structure.

Figure 1 provides an overview of how TALUS recognizes buggy algorithms. The student writes one or more functions to solve an assigned task. TALUS determines the stored algorithm that best matches the student's algorithm by partial matching, and maps the stored functions associated with that algorithm to the student's functions.

TALUS performs a best first search to choose between competing algorithms and to map student functions to stored functions. Nodes are partial mappings of student and stored functions for one of the competing algorithms.

Function mappings allow for missing or superfluous student functions while using constraints to reduce the search space. Student functions map to stored functions or to EXTRA; stored functions map to student functions or to MISSING. Two functions can be paired only if their parents have already been paired and the functions have the same task role.

A measure of dissimilarity is computed for each partial mapping. Each function pair contributes a penalty that is a weighted sum of the differences between the slots of the

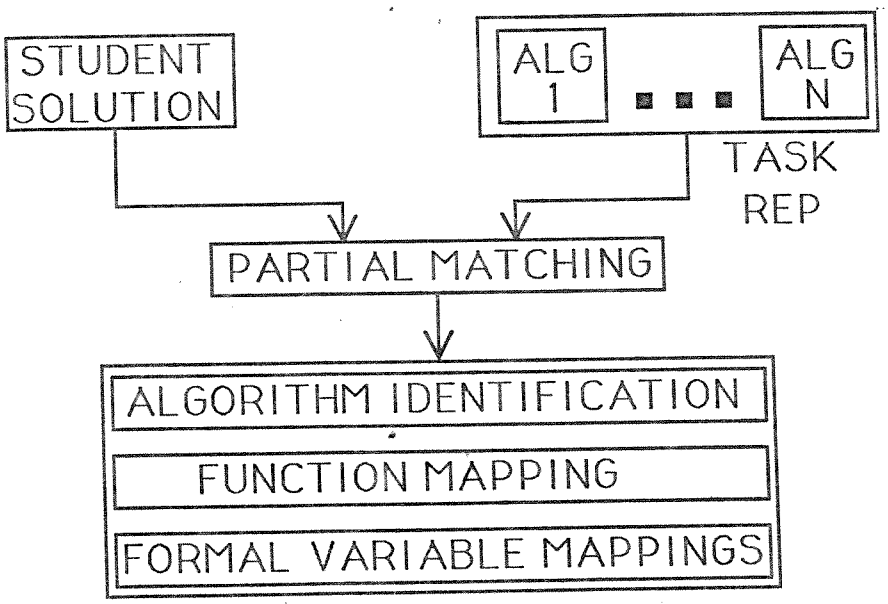

Figure 1: Algorithm Recognition in TALUS

corresponding E-frames. Additional penalties are added for functions mapped to MISSING or EXTRA.

Alternative functional decompositions of algorithms are represented either extensionally as additional task solutions or intensionally through the use of solution transforms (discussed in Section IV).

B. Bug Detection

Figure 2 illustrates how TALUS debugs a student function matched to a stored function. A binary tree represents each function, with nonterminal nodes representing conditional tests and terminal nodes representing function terminations or recursions (i.e. recursive calls). The terms that must be true or false to reach a terminal node are the terms governing that node. Each set of terms governing a terminal node is a case.

For each case, TALUS symbolically evaluates both the student and the stored function. For each function, symbolic evaluation involves reducing its conditionals to a symbolic value (i.e. a termination or recursion). Formal methods determine if a case implies that a conditional test is true or false. Case splitting can occur resulting in more than one symbolic value being returned.

TALUS compares the symbolic values returned by the student and stored functions to determine if they are equal under the assumed case. A theorem prover is used to check that a case implies the computational equivalence of two symbolic values. If they are not equal, a bug is present.

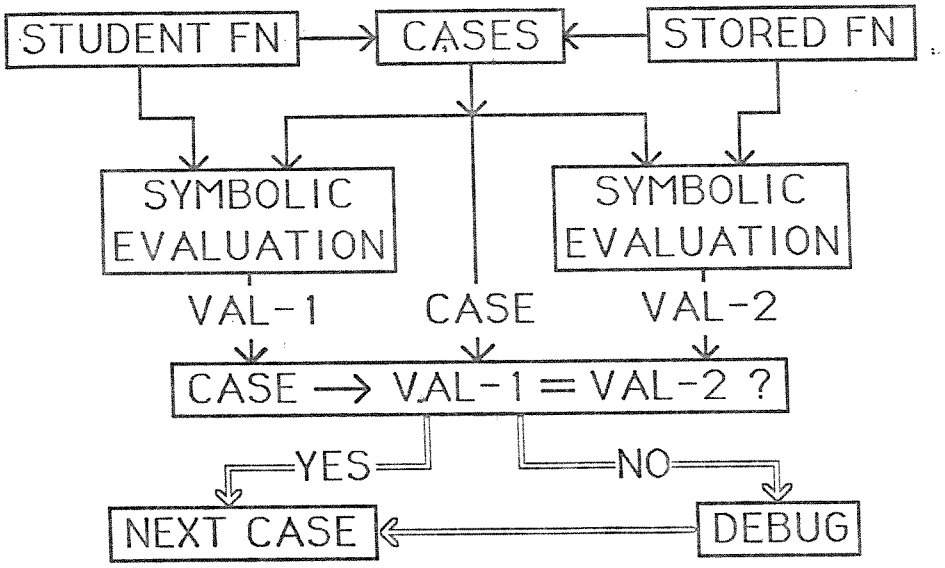

Figure 2: Bug Detection in TALUS

C. Bug Correction

TALUS debugs a student's code fragment by comparing it to the corresponding stored code fragment. Considering only top level expressions, TALUS tentatively replaces one expression in the student's code with the corresponding expression in the stored code. If the two code fragments are now functionally equivalent then the altered code fragment has no remaining bugs. TALUS applies its debugging procedure recursively to the expression replaced whenever possible.

If bugs remain after the replacement then another replacement is tried. If no further replacements are possible then the stored code fragment replaces the entire student code fragment.

IV. Annotated Scenario

The scenario below is edited for brevity. The scenario starts in the middle of a tutorial session, after the student has had some instruction in LISP programming.

Task - MEMTREE
Write a function that determines whether an atom is one of the leaves of a tree.

```
(DEFUN MEMTR (AT CONS)
  (IN AT (FLAT NIL CONS)))

(DEFUN FLAT (ANS TR)
  (IF (ATOM TR) ANS
      (FLAT (FLAT ANS (CDR TR))
            (CAR TR))))

(DEFUN IN (X L)
  (IF (LISTP L)
      (IF (EQUAL L (LIST X))
        L
          (IF (NOT (EQUAL (CAR L) X))
              (IN X (CAR L))
        L))
      NIL))
```

Figure 3: A Buggy Solution to the MEMTREE Task

A. Algorithm Recognition

TALUS must now recognize the algorithm the student has used. First, the functions MEMTR, FLAT, and IN are parsed into E-frames. The three E-frames together represent the student's solution.

TALUS knows of two different algorithms for the MEMTREE task. The TREE-WALK algorithm explores the CAR and the CDR of a tree separately to see if an atom is in the tree. The MEMTREE-FLATTEN algorithm first flattens the tree and then determines if the atom is a member of the resulting bag. The result of the algorithm recognition process is:

```
Algorithm used: MEMTREE-FLATTEN.

Student Fns Matched to Stored Fns:
   FLAT  to FLATTEN
   IN    to MEMBER
   MEMTR to MEMTREE

Solution Transform Applied:
   Transforming FLATTEN to MCFLATTEN to
   better match the student function FLAT.
```

TALUS selects the MEMTREE-FLATTEN algorithm as being more similar to the student's solution than the TREE-WALK algorithm. The stored functions FLATTEN, MEMBER, and MEMTREE, whose E-frames comprise the MEMTREE algorithm, are mapped to the student functions FLAT, IN, and MEMTR.

TALUS has stored global solution transforms that allow it to transform one solution to an equivalent solution, more closely matching the student's solution. Thus, when appropriate, MCFLATTEN replaces FLATTEN, and calls to MCFLATTEN replace calls to FLATTEN. A similar transform allows predicates and predicate calls to be simultaneously logically inverted.

TALUS now maps the formal variables of matched functions by using heuristics that take into account variable data type:

```
FLAT  to MCFLATTEN:  (TR/TREE, ANS/ANSWER).
IN    to MEMBER:     (L/BAG, X/ITEM).
MEMTR to MEMTREE:    (CONS/TREE, AT/ITEM).
```

B. Bug Detection

TALUS now debugs the student functions by comparing them to the stored functions they have been matched with. TALUS matched FLAT to FLATTEN and then transformed FLATTEN to MCFLATTEN to match FLAT better. The stored definition of MCFLATTEN is:

```
(DEFUN MCFLATTEN (TREE ANSWER)
  (IF (ATOM TREE)
      (CONS TREE ANSWER)
      (MCFLATTEN (CAR TREE)
                 (MCFLATTEN (CDR TREE)
                            ANSWER))))
```

In order to facilitate bug detection and allow TALUS to replace buggy student code with stored code to correct bugs, the code above is normalized by replacing the stored function and formal variable names with the student's, and then permuting the formal variable order to match the student's. The result is:

```
(DEFUN FLAT (ANS TR)
    (IF (ATOM TR)
        (CONS TR ANS)
        (FLAT (FLAT ANS (CDR TR))
              (CAR TR))))
```

By examining the stored function, TALUS determines that there are two cases to consider: either (ATOM TR) is true or (NOT (ATOM TR)) is true. By comparing the student's function (see Figure 3) and stored functions for these two cases, we can

determine if they compute the same values under the same conditions. If they do not then a bug is present. The case analysis follows:

```
[Bug found:

(IMPLIES (ATOM TR)                ;Case
         (EQUAL (CONS TR ANS);From Stored Fn
                ANS))             ;From Student Fn

is invalid.]

HINT: Looks like you used the variable ANS
instead of the function call (CONS TR ANS)
in FLAT.

[Check:

(IMPLIES (NOT (ATOM TR))
         (EQUAL (FLAT (FLAT ANS (CDR TR))
                      (CAR TR))
                (FLAT (FLAT ANS (CDR TR))
                      (CAR TR))))

is a theorem.]
```

Conjectures are first checked by a conjecture disprover that runs counterexamples. Counterexamples are stored sets of bindings of formal variables for each function in a stored task algorithm. If the conjecture evaluates true for all counterexamples then it is believed, otherwise it is definitely false.

Conjectures that are believed are then passed to the Boyer Moore Theorem Prover [Boye79] for formal verification. Functions involved in the conjectures are previously defined using the normalized stored function definitions.

If a conjecture is formally proved then no bug is present in the student's code for that case. If the proof of a believed conjecture fails, then either the conjecture is false or necessary lemmas for the proof to succeed are missing. In the example presented in this paper all conjectures that are believed are proven to be theorems by the Boyer Moore Theorem Prover.

For more complex examples, proofs may fail due to the absence of necessary lemmas. When this happens correct implementations are considered buggy and replaced by stored code fragments. With this approach, buggy implementations are always detected.

A more practical but less elegant approach is to accept as true the conjectures believed

by the conjecture disprover. The claim that TALUS relies on formal methods is weakened while its practical performance improves markedly. More complex programs can be debugged since the conjecture disprover needs no lemmas, but some bugs may be missed if no counterexample is found to an invalid conjecture. On the other hand, correct implementations are never considered buggy, and true conjectures that are difficult to prove formally are easily checked by the conjecture disprover.

C. Bug Correction

When a conjecture is invalid, TALUS debugs the student's code by minimally altering the student's code so that the conjecture becomes a theorem. Essentially, TALUS attempts to verify the student's program using the stored function both as its specification and as a source of corrections. Debugging consists of enforcing the verification conditions when necessary. Since the student and stored functions are not always equal when (ATOM TR) is true, a bug is present. TALUS fixes the student's code by replacing only the student's code fragment for this case with the corresponding stored code fragment. The debugged code is shown below:

```
(DEFUN FLAT (ANS TR)
    (IF (ATOM TR)
        (CONS TR ANS)
        (FLAT (FLAT ANS (CDR TR))
              (CAR TR))))
```

The function IN (see Figure 3) is debugged similarly by comparing it to the stored function MEMBER, which is normalized to:

```
(DEFUN IN (X L)
    (IF (NLISTP L)
        NIL
        (IF (EQUAL X (CAR L))
            T
            (IN X (CDR L)))))
```

TALUS generates the following conjectures to check whether the student and stored functions are logically equivalent predicates:

```
(IMPLIES (NLISTP L) (IFF NIL NIL))

(IMPLIES (AND (NOT (NLISTP L))
              (EQUAL X (CAR L))
              (EQUAL L (LIST X)))
         (IFF T L))
```

```
(IMPLIES (AND (NOT (NLISTP L))
              (EQUAL X (CAR L))
              (NOT (EQUAL L (LIST X))))
         (IFF T L))

(IMPLIES (AND (NOT (NLISTP L))
              (NOT (EQUAL X (CAR L))))
         (IFF (IN X (CDR L))
              (IN X (CAR L))))
```

The first three conjectures are theorems while the last is not, indicating a bug which TALUS corrects:

```
(DEFUN IN (X L)
    (IF (LISTP L)
        (IF (EQUAL L (LIST X))
            L
            (IF (NOT (EQUAL (CAR L) X))
                (IN X (CDR L))
                L))
        NIL))
```

The remaining function, MEMTR, has no bugs and its analysis is omitted.

V. Summary

This paper has illustrated a new approach to program debugging that combines both heuristic and formal methods to achieve greater power than either approach alone. Heuristic methods are not used merely to enhance efficiency but in a fundamentally different way: to represent inexact notions that are difficult to express formally, to allow robust algorithm recognition in the presence of bugs, and to generate conjectures to be formally tested.

Formal methods are equally important to the performance of TALUS. Rather than relying on a fixed set of rewrite rules to establish that one implementation is computationally equivalent to another, the full power of a theorem prover capable of inductive proofs can be brought to bear. Implementation equivalences that are only valid under certain conditions can be established. Logical implications, necessary for symbolic evaluation, can be determined.

By using heuristic and formal methods together, TALUS allows significant variability in student input, fully automatic and robust program debugging, and provides debugging skills necessary to a complete intelligent tutoring system that teaches programming.

Acknowledgements

Elaine Rich, Bruce Porter, Mark Miller, and Jim Miller have provided invaluable assistance in this research and its presentation here.

References

[Adam80] Adam, A. and J. Laurent, "LAURA, A System to Debug Student Programs." Artificial Intelligence 15 (1980) 75 - 122.

[Boye79] Boyer, R. and J Moore. A Computational Logic. Academic Press, Inc. 1979.

[Gold74] Goldstein, I. "Understanding Simple Picture Programs." MIT Artificial Intelligence Laboratory, TR 294. 1974.

[John84] Johnson, L. J. and E. Soloway, "Intention-Based Diagnosis of Programming Errors" In Proc. AAAI-84. Austin, Texas, August, 1984, pp. 162-168.

[Katz76] Katz, S. and Z. Manna, "Logical Analysis of Programs." Communications of the ACM 19:4 (1976) 188-206.

[Mill79] Miller, M.L. "A Structured Planning and Debugging Environment for Elementary Programming." International Journal of Man-Machine Studies 11 (1979) 79 - 95.

[Mill82] Miller, J., T. Kehler, P. Michaelis, and W. Murray. "Intelligent Tutoring for Programming Tasks." Technical Report ONR-TR-82-0818F, Texas Instruments, Dallas. 1982.

[Ruth73] Ruth, G. "Analysis of Algorithm Implementations." MIT Project MAC TR 130. 1973.

[Shap81] Shapiro, D. "Sniffer: A System that Understands Bugs." A.I. Memo 638, MIT Artificial Intelligence Laboratory. 1981.

[Shap83] Shapiro, E. Algorithmic Program Debugging. MIT Press. 1983.

[Solo82] Soloway, E., E. Rubin, B. Woolf, J. Bonar, W. L. Johnson. "MENO-II: An AI-Based Programming Tutor." Research Report 258. Department of Computer Science, Yale University. 1982.