Talus: Automatic Program Debugging for Intelligent Tutoring Systems

William R. Murray

AI TR86-32 August 1986

Support for this research was provided by the Army Research Office, under grant number ARO DAAG29-84-K-0060.
Table of Contents

1. Introduction 2
   1.1. Research Goal 2
   1.2. Debugging as Domain Expertise in Intelligent Tutoring Systems 2
   1.3. Bugs versus Misconceptions 3
   1.4. Significant Variability 4
   1.5. Handling Variability by Reasoning about Computational Semantics 4

2. Approaches to Automated Debugging 6

3. Program Verification Approaches to Automated Debugging 8

4. Automatic Program Debugging in Talus 10
   4.1. Bug Detection and Correction 11
       4.1.1. Verification Conditions 14
       4.1.2. Symbolic Evaluation 19
       4.1.3. Conjecture Evaluation 20
   4.2. Algorithm Recognition 23

5. Performance 28

6. Conclusion 32
   6.1. Capabilities and Limitations of Talus 32
   6.2. Reasoning about Computational Semantics in Talus 33
   6.3. Research Contributions 33
**List of Figures**

| Figure 1-1: | Relationship of Talus to a Complete ITS | 3 |
| Figure 1-2: | Two Solutions to FLATTEN Accepted by Talus | 5 |
| Figure 2-1: | A Taxonomy of Automated Debugging Approaches | 7 |
| Figure 3-1: | The Program Verification Process | 9 |
| Figure 3-2: | Automated Debugging based on Program Verification | 10 |
| Figure 3-3: | Bug Detection and Correction in Talus | 11 |
| Figure 4-1: | Three Steps in Talus Debugging | 12 |
| Figure 4-2: | Verification Condition for Base Step | 14 |
| Figure 4-3: | First Verification Condition for Induction Step | 16 |
| Figure 4-4: | Second Verification Condition for Induction Step | 17 |
| Figure 4-5: | Talus System Development | 18 |
| Figure 4-6: | Binary Tree Representations of FLAT and REF | 19 |
| Figure 4-7: | Symbolic Evaluation of FLAT and REF, Base Case | 20 |
| Figure 4-8: | Symbolic Evaluation of FLAT and REF, Induction Step | 21 |
| Figure 4-9: | Conjecture Evaluation in Talus | 21 |
| Figure 4-10: | Algorithm Recognition in Talus | 24 |
| Figure 4-11: | Algorithm Recognition in MEMTREE Example | 25 |
ABSTRACT

Program debugging is an important part of the domain expertise required for intelligent tutoring systems that teach programming languages. This article explores the process by which student programs can be automatically debugged in order to increase the instructional capabilities of these systems. The research presented provides a methodology and implementation for the diagnosis and correction of nontrivial recursive programs. In this approach, recursive programs are debugged by repairing induction proofs in the Boyer-Moore Logic. The induction proofs constructed and debugged assert the computational equivalence of student programs to correct exemplar solutions. Exemplar solutions not only specify correct implementations but also provide correct code to replace buggy student code. Bugs in student code are repaired with heuristics that attempt to minimize the scope of repair.

The automated debugging of student code is greatly complicated by the tremendous variability that arises in student solutions to nontrivial tasks. This variability can be coped with, and debugging performance improved, by explicit reasoning about computational semantics during the debugging process. This article supports these claims by discussing the design, implementation, and evaluation of Talus, an automatic debugger for LISP programs and by examining related work in automated program debugging.

Talus relies on its abilities to reason about computational semantics to perform algorithm recognition, infer code teleology and to automatically detect and correct nonsyntactic errors in student programs written in a restricted, but nontrivial, subset of LISP. Solutions can vary significantly in algorithm, functional decomposition, role of variables, data flow, control flow, values returned by functions, LISP primitives used, and identifiers used. Solutions can consist of multiple functions, each containing multiple bugs. Empirical evaluation demonstrates that Talus achieves high performance in debugging widely varying student solutions to challenging tasks.
1. Introduction

1.1. Research Goal

The goal of this research is to develop and implement an automated debugging methodology that provides domain expertise for intelligent tutoring systems that teach computer programming or programming languages. This capability to debug programs automatically can be used to provide intelligent hint generation to assist students in program debugging. This bug detection capability can also improve student modeling since bug detection is the first step in the inference of misconceptions from student programs. Furthermore, an improved program analysis capability can result in less restrictive tutorial environments.

The desired debugging methodology should be fully automated and capable of algorithm recognition, bug detection, and bug correction. Furthermore, the debugger should be tolerant of the tremendous variability in student’s solutions to exercises and should be capable of handling implementations that are not expressible in terms of a plan library or bug catalog. The program debugger described in this article, called Talus\(^1\), satisfies all these criteria. In contrast, most other debuggers provide bug detection but no algorithm recognition or bug correction capabilities. Furthermore other debuggers may be only partially automated and incapable of handling unanticipated implementations.

There are two assumptions in this research that are appropriate for the domain of intelligent tutoring systems. First, the task assigned to students is known. This is reasonable since code must always be debugged with respect to some specification so some task specification must be provided. Secondly, the task size and complexity, although nontrivial, is appropriate for introductory students learning a programming language.

1.2. Debugging as Domain Expertise in Intelligent Tutoring Systems

The relationship of Talus to a complete intelligent tutoring system (ITS) is shown in Figure 1-1. Talus detects bugs in student code and determines code edits that can remove the bugs. The student modeling component infers misconceptions from this data. The tutorial expert uses this information to devise remedial instruction that is mediated by the dialog manager. The tutorial expert may directly display the bug corrections, generate a hint, provide a counterexample, or monitor and guide the debugging process in some other manner. The remaining module, the courseware module, represents the corpus of material to be taught. Of the modules shown, only Talus has been fully implemented. The design and implementation of the remaining modules has not been addressed in this research.

\(^1\)Pronounced tay’ les [Samuel 71]. Talus is the last of the bronze race in classical mythology.
1.3. Bugs versus Misconceptions

In general, Talus does not infer misconceptions but rather detects and corrects bugs. Misconceptions are psychologically plausible misunderstandings of the student that account for bugs manifested in the student’s programs. Common misconceptions [Joni 83] include:

- **Overgeneralized Concepts.** The student believes a programming language construct is applicable in situations where it is not.

- **Inability to Use Multiple Constructs Together Correctly.** For example, the student can use loop variables and array indices separately but cannot coordinate their use in a loop to search through the elements of an array.

- **Incorrect Analogies.** The student treats the programming language like English, believing that the computer understands the tokens in his program.

Accurate bug detection is an important first step in the inference of these misconceptions. Misconceptions cannot be inferred without some means of detecting the bugs that provide evidence for their existence.

Misconceptions are manifested as one of two kinds of bugs, both of which Talus detects. The first kind, nonstylistic bugs, result in program nontermination or termination with incorrect values or an error. Using ADD1 instead of SUB1, or omitting a conditional test can result in this kind of error. The second kind of bug, stylistic bugs, are errors in programming style whose only effect are to render code more verbose, less efficient, or less intelligible to other programmers. Examples include nonmnemonic identifiers, unnecessary function definitions that redefine primitives, and unnecessary conditional tests.
1.4. Significant Variability
Detecting both stylistic and nonstylistic bugs is difficult because of the tremendous variability in student code. This variability complicates debugging since an automated debugger must distinguish between correct and incorrect implementation variants. Student solutions vary significantly because of the multitude of design decisions required to write programs that implement task solutions. Consider a sorting task. Several different algorithms can be selected: HEAPSORT, QUICKSORT, INSERTION-SORT, etc. Each algorithm can be procedurally decomposed in many different ways, and the student may omit necessary procedures or add superfluous procedures. Given a single procedure there can be an infinite number of correct implementations that vary according to the identifiers used, the control and data flow, the ordering and embedding of conditional tests, the results returned by functions, the specific programming constructs used, and whether or not side-effects can occur. Even more variability is introduced when buggy implementations are considered.

Suppose the task is to flatten a list, returning a list of those atoms that occur in the input at any level of embedding. The two function definitions shown in Figure 1-2 will both be accepted as correct by Talus, even though they vary in many superficial ways:

- **Identifiers.** FLATTEN and TREE vs. FLAT and LIS.
- **Conditional Tests.** FLATTEN has a single conditional test with the predicate (ATOM TREE). FLAT has two conditional tests with the predicates (LISTP LIS) and (ATOM (CAR LIS)).
- **Special Forms.** IF in FLATTEN vs. COND in FLAT.
- **Side-Effects.** No side-effects can occur in FLATTEN but NCONC can alter shared list structure in FLAT.
- **Function Terminations.** When the input (TREE) to FLATTEN is atomic it terminates and returns (LIST TREE). When the input (LIS) to FLAT is atomic it also terminates but returns (CONS LIS NIL).
- **Function Recursions.** The first function, FLATTEN, can be called recursively on the CAR and CDR of the input. A similar recursive call occurs in FLAT, but FLAT contains another recursive call where FLAT is only called on the CDR. No similar call occurs in FLATTEN.
- **Programming Language Constructs.** FLATTEN never calls CONS but does call APPEND. In contrast, FLAT never calls APPEND but does call CONS.

1.5. Handling Variability by Reasoning about Computational Semantics
Talus can handle this range of variability in student programs by relying on its ability to reason about computational semantics during the debugging process. The thesis of this research is that the potential debugging capabilities of an automated program debugger are proportional to its ability to reason about computational semantics. Support for this thesis is provided by examining related work in automated debugging in [Murray 86] and by considering the debugging process itself. Automated debugging can be modeled as a two-step process:

1. **Infer Code Teleology by Mapping Task Specifications to Code Fragments.** The first step is to determine what the purpose of the code is.
(DEFUN FLATTEN (TREE)
  (IF (ATOM TREE)
      (LIST TREE)
      (APPEND (FLATTEN (CAR TREE))
              (FLATTEN (CDR TREE))))
)

(DEFUN FLAT (LIS)
  (COND ((LISTP LIS)
          (COND ((ATOM (CAR LIS))
                 (CONS (CAR LIS)
                      (FLAT (CDR LIS))))
                 (T (NCONC (FLAT (CAR LIS))
                          (FLAT (CDR LIS)))))))
)

Figure 1-2: Two Solutions to FLATTEN Accepted by Talus

2. Determine if Code Fragments Satisfy Specifications. Next each code fragment must be checked to see if it successfully achieves its intended role in the task solution.

It is in the second step that computational semantics is most important. Prior research by Johnson [Johnson 85] has demonstrated the importance of inferring code teleology; this research claims that program reasoning capabilities play an equally important role. Furthermore, and more importantly, this research shows how program reasoning capabilities can be increased to boost performance in automated debugging.

The primary program reasoning capabilities that Talus relies on are its ability to recognize intended algorithms in buggy student programs, and its ability to reason about programming language semantics. Talus recognizes intended algorithms underlying buggy student programs by a heuristic reasoning process. The algorithm recognition problem in the LISP domain of Talus can be described as follows. Given one or more functions as a solution to some task, the following questions must be answered:

- What algorithm was intended?
- What is the purpose of each function in implementing the algorithm?
- What roles do the variables play in implementing the functions?

Heuristic reasoning is required since it is very difficult to formalize what it means for a buggy student program to appear more similar to the implementation of one algorithm than another. The partial matching process required for algorithm recognition is a heuristically guided search that is described in more detail in Section 4.2.

A more formal reasoning process is required to recognize logically equivalent predicates or computationally equivalent expressions. Talus can also use this ability to reason about programming language semantics to establish properties other than equivalence for the operators and constructs that
occur in student programs. This second, more formal, kind of program reasoning is performed by evaluating conjectures in the Boyer-Moore Logic. For example, the conjecture below:

\[(\text{IMPLIES} \ (\text{AND} \ (\text{NUMBERP} \ X) \ (\text{NUMBERP} \ Y)) \ (\text{EQUAL} \ (\text{PLUS} \ X \ Y) \ (\text{PLUS} \ Y \ X)))\]

is a well-formed formula in the Boyer-Moore Logic that asserts the commutativity of the operator "PLUS", provided that its arguments are both numbers. By proving that the conjecture above is a theorem, using the Boyer-Moore Theorem-Prover, Talus can establish that the conjectured property (commutativity) holds, and that the code fragments (PLUS X Y) and (PLUS Y X) are computationally equivalent under the specified conditions. Similarly, by proving that the conjecture

\[(\text{EQUAL} \ (\text{NOT} \ (\text{OR} \ X \ Y)) \ (\text{AND} \ (\text{NOT} \ X) \ (\text{NOT} \ Y)))\]

is a theorem, Talus can establish a logical equivalence of two predicates that is a result of DeMorgan’s Laws. Reasoning about logical implications can also be done by evaluating conjectures. For example, by proving that the conjecture

\[(\text{IMPLIES} \ (\text{GREATERP} \ X \ Y) \ (\text{NOT} \ (\text{LESSP} \ X \ Y)))\]

is a theorem, Talus confirms that X is greater than Y implies that X is not less than Y.

The remainder of this paper presents the debugging approach of Talus and contrasts it with related work in automated debugging. Section 2 surveys the various approaches to automated debugging to place the debugging approach of Talus into perspective. This approach, based on program verification, is discussed in Section 3. Section 4 provides case studies to illustrate the three steps in the debugging of Talus: algorithm recognition, bug detection, and bug correction. Performance issues are addressed in Section 5. Finally, Section 6 concludes by summarizing the research contributions of this work.

2. Approaches to Automated Debugging

This section classifies different debugging approaches to facilitate comparison and to emphasize trade-offs between alternate approaches. Figure 2-1 provides a taxonomic classification of the different approaches to automated debugging that have been developed over the past fifteen years. The main division occurs between approaches based on dynamic analysis and those based on static analysis. The former require program execution on specific examples; the latter examine program source code rather than execution traces.

Considering dynamic analysis in more detail, there is a spectrum of approaches depending on how much of the program execution history is analyzed. On one end of the spectrum, only input/output pairs are examined. This is the approach taken by the Basic Instruction Program, BIP [Barr 76]. MYCROFT [Goldstein 74] analyzes all program side-effects, not just input and output. Side-effects in MYCROFT's LOGO domain include changes to turtle state and position. An analysis of procedure calls and values returned suffices in functional programming languages where side-effects cannot occur.
Ehud Shapiro's PDS6 [Shapiro 83] debugs pure PROLOG programs by performing this kind of trace analysis. On the far end of the spectrum Daniel Shapiro's SNIFFER [Shapiro 81] examines not only procedure calls and returns but all side-effects, including side-effects to shared list structure.

The dynamic analysis approach can rapidly locate bugs in procedures with minimal program analysis. However, errors in programming style can be difficult to detect and some, such as unreachable code, cannot be detected at all. Nonstylistic bugs can also be missed if the program is tested on insufficient examples.

Static analysis can detect errors that are difficult or impossible to detect with dynamic analysis. But a more thorough program analysis is required, and partial or failed program analyses can result in a number of undetected bugs and erroneous bug reports (false alarms). Two different approaches to static analysis are plan-based program analysis and program verification. We will consider plan-based program analysis first.
Debuggers performing plan-based program analysis are form-based; this means that they look for surface structural forms, such as code templates, in student programs. These debuggers can be modeled as having two grammars, one for buggy programs and one for correct programs. Programs that cannot be parsed with either grammar are unanticipated implementations and are problematic for these approaches.

Approaches under the plan-based program analysis category further divide according to the parsing method. In the plan-parsing approach grammars of correct and buggy programs are represented explicitly by augmented transition networks or context free grammars. In heuristic plan-recognition, code templates from a plan library (e.g. MENO-II [Soloway 83] or PROUST [Johnson 84]) or a model program (e.g. LAURA [Adam 80]) are matched to student code. In the analysis-by-synthesis approach correct and buggy programs are synthesized. Following the student’s program design as it develops simplifies the program synthesis task. Both GREATERP [Reiser 85] and TURTLE [Miller 82] adopt this approach.

The other category under static analysis are approaches based on program verification. The student’s program is compared to a task specification. A proof of correctness is constructed. Failures in the proof are interpreted as indicating bugs in the student’s program. This is the approach taken by Talus, and discussed in more detail in the next section.

3. Program Verification Approaches to Automated Debugging

This section explains how program verification can be extended to provide a debugging methodology. Figure 3-1 is a conceptual\(^2\) model of Floyd-Hoare style program verification. Input-output specifications and loop invariants are the task specification for the target or student code. A verification condition generator generates conjectures that are well-formed formulas in some formal logic. The correctness of these verification conditions is sufficient to prove that the target code is correct with respect to the specifications. The verification process can be partially automated by introducing a theorem prover that attempts to establish the validity of the verification conditions. Failed verification conditions indicate that the specification is incorrect, or the target code contains bugs, or that additional lemmas are necessary to prove that a valid conjecture is a theorem.

Program verification can be extended to support automated debugging as shown in Figure 3-2. Unproved conjectures are interpreted as indicating bugs in the student or target program. Some means of determining bugs from failed conjectures drives a code editing process that alters the target code. The editing is performed in such a way that the verification conditions that result from the altered code are guaranteed correct. In essence, the verification conditions are enforced to ensure that there are no bugs in the program.

\(^2\)This diagram is also an accurate depiction of early program verification systems but current verification systems may intermix the stages shown.
The debugging approach of Katz and Manna [Katz 76], based on analyzing loop invariants, follows the design of Figure 3-2. Inductive invariants are mechanically synthesized from target code and compared to loop invariants either provided as part of the task specifications or synthesized from the input-output specifications. When the loop invariants from the target code are insufficient to support a proof of correctness, the target code is altered so the necessary invariants are obtained. However, it is difficult to synthesize loop invariants and to determine the proper alterations to perform on the target code. No implementation of this design exists to date.

The debugging approach of Talus, shown in Figure 3-3, does not require the synthesis of loop invariants. Recursive programs in a functional programming language are analyzed rather than imperative programs in a procedural programming language. Both the program to be debugged and the task specification are LISP function definitions. The task specification is provided by a reference function, this is an exemplar or model solution that correctly implements the assigned task. The student function is considered correct if it is computationally equivalent to the reference function. Since both functions can be recursive an inductive proof is required to establish their equivalence. The Principle of

---

3Talus has a limited facility to debug programs that can perform side-effects; much more work remains to be done on this problem. The present capabilities are discussed in [Murray 86].
Induction, a schema for induction proofs in the Boyer-Moore Logic, specifies the verification conditions sufficient for the proof of correctness. The Boyer-Moore Theorem-Prover evaluates the conjectures. Failed verification conditions are interpreted as indicating bugs in student code. The bugs are repaired and the verification conditions enforced by replacing buggy student code fragments with code fragments obtained from the reference function. Thus the reference function acts both as a specification and as a source of corrections.

4. Automatic Program Debugging in Talus

This section presents examples to clarify the debugging process just described. The example in Section 4.1 illustrates the bug detection and correction process of Figure 3-3. Section 4.2 addresses the problem of determining the most appropriate specifications for student programs prior to applying the bug detection and correction process. The case study there illustrates a solution containing multiple LISP functions, some of which contain multiple formal variables. Talus must determine the algorithm intended in this solution, and then pair student functions to reference functions. Then bug detection and
correction proceeds with each student-reference function pair.

4.1. Bug Detection and Correction

There are actually three steps to the debugging process of Talus, as shown in Figure 4-1. At this point, we are only considering the bug detection and correction process depicted in Figure 3-3. Later in Section 4.2 we will return to the algorithm recognition process.

Consider a task to flatten an s-expression, as described to the student below:

**Task:** Write a function to return a list of all the atoms in a tree.

Suppose the student's solution is the following code:
(DEFUN FLAT (TREE)
  (IF (LISTP TREE)
      (IF (ATOM (CAR TREE))
          (CONS (FLAT (CAR TREE))
                (CONS (FLAT (CAR TREE)) ;Bug: CONS used instead of APPEND
                           (FLAT (CAR TREE)))
          (CONS TREE NIL)))

If TREE is atomic then (CONS TREE NIL) is returned. This is correct. If TREE is nonatomic then the second conditional test determines if the CAR of TREE is atomic or not. This second conditional test is unnecessary and makes the code more verbose. Thus it is a stylistic error. If the CAR of TREE is atomic then FLAT is called recursively on the CDR of TREE and the CAR of TREE is CONSed onto the result. This is also correct. However if the CAR is nonatomic then FLAT is called recursively on the CAR and CDR and the results are also CONSed together. The second CONS has been underlined here to point out that it is incorrect: APPEND should be used to join the results together. This use of CONS instead of APPEND is an example of a nonstylistic error.

With this code as input, Talus returns the following debugged code as output:
(DEFUN FLAT (TREE)
  (IF (LISTP TREE)
      (IF (ATOM (CAR TREE))
          (CONS (CAR TREE) (FLAT (CDR TREE))))
      (APPEND (FLAT (CAR TREE))
              (FLAT (CDR TREE))))
    (CONS TREE NIL))

Talus displays both the original and debugged code. Buggy s-expressions are highlighted in the original code and bug fixes are highlighted in the corrected code. Talus also generates the hints shown below:

**HINT:** Looks like you used the function CONS instead of the function APPEND in FLAT.

**NOTE:** Extra condition in FLAT. The extra condition is (ATOM (CAR TREE)).

Notice that Talus actually corrected the nonstylistic error but only commented on the stylistic error.

It is worth reemphasizing that Talus does not address pedagogical issues in the use of this information. An actual ITS incorporating Talus would probably not display debugged code or generate the hints above. Instead the information above would be used by the student modeling component and tutorial expert (see Figure 1-1) to update the student model and determine appropriate remedial instruction. The hints above only demonstrate the detail of advice that can be provided.

Now we examine the debugging process in detail. The reference function, which acts as the task specification, is shown below:

(DEFUN REF (TREE)
  (IF (ATOM TREE)
      (LIST TREE)
      (APPEND (REF (CAR TREE))
              (REF (CDR TREE)))))

Notice that it differs from the student function in several ways:

- **Different Function Terminations.** REF terminates with (LIST TREE) while FLAT terminates with (CONS TREE NIL).

- **Different Conditional Tests.** The first conditional test in FLAT is (LISTP TREE) while the first test in REF is (ATOM TREE). Also, FLAT tests to see if the CAR of TREE is atomic while REF does not.

- **Different Recursive Calls.** FLAT contains an extra recursive call not present in REF.

- **Different Programming Language Constructs.** REF uses APPEND but not CONS while FLAT uses CONS but not APPEND.

The ability to reason about programming language semantics during the debugging process allows Talus to ignore the superficial variability between FLAT and REF while detecting the bugs in FLAT.

The debugging process can be thought of in two different ways. In the first way, a case analysis is performed to break down the reference and code functions into individual code fragments. Then
corresponding code fragments are compared for functional equivalence. This is a convenient approximation but is not quite accurate. In the second (correct) way of thinking about the debugging process, bug detection is related to induction theorem proving. The verification conditions generated for bug detection are those necessary for an induction proof of functional equivalence between the student and reference function. The difference between these two ways of thinking will be pointed out below.

4.1.1. Verification Conditions

Talus determines cases to consider by extracting conditional tests from the reference and student functions. By extracting the test (ATOM TREE) from REF, Talus determines that there are two cases to consider: either TREE is atomic or it is not. In the first case TREE is atomic and the reference function simplifies to (LIST TREE). The student function simplifies to (CONS TREE NIL). These two code fragments must be computationally equivalent when TREE is an atom, otherwise REF and FLAT cannot be equal. This first case, shown in Figure 4-2, corresponds to the base step in the induction proof of equivalence between REF and FLAT.

---

**Student Function**

```
(defun flat (tree)
  (if (listp tree)
      (if (atom (car tree))
          (cons (car tree) (flat (cdr tree)))
          (cons (flat (car tree)) (flat (cdr tree))))
      (cons tree nil))
```

**Reference Function**

```
(defun ref (tree)
  (if (atom tree)
      (list tree)
      (append (ref (car tree)) (ref (cdr tree))))
```

---

**Base Case: (ATOM TREE)**

```
(atoms tree) ;Case
(equal (cons tree nil) ;From Student Fn
      (list tree)) ;From Reference Fn
---> true
```

Figure 4-2: Verification Condition for Base Step

---

4Functional equivalence is a generalization of computational equivalence that only requires predicates to be logically equivalent.
The second case occurs when TREE is not atomic. In that case the code fragments and verification conditions that result are shown in Figure 4-3 and Figure 4-4. Two verification conditions result, corresponding to two different code fragments in the student’s code, because further case-splitting is required to evaluate the student’s code when (NOT (ATOM TREE)) is assumed true. Case-splitting on the predicate (ATOM (CAR TREE)) occurs since (NOT (ATOM TREE)) is insufficient to either imply or falsify that predicate. So in the first verification condition (Figure 4-3) the predicate is assumed true and in the second verification condition (Figure 4-4) it is assumed false.

The first verification condition is true while the second (Figure 4-4) is not, indicating a bug in the code fragment

(CONS (FLAT (CAR TREE)) (FLAT (CDR TREE)))

in the student’s code. This bug can be repaired by determining the minimal replacement from the corresponding code fragment derived from the reference code:

(APPEND (FLAT (CAR TREE)) (FLAT (CDR TREE)))

that will result in a verification condition that is true. A linear substitution scheme is applied where s-expressions in the student’s code fragment are replaced by s-expressions in the reference code fragment. When the verification condition that would result is true the bugs have been corrected. If no further isolation is possible then the entire buggy student code fragment is replaced. However it should be emphasized that Talus attempts to minimize the amount of code replaced wherever possible. During the generation of verification conditions, case analysis allows bugs to be isolated to code fragments within conditional expressions. Then the bug isolation heuristics described above can frequently isolate bugs to individual atoms within these code fragments. For the current example, Talus determines that replacing CONS by APPEND is sufficient. The following hint is generated from a template:

HINT: Looks like you used the function CONS instead of the function APPEND in FLAT.

This example illustrates how Talus can reason about programming language semantics to distinguish between correct and incorrect implementation variants. Of the three occurrences of CONS in FLAT, Talus correctly selects the one that is buggy while not flagging the others as buggy, even though the reference function itself never uses CONS. Other important points in this example are:

- **Reasoning about Computational Semantics Replaced:**
  - A Plan Library. There was no database of rewrite rules to enumerate anticipated implementation variants. (CONS TREE NIL) and (LIST TREE) were shown to be equivalent by proving a theorem rather than retrieving a rewrite rule.
  - A Bug Catalog. There was no library of anticipated bugs. The bug where CONS was used in place of APPEND was detected without any sort of CONS/APPEND bug rules.
  - Pattern Matching. No syntactic pattern matching occurred at any time.

---

Note that the function identifier REF has been replaced with FLAT. Induction hypotheses allow the substitution of function identifiers in the verification conditions and bug corrections.
Student Function

\[
(\text{DEFUN FLAT} \ (\text{TREE})
  \ (\text{IF} \ (\text{LISTP} \ \text{TREE})
  \ (\text{IF} \ (\text{ATOM} \ (\text{CAR} \ \text{TREE}))
  \ (\text{CONS} \ (\text{CAR} \ \text{TREE}) \ (\text{FLAT} \ (\text{CDR} \ \text{TREE})))
  \ (\text{CONS} \ (\text{FLAT} \ (\text{CAR} \ \text{TREE}))
  \ (\text{FLAT} \ (\text{CDR} \ \text{TREE})))
  \ (\text{CONS} \ \text{TREE} \ \text{NIL})))
\]

Reference Function

\[
(\text{DEFUN REF} \ (\text{TREE})
  \ (\text{IF} \ (\text{ATOM} \ \text{TREE})
  \ (\text{LIST} \ \text{TREE})
  \ (\text{APPEND} \ (\text{REF} \ (\text{CAR} \ \text{TREE}))
  \ (\text{REF} \ (\text{CDR} \ \text{TREE}))))
\]

Induction Step, Case 1:

\[
(\text{IMPLIEDS} \ (\text{AND} \ (\text{NOT} \ (\text{ATOM} \ \text{TREE})))
  \ (\text{ATOM} \ (\text{CAR} \ \text{TREE})))
  \ (\text{EQUAL} \ (\text{CONS} \ (\text{CAR} \ \text{TREE})
  \ (\text{REF} \ (\text{CDR} \ \text{TREE})))
  \ (\text{APPEND} \ (\text{REF} \ (\text{CAR} \ \text{TREE}))
  \ (\text{REF} \ (\text{CDR} \ \text{TREE}))))
\]

\[\Rightarrow \text{true}\]

Figure 4-3: First Verification Condition for Induction Step

- No Program Execution was Required. The student's program was never executed during debugging.

- Case Splitting allowed Bug Isolation. The case splitting in the induction proof allowed bugs to be isolated to code fragments containing no IF-expressions. In many cases heuristics allow bugs to be isolated to atomic s-expressions within these code fragments.

- Bugs were Detected by Failed Verification Conditions. The bug in the student's code was detected when the second verification condition for the induction step failed.

- Bugs were Corrected by Enforcing Verification Conditions. The code was corrected by altering the student's code so that the new verification conditions would be valid.

- The Reference Function Acted both as
  - A Task Specification. No loop invariants were required.
  - A Source of Bug Corrections. The bug correction (APPEND/CONS) was obtained from the reference function.
Student Function

\[
\text{(DEFUN FLAT (TREE)}
\begin{align*}
\text{ (IF (LISTP TREE)} & \text{)} \\
& \text{ (IF (ATOM (CAR TREE))} \\
& \text{ (CONS (CAR TREE) (FLAT (CDR TREE)))} \\
& \text{ (CONS (FLAT (CAR TREE))} \\
& \text{ (FLAT (CDR TREE)))} \\
& \text{ (CONS TREE NIL)))}
\end{align*}
\]

Reference Function

\[
\text{(DEFUN REF (TREE)}
\begin{align*}
\text{ (IF (ATOM TREE)} & \text{)} \\
& \text{ (LIST TREE)} \\
& \text{ (APPEND (REF (CAR TREE))} \\
& \text{ (REF (CDR TREE))))
\end{align*}
\]

Induction Step, Case 2:

\[
\text{(IMPLIES (AND (NOT (ATOM TREE))} & \text{)} \\
\text{ (NOT (ATOM (CAR TREE))))} & \text{ ;Case} \\
\text{ (EQUAL (CONS (REF (CAR TREE))} & \text{ ;From Student Fn} \\
\text{ (REF (CDR TREE))))} \\
\text{ (APPEND (REF (CAR TREE))} & \text{ ;From Reference Fn} \\
\text{ (REF (CDR TREE))))
\]

\[\rightarrow false\]

HINT: Looks like you used the function \text{CONS} instead of the function \text{APPEND} in \text{FLAT}.

\text{Figure 4-4: Second Verification Condition for Induction Step}
- *Multiple Formal Variables*. Functions can use more than one formal variable.

- *Multiple Algorithms and Functions*. Solutions to tasks can contain more than one function and tasks can be solved with different algorithms.

- *Extended LISP*. Some constructs, such as COND and PROG, that are not present in the Boyer-Moore Logic can be allowed.

- *Side-Effects*. Additional constructs, such as NCONC or PUTPROP, that cause side-effects can also be allowed.

A full treatment of these extensions is provided in [Murray 86]. Only a brief overview of the extensions to detect missing and extra conditional tests and to perform algorithm recognition will be presented here.

---

**Figure 4-5:** Talus System Development
4.1.2. Symbolic Evaluation

Both the case analysis and the detection of the extra conditional expression in the example above relied on symbolically evaluating the student and reference functions. To perform this symbolic evaluation the student and reference function are represented as binary trees as shown in Figure 4-6. Nonterminal nodes represent conditional tests in functions and terminal nodes represent function terminations and recursions. Cases are determined by examining the nodes above terminal nodes.

![Binary Tree Representations of FLAT and REF](image)

Figure 4-6: Binary Tree Representations of FLAT and REF

Symbolic evaluation is modeled as determining a path or paths through these trees to terminal nodes. The situation for the base case is shown in Figure 4-7. Note that each function evaluates to one leaf, and this evaluation can be viewed as a one-to-one mapping between the two leaves. The induction step is different since case-splitting is forced on the predicate \( \text{ATOM (CAR TREE)} \), which is neither implied nor falsified when \( \text{NOT (ATOM TREE)} \) is assumed. This situation is shown in Figure 4-8. In this case the symbolic evaluation can be viewed as determining a one-to-many mapping. Whenever a one-to-many mapping occurs either an extra or a missing conditional test is signaled, depending on the direction of mapping.
4.1.3. Conjecture Evaluation

The discussion of symbolic evaluation and verification conditions has assumed that the Boyer-Moore Theorem-Prover is the sole means of evaluating conjectures. Actually, there are two other means of conjecture evaluation available to Tatus, as shown in Figure 4-9. The conjecture disprover and counterexample generator, described below, both act as filters to the Boyer-Moore Theorem-Prover. Each of the three means of conjecture evaluation can act alone or in concert with the other methods.

The conjecture disprover searches stored examples for a counterexample to a conjecture. The stored examples are sets of bindings of formal variables associated with each reference function. They are part of the task representation. If a conjecture evaluates true for all stored examples then it is believed otherwise it is definitely false and one of the stored examples provides a counterexample to it. For example, one of the stored examples for the reference function for FLAT binds TREE to the list '((A) B C). This example is sufficient to falsify the conjecture.
Figure 4-8: Symbolic Evaluation of FLAT and REF, Induction Step

Figure 4-9: Conjecture Evaluation in Talus
\( (\text{IMPLIES} (\text{AND} (\text{NOT} (\text{ATOM TREE}))) \quad (\text{NOT} (\text{ATOM} (\text{CAR TREE})))) \quad (\text{EQUAL} (\text{CONS} (\text{FLAT} (\text{CAR TREE}))) \quad (\text{FLAT} (\text{CDR TREE}))) \quad (\text{APPEND} (\text{FLAT} (\text{CAR TREE})) \quad (\text{FLAT} (\text{CDR TREE})))) \) 

which arose in the example of Section 4.1.

The counterexample generator is more powerful than the conjecture disprover since it applies heuristics to actively generate counterexamples. To generate counterexamples for conjecture, Talus relies on an example generator, the EGS system [Kim 85]. The EGS system attempts to construct an example for (NOT conjecture). The example generator works by unfolding functions, merging stored examples, and reducing the conjecture by transformations and in the process building up examples.

Conjectures that are believed can be passed to the Boyer-Moore Theorem-Prover for formal verification. Functions involved in the conjectures are previously defined with the reference functions since they are known to be correct. 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. All valid conjectures presented in this paper can be proven to be theorems by the Boyer-Moore Theorem-Prover.

For more complex tasks, such as the SINGLETONS task presented in Section 5, proofs may fail due to the absence of necessary lemmas. When this happens correct implementations are considered buggy and replaced by stored code fragments. However, buggy implementations will always\(^6\) be detected when the Boyer-Moore Theorem-Prover evaluates conjectures.

A more practical but less elegant approach is to accept as true the conjectures believed by the conjecture disprover. 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\(^7\), and true conjectures that are difficult to prove formally are easily checked by the conjecture disprover. In fact, Talus normally evaluates conjectures only with the disprover because of its greater speed and lower rate of false alarms relative to the other means of conjecture evaluation.

All these methods of conjecture evaluation are imperfect. No perfect means of evaluation is possible since both program equivalence and the Boyer-Moore Logic are undecidable. In practice, Talus does not deal with arbitrarily complex conjectures. Instead it deals with conjectures constructed

---

\(^6\)Since we are only discussing bug detection and correction at this point, the discussion in this section assumes that the algorithm recognition heuristics described in Section 4.2 operate correctly.

\(^7\)Pedagogically, it is more desirable to miss some bug diagnoses than to provide inappropriate instruction based on false alarms.
from code fragments taken from small student and reference functions. Empirical evaluation, discussed in Section 5, demonstrates that Talus can reliably evaluate these conjectures in the great majority of cases. Heuristics, discussed in Section 4.2, also simplify the program verification task by selecting those reference functions most similar to student functions, based on abstract computational features.

4.2. Algorithm Recognition

Now we return to consider the first step in the three-step debugging process shown in Figure 4-1, the process of algorithm recognition. The student's solution can consist of several function definitions. The solution is heuristically mapped to representations of algorithms that are part of the prestored task representation of each task. Figure 4-10 illustrates this process. There are three results to the partial matching process:

- **Algorithm Identification.** The student's algorithm is identified as one of those in task representation.
- **Function Mapping.** The student's functions are mapped to the reference functions associated with the identified algorithm. These reference functions are also part of the task representation.
- **Formal Variable Mappings.** The formal variables of the reference function are mapped to those of the student function for each student-reference function pair.

The partial matching process does not involve pattern matching. Instead it is implemented as a best first search through a space of function mappings that are scored for plausibility by a heuristic evaluation function.

An example illustrating the input and output of the algorithm recognition process will be provided before considering the details of process itself. Assume the task is as described below:

**Write a function that determines whether an atom is one of the leaves of a tree.**

Consider the solution below:

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

(DEFUN FLAT (TREE) ;Note: same definition as earlier
  (IF (LISTP TREE)
       (IF (ATOM (CAR TREE))
           (CONS (CAR TREE) (FLAT (CDR TREE)))
           (CONS (FLAT (CAR TREE))
                 (FLAT (CDR TREE)))
           (CONS TREE NIL)))
```
(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))

The solution consists of three functions: MEMTR, FLAT, and IN. MEMTR flattens CONS by calling FLAT and then determines if AT is a member of the resulting bag with IN. FLAT has the same definition as in the earlier case study to simplify this example. IN has an unnecessary conditional test, (EQUAL L (LIST X)), and an incorrect recursive call.

The algorithm recognition process and its results are shown in Figure 4-11. There are two algorithms represented in the task representation of the MEMTREE task. The first algorithm, called TREE-WALK, performs a tree-walk over the input to search for the atom. The second algorithm, called MEMTREE-FLATTEN, flattens the input and searches for the atom in the flattened bag. It consists of three functions: FLATTEN to flatten the input, MEMBER to test for set membership, and the top-level calling function MEMTREE that calls these two functions. The results of the algorithm process are:
Algorithm used: MEMTREE-FLATTEN.

Student Fns Matched to Reference Fns:
- FLAT to FLATTEN.
- IN to MEMBER.
- MEMTR to MEMTREE.

Formal Variable Mappings:
- FLAT to FLATTEN: (TR/TREE).
- IN to MEMBER: (L/BAG, X/ITEM).
- MEMTR to MEMTREE: (CONS/TREE, AT/ITEM).

---

**Algorithm: MEMTREE-FLATTEN**

**Function and Formal Variable Mappings:**
- MEMTR/MEMTREE \textit{vars} AT/ITEM, CONS/TREE
- FLAT/FLATTEN \textit{vars} TR/TREE
- IN/MEMBER \textit{vars} L/BAG, X/ITEM

**Figure 4-11:** Algorithm Recognition in MEMTREE Example

---

FLAT is debugged as before. IN is debugged similarly, by comparing it to the reference function MEMBER.
(DEFUN MEMBER (ITEM BAG)
  (IF (NLISTP BAG)
    F
    (IF (EQUAL ITEM (CAR BAG))
      T
      (MEMBER ITEM (CDR BAG)))))

The verification conditions and hints generated for the function IN are shown below:

(IMPLIES (NLISTP L) (IFF F NIL))
--> true

(IMPLIES (AND (NOT (NLISTP L))
  (EQUAL X (CAR L))
  (EQUAL L (LIST X)))
  (IFF T L))
--> true

(IMPLIES (AND (NOT (NLISTP L))
  (EQUAL X (CAR L))
  (NOT (EQUAL L (LIST X)))
  (IFF T L))
--> true

(IMPLIES (AND (NOT (NLISTP L))
  (NOT (EQUAL X (CAR L)))
  (IFF (IN X (CDR L))
  (IN X (CAR L))))
--> false

The first three conjectures are theorems while the last is not indicating a bug in the recursive call in IN. MEMTR, the remaining function, has no bugs when compared to MEMTREE. The final debugged code is shown below:

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

(DEFUN FLAT (TREE)
  (IF (LISTP TREE)
    (IF (ATOM (CAR TREE))
      (CONS (CAR TREE) (FLAT (CDR TREE)))
      (APPEND (FLAT (CAR TREE))
      (FLAT (CDR TREE)))
      (CONS TREE NIL)))

(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 partial matching of the algorithm recognition process occurs on abstract computational features of the student and reference functions. No graph matching on code structure occurs at any time. Student and reference functions are represented by frames called E-frames. The "E" stands for enumeration since E-frame slots represent abstract properties of the enumeration of recursively defined data structures. Slots represent:

- **When a function terminates or calls itself recursively.**
- **When and how variables are updated.**
- **Inferred variable and output datatypes.**
- **Inferred variable role.** Whether variables are used to accumulate results or to control function termination.
- **The datatype of the data structure enumerated.** One of TREE, LIST, or NUMBER.
- **The position of the function in the static calling structure of the solution.**
- **Whether or not side-effects can occur.**

- **Task Role.** The role which the function plays in implementing the algorithm. One of TOP-LEVEL, MAIN, SUPPORTING-CONSTRUCTOR, SUPPORTING-PREDICATE, or EXTRA as inferred from examining the functions and the static calling structure of the solution.

- **Recursion Type.** Whether the function is recursive (RECURSIVE), calls only primitives (SIMPLE), or calls other user defined functions (CALLING).

For example, the E-frame for the reference function FLATTEN is shown below:

```
Function Name - FLATTEN

Formals - (TREE)

Definition - (DEFUN FLATTEN (TREE)
  (IF (ATOM TREE)
    (LIST TREE)
    (APPEND (FLATTEN (CAR TREE))
          (FLATTEN (CDR TREE))))

Terminations - When (ATOM TREE) Return (LIST TREE);

Recursions - When (NOT (ATOM TREE)) Call (FLATTEN (CAR TREE));
             When (NOT (ATOM TREE)) Call (FLATTEN (CDR TREE));

Constructions - When (NOT (ATOM TREE))
              Return (APPEND (FLATTEN (CAR TREE))
                      (FLATTEN (CDR TREE)));

Variable-Updates - When (NOT (ATOM TREE)) Update TREE To (CAR TREE);
                  When (NOT (ATOM TREE)) Update TREE To (CDR TREE);

Variable-Data-Types - TREE Should-Be CONS
```
Algorithm recognition is implemented as a best first search through a space of function mappings. Each node in the search space is a mapping of student to reference functions for one of the stored algorithms. A heuristic evaluation scores each function for mapping. The scores are a weighted sum of the differences of the E-frame slots for paired functions. Both student and reference functions can be mapped to EXTRA, resulting in additional penalties to be added into the score. A student function mapped to EXTRA is interpreted as superfluous to a correct solution. A reference function mapped to EXTRA is interpreted as necessary to a correct solution but missing from the student’s solutions. The heuristic evaluation function and other means that Talus uses to account for alternate functional decompositions are described in [Murray 86].

5. Performance

This section addresses performance issues and discusses an empirical evaluation of Talus. Talus performs well debugging tasks of the size and complexity shown below. The task, taken from an undergraduate programming languages survey course, is described below:

Design a function (SINGLETONS X) which takes as argument an s-expression X and produces as a result a list of all atoms other than NIL which occur exactly once in X.

Use a function (FLATTEN X), where the argument X is an s-expression with subexpressions nested to any depth, such that the result of (FLATTEN X) is just a list of atoms with the property that all atoms other than NIL appearing in X also appear in (FLATTEN X).

Talus finds no bugs in the (correct) solution shown below:
(DEFUN MEMBER (EL SET)
 (COND ((NULL SET) F)
 ((EQUAL (CAR SET) EL) T)
 (T (MEMBER EL (CDR SET))))
)

(DEFUN FLATTEN (X)
 (COND ((NULL X) NIL)
 ((ATOM X) (LIST X))
 (T (APPEND (FLATTEN (CAR X))
 (FLATTEN (CDR X))))))

(DEFUN REMOVE (A X)
 (COND ((NULL X) NIL)
 ((EQUAL A (CAR X))
 (REMOVE A (CDR X)))
 (T
  (CONS (CAR X)
  (REMOVE A (CDR X))))))

(DEFUN ELIM-DUPL2 (L S D)
 (COND ((NULL L) S)
 ((MEMBER (CAR L) D)
 (ELIM-DUPL2 (CDR L) S D))
 ((MEMBER (CAR L) S)
 (ELIM-DUPL2
  (CDR L)
  (REMOVE (CAR L) S)
  (CONS (CAR L) D)))
 (T (ELIM-DUPL2
  (CDR L)
  (CONS (CAR L) S)
  D)))))

(DEFUN SINGLETONS2 (X)
 (ELIM-DUPL2 (FLATTEN X) NIL NIL))

The top-level calling function is SINGLETONS2. FLATTEN is used to flatten the input and remove all occurrences of NIL. ELIM-DUPL2 is the main function. L is the list left, S is a list of those atoms that occur singly up to this point in the list, and D is a list of those atoms that have duplicates. MEMBER is used to see if the CAR of the list left occurs in either S or D. If it is in S then it is deleted from S by calling REMOVE and then placed in D. It is in D then it is merely skipped over. Otherwise it is placed in S.

The task representation of the SINGLETONS task contains representations for 7 correct algorithms and 3 buggy algorithms. "Buggy algorithms" are algorithms where the program design itself is faulty, perhaps because the student misread or misunderstood the task assignment. For example, in the SINGLETONS task described above many students implement programs that flatten the input and then return a unique occurrence of every atom in the resulting bag. This program design does not solve the assigned task which requires that only those atoms that occur uniquely be returned. An example of such
a solution, obtained from student data, is shown below:

```lisp
(DEFUN SINGLETONS1 (X)
  (COND ((NULL X) NIL)
        ((ATOM X) (LIST X))
        (T (SINGFLAT1 (SMASH X))))
)

(DEFUN SMASH (S)
  (COND ((NULL S) NIL)
        ((ATOM S) (CONS S NIL))
        (T (CONS (SMASH (CAR S)) (SMASH (CDR S))))))
)

(DEFUN SINGFLAT1 (X)
  (COND ((ABSENT (CDR X) (CAR X))
        (APPEND (SINGFLAT1 (CDR X)) (LIST (CAR X))))
        (T (SINGFLAT1 (CDR X))))
)

(DEFUN ABSENT (L A)
  (COND ((NULL L) T)
        ((EQUAL A (CAR L)) NIL)
        (T (ABSENT (CAR L) A))))
)
```

Talus identifies the student’s algorithm as being one of the buggy algorithms in the SINGLETONS task representation. The following commentary is generated by Talus to address the student’s misconception:

> Looks like you misunderstood the task. The function SINGLETONS1 flattens its input and passes the resulting bag to the function SINGFLAT1. The function SINGFLAT1 uniquifies its input. The result is that one copy of every atom in the input to SINGLETONS1 will be returned. This is not the same as returning only those atoms that occur exactly once.

For example,

```lisp
(SINGFLAT1 '(A B A C C E)) should be (B E), not (A B C E).
```

Talus performs well on pure LISP solutions to tasks of the size of SINGLETONS. Performance begins to degrade as imperative programming constructs (i.e. PROG with SETQ and GO) or side-effects (i.e. RPLACA or PUTPROP) are introduced. This degradation is not surprising since this debugging approach is based on program verification and verification of programs with these constructs is also more difficult. No experiments have been done on tasks larger than SINGLETONS. It is expected that performance in algorithm recognition will degrade as task size is increased, unless additional sources of knowledge are introduced to reduce the search space. Additional domain knowledge or user query could compensate for the increased search space. Presently, Talus is fully automatic and does not require user query.

An empirical evaluation of Talus was performed, using student data collected in class and from
take-home assignments. 106 solutions were collected to 5 different tasks. The tasks ranged in complexity from simple tasks such as FLATTEN to larger tasks such as SINGLETONS. The solutions contained a total of 176 function definitions with a total of 205 formal variables.

Performance in each of the three steps of the debugging process was evaluated separately, as shown below:

<table>
<thead>
<tr>
<th>Algorithm Recognition</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely Correct</td>
<td>97</td>
<td>91.5%</td>
</tr>
<tr>
<td>Incorrect</td>
<td>9</td>
<td>8.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bug Detection</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>Detected</td>
<td>120</td>
<td>92.3%</td>
</tr>
<tr>
<td>Not Detected</td>
<td>10</td>
<td>7.7%</td>
</tr>
<tr>
<td>False Alarms</td>
<td>3</td>
<td>2.3%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bug Correction</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempted Edits</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>Succeeded</td>
<td>118</td>
<td>98.3%</td>
</tr>
<tr>
<td>Failed/Incorrect</td>
<td>2</td>
<td>1.7%</td>
</tr>
</tbody>
</table>

Each phase assumes correct performance in the previous stage of debugging. Thus, errors in bug detection are only counted if algorithm recognition is correct. Similarly, errors in bug correction are counted only if both algorithm recognition and bug detection is correct. In all three phases of debugging, performance is greater than 90%.

Bugs not detected and false alarms can be attributed to:

- *Heuristics in the Algorithm Recognition Process*. These are, by definition, imperfect.

- *Missing Counterexamples to Invalidate Conjectures*. Only the conjecture disprover was used to evaluate conjectures, and in 6 out of 106 solutions stored examples were insufficient to invalidate false conjectures.

- *Misapplication of Program Simplification Transforms*. Talus rewrites student programs to computationally equivalent programs that are easier to analyze prior to debugging. For example, COND is rewritten to IF. Occasionally, but not frequently, this process backfires and programs are rewritten in a way that masks bugs in the original code.

- *Inability to Trace Edits through Program Simplification Transforms*. Talus can usually invert bug fixes from simplified code back to original code. However, some edits cannot be successfully edited back to the original code due to the procedural nature of some of the transformations.

However in spite of these problems the overall performance of Talus compares favorably with other debugging systems that have been empirically evaluated, such as PROUST [Johnson 85].
6. Conclusion

This section first summarizes the capabilities and limitations of Talus. Next the program reasoning capabilities of Talus are discussed to support the thesis that ultimate debugging capabilities depend on the ability to reason about computational semantics. Finally, Section 6.3 summarizes the specific contributions of this research.

6.1. Capabilities and Limitations of Talus

Talus addresses issues in automated debugging that are not addressed in related work on this subject. These issues include:

- **Unanticipated Implementations.** No plan library and bug catalog is necessary for bug detection and correction. Although it can be argued that stored algorithm representations constitute a plan and bug library, these are only necessary for the algorithm recognition phase and are not used to account for implementation variability within functions. In contrast the plan-difference rules of other systems (e.g. PROUST [Johnson 83]) have rewrite rules that account for implementation variants within procedures. For example PROUST has rules that establish that the boolean expressions "X > Y" and "Y < X" are equivalent, or that the arithmetic expressions "X + Y" and "Y + X" are equivalent. Talus has no such rules.

- **Detection of Syntactic, Stylistic, and Nonstylistic Bugs.** Most systems based on dynamic analysis (e.g. PDS6 [Shapiro 83]) do not detect stylistic errors.

- **Correction of Nonstylistic Bugs.** Missing conditional tests are inserted into buggy code, and buggy code fragments are replaced. Very few implemented automated debuggers can determine bug fixes.

- **Multiple Procedures and Algorithm Recognition.** Talus allows solutions to tasks to contain multiple procedures, each of which can have one or more formal variables. In contrast most other debugging systems only allow a single procedure per solution. With this restriction the problem of determining the teleology of individual procedures within a solution does not surface. Thus Talus addresses issues in algorithm recognition that other systems do not.

- **Limited Handling of Side-Effects.** Some debuggers, such as PDS6, do not allow side-effects to program state during program execution. Although the ability to handle side-effects in Talus is limited, constructs such as RPLACA, RPLACD, NCONC, PUTPROP, GET, and REMPROP are permitted.

- **Debugging Recursive Programs.** Many systems cannot debug recursive programs whereas Talus is specifically designed to debug recursive programs.

All automated debuggers are limited since no scheme for bug diagnosis is infallible. If there were then the Halting Problem would be solved since nontermination is one of the bugs to be detected. The most salient limitations of Talus appear to be its ability to debug programs with side-effects and imperative programming constructs. Other limitations relating to conjecture evaluation have been discussed in Section 4.1.3 and Section 5.
6.2. Reasoning about Computational Semantics in Talus

The complexity of the debugging process can obscure the program reasoning capabilities of Talus. These capabilities allow Talus to permit significant variability in student code while providing fully automated algorithm recognition, bug detection, and bug correction. Three kinds of reasoning occur:

- **Heuristic Reasoning.** Heuristic reasoning occurs during algorithm recognition. It is very difficult to formalize what it means for a student program to appear more similar to the implementation of one algorithm than another. Heuristic evaluation functions and search are required to map functions and formal variables prior to bug detection and correction.

- **Conjecture-Based Reasoning.** This more formal kind of reasoning, based on the evaluation of conjectures in the Boyer-Moore Logic, occurs during the bug detection and correction phases. Conjectures are evaluated to reason about:
  - *Functional Equivalence.* Program fragments are compared to see if they are implementation variants.
  - *Logical Implications.* Symbolic evaluation is performed by determining paths through trees (as in Figures 4-7 and 4-8) based on logical implications. Symbolic evaluation is required to perform a case analysis, construct verification conditions, and detect anomalous conditional tests in student functions.
  - *Violations of Well-Founded Relations.* Talus also generates another kind of verification condition, not previously discussed, that checks that recursive calls in student functions lead to function termination ultimately. These termination verification conditions are discussed in [Murray 86].

- **Procedural Reasoning about LISP Semantics.** Procedurally encoded knowledge is used in:
  - *Parsing Functions into E-frames.* The procedures that determine slot values examine LISP functions to determine their computational features.
  - *Analyzing Functions to See if they Return Fresh List Structure.* One way of handling side-effects is to determine if functions such as NCONC are applied to freshly CONSed list structure. If so, then nondestructive equivalents can replace them (e.g., APPEND could replace NCONC) prior to bug detection and correction.
  - *Program Simplification Transforms and Edit Inversion.* Some constructs such as PROG are simplified by rewriting student functions into equivalent functions not containing the construct. Then debugging is performed on the simplified functions and the bug fixes are regressed back through the program transformations to the original functions. The application of these program simplification transforms and the inversion of edits through them is performed by procedures specific to the transforms.

6.3. Research Contributions

This work contributes to research in program analysis, independent of the ITS application proposed here. The E-frame program representation contrasts sharply with the PLAN representation of programs in the Programmer's Apprentice Project. The PLAN representation is a flow-graph representation of a

---

8Such fresh list structure would have a reference count of 0.
program's data and control flow. The E-frame representation also represents control and data flow, but more abstractly. Furthermore, the E-frame representation captures abstractions other than data and control flow in its abstract characterization of recursive functions. A function's E-frame represents features such as the data structure enumerated and constructed, the inferred variable datatypes and roles, and function terminations, recursions, and variable updates. These could be inferred from the PLAN representation but they are not explicitly represented at present.

The algorithm recognition process of Talus also differs from that proposed for the Programmer's Apprentice [Brotsky 84]. Algorithm recognition in the Programmer's Apprentice is based on a bottom-up parsing of flow-graphs using a web-grammar. This approach generalizes parsing with a context-free grammar from strings to graphs. In contrast, algorithm recognition in Talus is performed by searching through a space of function mappings with no graph matching required at any time. For the particular application of recognizing algorithms in buggy student programs this latter approach seems less sensitive to the presence of bugs in student programs. The algorithm recognition process of the Programmer's Apprentice requires subgraph matching. When partial matching is introduced to allow for bugs in student programs, there is a high potential for combinatorial explosion in the matching process, even when only simple programs are considered.

On the other hand, the Programmer's Apprentice Project is intended for a wider range of applications and can support deeper kinds of program reasoning than Talus. For example, the Programmer's Apprentice can determine the results of program modifications whereas Talus cannot. The point here is that the program representation and algorithm process of Talus may find other applications in program analysis than solely in the automated debugging system implemented.

Another research contribution of Talus, independent of its particular program verification debugging approach, is the taxonomic classification of debugging approaches. This classification, shown in Figure 2-1, can be applied to characterize trade-offs in the use of one debugging approach compared to another. For example, the analysis by synthesis approach can be more easily adapted to the generation of hints regarding program design than the heuristic plan-recognition approach. On the other hand the latter approach is more tractable for domains where program synthesis is difficult or where it is difficult to track or infer the student's program design decisions. Further discussion of other trade-offs appears in [Murray 86].

The role of computational semantics in the debugging process has not been addressed previously. This research stresses the importance of program reasoning as being just as important as inferring code teleology. While this may appear obvious in retrospect, the issue of explicit reasoning about computational semantics has been largely ignored in previous work in automated debugging.\(^9\)

---

\(^9\)A notable exception is Ehud Shapiro's research on interactive bug diagnosis and correction by trace analysis and program synthesis. That research concentrates on analysis of program execution while the research presented here emphasizes other aspects of program reasoning such as algorithm recognition and distinguishing correct and incorrect implementation variants in source code.
The major part of this research has been the investigation of a little explored approach to automated debugging based on program verification. A design has been developed and implemented for the domain of debugging novice LISP programs. An empirical evaluation has been conducted that demonstrates that this approach compares favorably with related work in this field. Finally, this approach has been characterized in terms of its advantages and disadvantages compared to alternate approaches to automated debugging.

Acknowledgements

I would like to thank Bruce Porter, Elaine Rich, and Mark L. Miller for the invaluable help they have provided in this research. Their time and contributions are deeply appreciated. I would also like to thank Woody Bledsoe and Chris Lengauer for their assistance. Finally, I am grateful to J Moore and Bob Boyer for help understanding their logic and theorem prover. Key ideas in this research can be traced to their work in automatic theorem proving.
References

[Adam 80] A. Adam and J. Laurent. 
LAURA, A System to Debug Student Programs. 

[Barr 76] A. Barr, M. Beard, and R. Atkinson. 
The Computer as a Tutorial Laboratory. 

[Brotsky 84] D. C. Brotsky. 
*An Algorithm for Parsing Flow Graphs*. 

[Goldstein 74] I. Goldstein. 
*Understanding Simple Picture Programs*. 

[Goldstein 75] I. Goldstein. 
Summary of MYCROFT: A System for Understanding Simple Picture Programs. 

[Johnson 84] W. L. Johnson and E. Soloway. 
Intention-based Diagnosis of Programming Errors. 

[Johnson 85] W. L. Johnson. 
*Intention-Based Diagnosis of Errors in Novice Programs*. 


Logical Analysis of Programs. 

EGS: A Transformational Approach to Automatic Example Generation. 

A Structured Planning and Debugging Environment for Elementary Programming. 

*Intelligent Tutoring for Programming Tasks*. 
Heuristic and Formal Methods in Automatic Program Debugging.

*Automatic Program Debugging for Intelligent Tutoring Systems.*
Technical Report AI TR86-27, University of Texas at Austin, Artificial Intelligence Laboratory, June, 1986.

Dynamic Student Modelling in an Intelligent Tutor for Lisp Programming.

*Analysis of Algorithm Implementations.*

*Dictionary of Pronunciation.*

[Shapiro 81] D. Shapiro.
Sniffer: A System that Understands Bugs.
MIT A.I. Memo 638. 1981

*Algorithmic Program Debugging.*
MIT Press, MIT, 1983.

MENO-II: An AI-Based Programming Tutor.