MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1100 January 1989

Intelligent Assistance for Program Recognition, Design, Optimization, and Debugging

by

Charles Rich and Richard C. Waters

Abstract

We describe research in four related areas, based on the following theoretical principles: the assistant approach (incremental automation) and the exploitation of clichés (using knowledge of common engineering practice).

Each investigation involves the construction of a prototype system to provide intelligent assistance for a person performing the task: A recognition assistant will help reconstruct the design of a program, given only its source code. A design assistant will assist a programmer by detecting errors and inconsistencies in his design choices and by automatically making many straightforward implementation decisions. An optimization assistant will help improve the performance of programs by identifying intermediate results that can be reused. A debugging assistant will aid in the detection, localization, and repair of errors in designs as well as completed programs.

These prototypes will be constructed using two shared technologies: a programming language independent formal representation for programs and programming knowledge (the Plan Calculus) and an automated reasoning system (CAKE), which supports both general logical reasoning and special-purpose decision procedures.

Revised version of proposal submitted to the National Science Foundation.

The research described here was conducted at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the following organizations: National Science Foundation under grant IRI-8616644, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, Microelectronics and Computer Technology Corporation, and Siemens Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the policies, expressed or implied, of these organizations.

Contents

Acknowledgements

This paper draws in large part on the work of the following students in the Programmer's Apprentice group: Linda Wills (recognition), Yang Meng Tan (design), Robert Hall (optimization) and Ron Kuper (debugging).

1 Introduction

This paper describes new and ongoing research in the Programmer's Apprentice project [49]. Each of the following four sections describes a three-year investigation into automating an aspect of the programming task within the framework of the assistant approach and the exploitation of clichés. Each investigation also involves the construction of a prototype system using the shared technologies of the Plan Calculus and CAKE.

In the area of program recognition, we have demonstrated the essential feasibility of automatically recognizing programming clichés in undocumented source code, using the Plan Calculus as an intermediate language. It is clear however, that the completely automatic, bottom-up graph-parsing approach we have developed will not scale up to programs of realistic size—additional guidance will have to be provided, for example, by a person. In order to arrange this division of labor appropriately, a key question to be answered is what are the fundamental limits of bottom-up recognition.

In the area of program design, we are building a prototype Design Apprentice, which will assist a programmer in the process of detailed design by detecting errors and inconsistencies in his design choices and automatically making many straightforward implementation decisions. The Design Apprentice extends our earlier work (KBEmacs), which used the Plan Calculus and a library of clichés to partially automate program implementation. This extension is made possible by the availability of CAKE to provide the necessary automated deduction facilities.

In the area of program optimization, our work is motivated by the fact that constructing software using clichés (or more generally, any kind of reuse) often sacrifices run-time efficiency. Our solution to this problem is to develop a powerful program optimization system, using the Plan Calculus as the intermediate language and using CAKE to verify the necessary properties. As in the case of program recognition, we plan to explore the limits of completely automatic optimization, as a prelude to applying the assistant approach.

Finally, in the area of program debugging, we have demonstrated the feasibility of using the Plan Calculus and the truth-maintenance facilities of CAKE to assist in the localization of bugs using test cases. We plan to extend and generalize this work in several directions, such as intelligent assistance for bug repair and the application of similar techniques to incomplete programs with partial specifications. We also plan to investigate the utility of bug clichés within this framework.

1.1 The Assistant Approach

One approach to solving current software problems is to totally eliminate programmers through "automatic programming." As typically conceived, automatic programming calls for an end user to write a complete specification for what he wants; a completely automatic system then generates a program satisfying this specification. Program generators of this type have been successfully developed for a number of narrow applications. However, completely automatic programming for a broad range of applications is not a realistic near-term goal [48].

An alternate approach is to assist programmers, rather than replace them. A provocative

example of the assistant approach was proposed by IBM's Harlan Mills in the early 1970's. He suggested creating "chief programmer teams" by surrounding expert programmers with support staffs of human assistants, including junior programmers, documentation writers, program librarians, and so on. The productivity of the chief programmer was thereby increased, because he could apply his full effort to the most difficult parts of a given software task without getting bogged down in the routine details that currently use up most of every programmer's time. Experience has shown that this division of labor can be very successful. The long-term goal of our project is to provide *every* programmer with a support team in the form of an intelligent computer program, called the Programmer's Apprentice.

The assistant approach lends itself to incremental progress. Initially, the Apprentice will be able to take over only the simplest and most routine parts of the programming task. As technology advances, however, the proportion of the task handled by the Apprentice will increase—we don't have to wait until some activity can be totally automated. Furthermore, some parts of a given program may always have to be hand coded due, for example, to extremely strict performance requirements. However, especially in large systems, there is always plenty of routine work to be done.

The key to making the assistant approach work is the communication between the programmer and the Apprentice: It must be based on a substantial body of shared knowledge of programming techniques. If the programmer had to explain everything to the Apprentice in elementary terms, it would be easier to do the work himself. This leads us to the second principle.

1.2 Clichés

Human programmers seldom think only in terms of primitive elements, such as assignments and tests. Rather, like engineers in other disciplines, they mostly think in terms of commonly used combinations of elements. We call these familiar combinations *clichés*. *Successive approximation*, *device driver*, and *information system*, are examples of programming clichés spanning the range from low-level implementation ideas to high-level specification concepts. (Soloway and Ehrlich [56] have conducted a number of empirical studies with programmers that support the psychological reality of programming clichés.)

In general, a cliché consists of *roles* and *constraints*. The roles of a cliché are the parts that vary from one occurrence of the cliché to the next. The constraints are used to specify fixed elements of structure (parts that are present in every occurrence), to check that the parts that fill the roles in a particular occurrence are consistent, and to compute how to fill empty roles in a partially specified occurrence of a cliché.

An essential property of clichés is their relationship to one another. For example, a cliché may be a special case or an extension of another cliché. Algorithmic and data structure clichés may be related as possible implementations of specification clichés.

Given a library of clichés, it is possible to perform many programming tasks *by inspection*, rather than by reasoning from first principles. For example, in analysis by inspection, properties of a program are deduced by recognizing occurrences of clichés and referring to their known properties. In synthesis by inspection, implementation decisions are made by recognizing clichés in specifications, and then choosing among various implementation

clichés.

Our work focuses on the use of inspection methods to automate programming, as opposed to more general, but harder to control methods, such as deductive synthesis or program transformations. In human programming, inspection methods are the most effective approach to use whenever they are applicable. However, since inspection methods are ultimately based on experience, they are applicable only to the routine parts of programming problems. This is compatible with the intended division of labor between the programmer and the Apprentice.

As we will see in the upcoming sections, codifying clichés is a central activity in the project. We are building libraries of clichés in the areas of program implementation, design, and requirements. We will also see how a shared vocabulary of clichés serves as the language of communication between the programmer and the Apprentice.

1.3 The Plan Calculus

The Plan Calculus is a formal representation for programs and programming clichés. Since this formalism is now quite well known, we will summarize only its key properties here. (Program representations related to or derived from the Plan Calculus have been used by many others for work in program recognition [21, 34], tutoring [16, 30], translation [18, 60], algorithm design [26, 27], debugging [54, 33], and maintenance [35].)

To a first approximation, the Plan Calculus can be thought of as combining the representation properties of flowcharts, data flow schemas, and abstract data types. A *plan* is essentially a hierarchical graph structure made up of different kinds of boxes (denoting operations and tests) and arrows (denoting control and data flow). The representation has both a graphical notation [46] and a formal semantics used for reasoning [44, 43].

The major advantage of the Plan Calculus is that it abstracts away from the details of algorithms and data structures that are a result only of their expression in a particular programming language. The explicit representation of control and data flow in the Plan Calculus greatly simplifies all of the manipulations that need to be performed for intelligent programming assistance, as well as making it possible to implement assistants that are, to a large degree, programming-language independent. For example, a data flow arc between two operations abstracts away from whether a temporary variable was used, or whether the producing operation was nested inside the invocation of the consuming operation. Similarly, the same net control flow graph can be achieved by many different combinations of control primitives.

The Plan Calculus is a wide-spectrum formalism. Each operation and test in a plan has associated with it a set of preconditions and postconditions specified in a logical language. A plan in which all of the operations and tests are primitives in some programming language is equivalent to a concrete program. However, the Plan Calculus is also used to represent partially designed programs and programming clichés, in which case the operations and tests may may have abstract or incomplete specifications, and there may be arbitrary logical constraints, as well as data and control flow.

Taxonomic relationships between clichés, such as specialization, are handled by special-purpose mechanisms in the cliché library. The relationship between a specification and an implementation is represented in the Plan Calculus by an *overlay*. Formally, an overlay

defines a mapping from the set of instances of the implementation plan to the set of instances of the specification. (This is a generalization of the *abstraction function* in the abstract data type methodology.) A cliché library may contain different overlays with the same domain and/or range, corresponding to different ways of abstracting the same implementation and/or different ways of implementing the same specification. An key feature of overlays is that they are used for both analysis and synthesis.

1.4 A Hybrid Reasoning System

The degree of intelligent assistance the Apprentice can provide ultimately depends on its ability to reason about structured objects (programs, specifications, requirements) and their properties. Our approach to this reasoning task is to use a combination of special-purpose techniques and general-purpose logical reasoning.

Special-purpose representations and algorithms are essential to avoid the combinatorial explosions that typically occur in general-purpose logical reasoning systems. On the other hand, logic-based reasoning is very valuable when used, under tight control, as the "glue" between inferences made in different special-purpose representations.

We have developed a hybrid knowledge representation and reasoning system, called CAKE [45, 19, 20], that we are using for all current work in the project. Figure 1 shows the architecture of CAKE. CAKE combines special-purpose representations, such as frames and the Plan Calculus, with general-purpose logical and mathematical reasoning. Each layer of CAKE builds on facilities provided by the layers below.

Figure 2 is a short transcript from the currently running version of CAKE, illustrating some of the facilities provided in the propositional, algebraic, and frame layers. (Line numbers in the following discussion refer to Figure 2.)

The propositional layer of CAKE provides three principal facilities. First, it automatically performs simple "one-step" deductions (lines 1–3) (technically, unit propositional resolution). The use of very limited form of logical inference here is an example of what we mean by keeping tight control over general-purpose mechanisms.

Second, the propositional layer acts as a recording medium for dependencies (what is often called a truth-maintenance system), and thus supports explanation (line 3) and retraction (lines 4–5). These facilities are motivated by the observation that when you delegate work to an assistant, you also need to have accountability and the ability to recover from mistakes, in case it doesn't do what you expected.

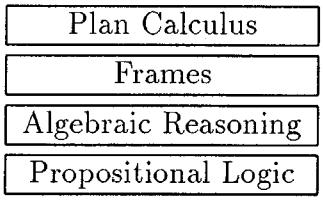

Figure 1. CAKE has a layered architecture.

```
 1>  (Assertq P)
 2>  (Assertq (Implies P Q))
 3>  (Whyq Q)
     Q is TRUE by Modus Ponens from:
       1. (IMPLIES P Q) is TRUE as a premise.
       2. P is TRUE as a premise.
 4>  (Retractq P)
 5>  (Whyq Q)
     I don't know whether or not Q is true.
 6>  (Assertq (And P (Not Q)))
     >>Contradiction: There is a conflict between the premises:
         1. (AND P (NOT Q)) is TRUE.
         2. (IMPLIES P Q) is TRUE.
     Type cr to postpone dealing with this contradiction.
     Type premise number to retract one of the premises.
 7>  1
     Retracting (AND P (NOT Q)) being TRUE...
     #<Node (AND P (NOT Q)): False>
 8>  (Assertq (= I J))
 9>  (Whyq (= (F I) (F J)))
     (= (F I) (F J)) is TRUE by Equality from:
       1. (= I J) is TRUE as a premise.
10>  (Assertq (Transitive R))
11>  (Assertq (R W X))
12>  (Assertq (R X Y))
13>  (Assertq (R Y Z))
14>  (Whyq (R W Z))
     (R W Z) is TRUE by Transitivity from:
       1. (R W X) is TRUE as a premise.
       2. (R X Y) is TRUE as a premise.
       3. (R Y Z) is TRUE as a premise.
       4. (TRANSITIVE R) is TRUE as a premise.
15>  (Assertq (Subset A B))
16>  (Assertq (Member X A))
17>  (Whyq (Member X B))
     (MEMBER X B) is TRUE by Subsumption from:
       1. (SUBSET A B) is TRUE as a premise.
       2. (MEMBER X A) is TRUE as a premise.
18>  (Deftype Address (:Specializes Number))
19>  (Deframe Interrupt
       (:Roles (Location Address) Program))
20>  (Deframe Device
       (:Roles (Transmit Address) (Receive Address)))
21>  (Deframe Interface
       (:Roles (Target Device) (From Interrupt) (To Interrupt))
       (:Constraints (= (Location ?From) (Receive ?Target))
                     (= (Location ?To) (Transmit ?Target))))
22>  (FInstantiate 'Interface :Name 'K7)
23>  (FPut (>> 'K7 'Target 'Receive) 777777)
24>  (FGet (>> 'K7 'From 'Location))
     777777
25>  (Why ...)
     (= 777777 (LOCATION (FROM K7))) is TRUE by Equality from:
       1. (= (LOCATION (FROM K7))
             (RECEIVE (TARGET K7))) is TRUE.
       2. (= (RECEIVE (TARGET K7)) 777777) is TRUE as a premise.
```

Figure 2. A transcript from the currently running version of CAKE, illustrating the reasoning capabilities of the propositional, algebraic, and frame layers.

Third, the propositional layer detects contradictions (lines 6–7). Furthermore, contradictions are represented explicitly in such a way that reasoning can continue with other information not involved in the contradiction. This feature is motivated by our desire to support an evolutionary programming process. In this kind of process, the programmer's knowledge is very often in an inconsistent state, particularly during the requirements acquisition and analysis phase.

The algebraic layer of CAKE contains special-purpose decision procedures for equality reasoning, common algebraic properties of operators (such as commutativity, associativity, and transitivity), partial functions, and the algebra of sets. The congruence closure algorithm in this layer determines whether or not terms are equal by substitution of equal subterms (lines 8–9). The decision procedure for transitivity (lines 10–14) determines when elements of a binary relation follow by transitivity from other elements. The algebra of sets (lines 15–17) involves the theory of membership, subset, union, intersection and complements.

Equality reasoning is very important in CAKE, because the formal semantics of the Plan Calculus makes heavy use of equality. Data flow arrows in plans imply equalities between terms representing the source and destination points; correspondences in overlays are also equalities. Other algebraic properties, such as transitivity, commutativity, etc., come up everywhere in the formal modeling of data structures.

The frames layer of CAKE supports the standard frame notions of inheritance (`:Specializes` in line 18), slots (`:Roles` in lines 19–21), and instantiation (line 22). The organization of cliché libraries is based on frame inheritance.

A notable feature of CAKE's frame system is that constraints are implemented in a general way. For example, the definition of the *interface* frame (line 21) has constraints between the roles of the instances filling its roles. When an instance of an interface is created (line 22) and a particular value (777777) is put into one of its "second level" roles (line 23), the same value can be retrieved from the other constrained role (line 24). This propagation is not achieved by *ad hoc* procedures, but by the operation of the underlying logical reasoning system, including dependencies (line 25). Constraint propagation makes it possible to incrementally acquire information in any order.

The Plan Calculus layer of CAKE supports graph-theoretic manipulations of plan diagrams and overlay diagrams, such as following arcs. It also implements the formal semantics of the Plan Calculus, so that hybrid reasoning can take place involving both structure (as expressed by the diagrams) and function (as expressed in the preconditions, postconditions, and other logical annotations). In the semantics of the Plan Calculus, names of plans become predicate symbols, names of roles and overlays become function symbols, correspondences become equalities, data flow becomes a combination of equalities and a partial order, and control flow becomes a combination of an equivalence relation and a partial order.

2 Recognition

Program recognition is the process of identifying occurrences of clichés in a program. The result of the process is a hierarchical description of the program that can be interpreted as its design in terms of a given library of clichés (in general, there may be several possible such descriptions). Recognition is an important task to study for both practical and theoretical

reasons.

The main practical motivation for studying program recognition is the observation that recognition is a prominent activity in program maintenance, which is currently the dominant software cost. In the long run, we advocate recording the information during design that will be needed for maintenance (see next section). However, in the meanwhile, there is a huge body of extant code being maintained, for which the design record is effectively unavailable. Automated program recognition could be of great assistance to program maintainers in this situation.

Program recognition is important from a theoretical standpoint because it provides an additional constraint on the representation of clichés. Earlier work on representing programming knowledge using source-to-source transformations (e.g., [5]) suffered from being overly biased toward using the knowledge only in the synthesis direction; the Plan Calculus supports both analysis and synthesis in the same representation.

Program recognition may also serve to enhance the power of techniques for other tasks, such as program optimization and debugging (see subsequent sections on these topics). For example, recognition could compensate for the absence of design information required for optimization. Similarly, debugging assistance may be enhanced by the ability to recognize clichés for incorrect programs.

As a simple example of the recognition process, consider the program in Figure 3. A prominent feature of this program is the clichéd usage of CAR, CDR, and NULL in the loop to enumerate the list in the variable BUCKET. The form of the COND test in the loop indicates that the loop is implementing a membership test for ELEMENT in BUCKET and that the BUCKET list is ordered. Finally, the way BUCKET is computed from STRUCTURE and ELEMENT using SXHASH indicates that STRUCTURE is a hash table and, therefore, that the program as a whole is testing for the membership of ELEMENT in this hash table.

```
(DEFUN HASH-TABLE-MEMBER (STRUCTURE ELEMENT)
  (LET ((BUCKET (AREF STRUCTURE (SXHASH ELEMENT)))
        (ENTRY NIL))
    (LOOP
      (IF (NULL BUCKET) (RETURN NIL))
      (SETQ ENTRY (CAR BUCKET))
      (COND ((STRING> ENTRY ELEMENT) (RETURN NIL))
            ((EQUAL ENTRY ELEMENT) (RETURN T)))
      (SETQ BUCKET (CDR BUCKET)))))
```

Figure 3. Undocumented Common Lisp program.

Like visual recognition, program recognition often appears effortless when people do it. However, program recognition has a number of features that make it difficult to automate:

- *Syntactic variation* — There are typically many different ways to achieve the same net flow of data and control (e.g., the program in Figure 3 could have had more or fewer temporary variables, or used DO instead of LOOP).

- *Implementation variation* — There are typically many different concrete algorithms that can be used to implement a given abstraction (e.g., the hash table in Figure 3 could have been implemented using a hash/rehash scheme rather than bucket lists).
- *Diffuse structure* — Pieces of a cliché are sometimes widely scattered through the text of a program, rather than being contiguous. (As discussed in [31], this can be a significant source of difficulty for human program recognition as well.)
- *Unrecognizable structure* — Not all programs are completely constructed out of clichés. Recognition must be be able to ignore unrecognizable structure.

2.1 A Feasibility Demonstration

A prototype system has been implemented, which automates the recognition of clichés [61]. Given a program and a library of clichés, the recognizer finds all occurrences of the clichés in the program and builds a hierarchical description of the program in terms of the clichés found and the relationships between them. As shown in Figure 4, the system can present this description in the form of a textual explanation of the program design.

```
HASH-TABLE-MEMBER is a Set Membership operation.
   It determines whether or not ELEMENT is an element of the set STRUCTURE.
The Set is implemented as a Hash Table.
The Hash Table is implemented as an Array of buckets, indexed by hash code.
   The buckets are implemented as Ordered Lists.  They are ordered
   lexigraphically.  The elements in the buckets are strings.  An Ordered
   List Membership is used to determine whether or not ELEMENT is in the
   fetched bucket, BUCKET.
```

Figure 4. An explanation of the design of the program in Figure 3, automatically produced by the prototype program recognition system. Upper case (e.g., `ELEMENT`) indicates identifers in the program; initial capitalization (e.g., Set Membership) indicates the names of clichés in the library and their roles.

The architecture of the prototype recognition system is shown in Figure 5. The program to be analyzed is first converted into a plan. This translation deals with most of the syntactic variation among programs by mapping syntactically distinct programs to identical plans. Using the Plan Calculus also addresses the problem of diffuse structure, since many clichés are much more localized in a data and control flow graph than in program text.

The recognizer is able to deal with implementation variation by virtue of the fact that the cliché library contains not only clichés, but also information (in overlays) about how they implement one another.

The actual recognition is performed by means of a graph parsing algorithm [10], which has been extended to "skip over" unrecognizable sections of the input graph. The cliché library is treated as a graph grammar [17], in which each cliché defines a non-terminal. Overlays are treated as additional rules that capture implementation decisions. The output of the parsing algorithm is, in general, a set of derivations, which encode a hierarchical decomposition of

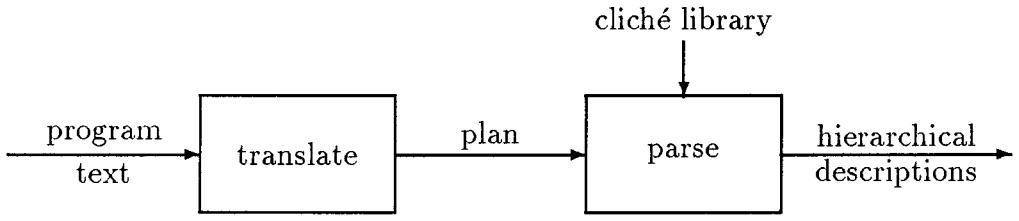

Figure 5. Architecture of the prototype recognizer.

the original program in terms of clichés and implementation decisions in the library. (There may be several different paths through the grammar that result in the same program.) By knitting together pieces of "canned" text attached to the clichés, it is straightforward to produce a textual explanation, such as Figure 4, from a derivation (see [14]).

Most other attempts to automate the program recognition process use an architecture that is fundamentally similar to Figure 5. However, they differ in their intermediate representation for programs and clichés, and in the way parsing is performed.

Most existing [25, 52] and proposed [32] recognition systems operate directly on program text. This limits the variability and complexity of the structures that can be recognized, because these systems must wrestle directly with syntactic variations, performing source-to-source transformations to twist the code into a recognizable form. Most of these systems' effort is expended on trying to canonicalize the syntax of a program, rather than recognizing its deeper semantic content.

The Laura system [2] uses a graphical representation for programs. These graphs, however, only represent control flow; assignment statements become nodes in the graph. As a result, this system does not handle syntactic variability in data flow any better than systems operating directly on source text.

A few existing [30] and proposed [21] systems have adopted plan-like intermediate notations. This aids them significantly in dealing with syntactic variations. However, like the other systems described above, these systems depend on heuristic techniques to recognize features of a program, rather than on the kind of systematic and exhaustive approach presented here. These heuristic approaches have advantages with regard to efficiency, but limit the power and extensibility of the systems.

One system that takes the same approach as the prototype recognizer we have built is Lutz's system [33, 34]. This system uses the Plan Calculus representation for programs and a graph parsing algorithm. The parsing algorithm is based on generalization of a chart parsing technique for strings.

2.2 Determining the Limits of Bottom-Up Recognition

Full scale, automated program recognition will probably require a hybrid approach, combining the kind of bottom-up recognition demonstrated above with top-down processing driven by expectations provided by a person or automatically gleaned from documentation (including comments). A key question in this architecture is just how much can be done bottom-up.

In the first year of the research, we plan to explore this question empirically. To do so, the prototype recognition system needs to be extended in several ways. In particular, the current prototype cannot yet handle recursion (other than tail recursion for loops), destructive operations on data structures, or incomplete programs. In addition, the prototype lacks some needed logical inference capabilities. All of these deficiencies are a result of the fact that this prototype was implemented on top of an earlier version of the Plan Calculus, rather than using CAKE. In addition, the cliché library will need to be expanded and a module needs to be implemented to automatically convert the library into an equivalent graph grammar (this is currently a manual process). Our goal for the first year is to begin to experiment with completely automatic recognition of programs in the order of hundreds to thousands of lines.

Our goal for the second year of the research is to come to a theoretical understanding of the limits of this approach. This will begin with the metering of specific experiments and culminate with an abstract characterization of the complexity of the full-scale system. This theoretical analysis will suggest, among other things, where the division of labor should be in a hybrid system. For example, if the complexity of the algorithm is dominated by the size of the grammar, then we will be led to consider how user guidance can be utilized to narrow down the set of clichés to be searched for at any particular time. The complexity analysis may also point out opportunities for utilizing parallel hardware (we have already started a small exploration in this direction [51]).

In the third year of the research, we plan to build a first prototype of a hybrid recognition system, which combines bottom-up algorithmic techniques with expectation-driven mechanisms, including user guidance.

3 Design

Our first prototype of a part of the Programmer's Apprentice, called KBEmacs (Knowledge-Based Editor in Emacs) [59, 58], focused on intelligent assistance for program implementation. KBEmacs allows a programmer to construct a program more rapidly and reliably than using a conventional program editor, by using the Plan Calculus as an internal representation of the algorithmic structure of a program and by providing a library of programming clichés. The research described in this section is motivated partly by problems uncovered in working on KBEmacs and partly by the desire to extend the range of assistance into low-level specifications and design.

3.1 What KBEmacs Can and Cannot Do

KBEmacs augments the existing text- and syntax-based commands of Emacs [57] with a new, higher, level of editing commands. Using KBEmacs, changes in the algorithmic structure of a program can be achieved by a single command, even when they correspond to widespread changes in the program text. In particular, an entire cliché can be merged into a program with a single high-level command. Also, the same high-level interface can be used to define new clichés. Due to the use of the Plan Calculus as an internal representation, KBEmacs is predominantly programming-language independent. Although the first running version of

the system was constructed to operate on Lisp programs, relatively little effort was required to extend it operate on Ada programs as well.

Figure 6 illustrates the power of KBEmacs. The commands in this figure refer to a number of implementation-level clichés such as *simple report*, *chain enumeration*, and *query user for key*. The terms *enumerator*, *main file key*, *title*, and *summary* are the names of roles in these clichés. (Our current library of implementation clichés is similar in scope and level to earlier machine-usable codifications of program implementation knowledge, such as [22]; however, as discussed earlier, the Plan Calculus provides a better representation of this knowledge.)

```
Define a simple_report program UNIT_REPAIR_REPORT.
Fill the enumerator with a chain_enumeration of UNITS and REPAIRS.
Fill the main_file_key with a query_user_for_key of UNITS.
Fill the title with ("Report of Repairs on Unit " & UNIT_KEY).
Remove the summary.
```

Figure 6. These 5 KBEmacs commands produce a 55 line Ada program.

Given that a programmer knows which clichés he wants to use, KBEmacs can be very helpful in taking care of the details of putting the clichés together correctly. However, KBEmacs is not in a position either to suggest to the programmer which clichés to choose, or to critique the programmer's choices. Another area of difficulty with KBEmacs is the overly imperative, order-dependent nature of KBEmacs commands.

These problems with KBEmacs stem from the fact that it has no notion of specifications or of the relationship between specification and implementation clichés. This is because KBEmacs was implemented on top of an earlier version of the Plan Calculus, rather than using CAKE. The availability of CAKE, which supports logical reasoning within the Plan Calculus, now makes it possible to provide a more declarative and design-oriented language for the programmer to use. The reasoning facilities in CAKE will also support the detection and explanation of some errors made by the programmer and some automatic selection of clichés.

3.2 A Target Scenario

Figure 7 shows a hypothetical interaction between a programmer and the proposed Design Apprentice. This is the first part of a much longer target scenario [47], in which the initial design shown is further elaborated and changed by the programmer, with assistance from the Apprentice.

Figure 7 begins with the programmer providing a specification. This could be done interactively or prepared with a text editor and submitted to the Apprentice all at once. The specification consists of two parts. The first part describes a hardware device (an imaginary device called the K7). The second part describes a driver program for the K7. (The domain of device drivers was chosen, because it is a good example of the kind of domain in which the Apprentice approach is most appropriate—namely a domain where there are many similar programs, but where each program is likely to have some unanticipated idiosyncrasies.)

```
> The "K7" is an interactive display device where:
    The screen height is 4 lines.
    The screen width is 40 characters.
    The I/O is in terms of ASCII characters.
    Direct cursor positioning is not supported.
    The bus interface is a standard SLU except that:
      Writing a 1 in Bit 1 of the XCSR initializes the device.
        Initialization of the device blanks the screen and homes the cursor.
        Completion of initialization is signaled in the same way as the
          transmission of a character.
        Sending characters to the K7 and initialization of the K7 cannot be
          done at the same time.
> The K7 driver is an interactive display driver where:
    Echoing is not supported.
    Rubout handling is not supported.
    The basic software functions are:
      Supported: PUTC, WRITE & GETC.
      Ignored: OPEN & CLOSE.
      Not supported: READ & SEEK.
    The control codes are:
      :CLEAR - blanks the screen.
      :RESET - does a :CLEAR and flushes pending input.
      :LINE-NUMBER - returns the cursor line.
    The implementation guidelines are:
      1- Do no consing.
      2- Favor space efficiency.
      3- Defer error checking.
> Write code for the K7 driver.
```

Request for advice regarding implementation of `:CLEAR`.
The following suggest implementing `:CLEAR` *by initialization of the* `K7`*:*
(1) `:CLEAR` *blanks the screen.*
(2) The only `K7` *operation that blanks the screen is initialization.*
However, this decision leads to the following constraint violations:
(3) Initialization homes the cursor, which is not part of the specification of `:CLEAR`.
(4) A device should only be initialized by its initialization function.

```
>Override constraints.
```

Request for advice regarding implementation of `:LINE-NUMBER`.
The following suggest no choices for implementing `:LINE-NUMBER`*:*
(1) `:LINE-NUMBER` *returns the cursor line.*
(2) No `K7` *operation returns the cursor line.*

```
> Postpone.
```

Constraint Violation: Interactive device drivers usually have a lookahead operation.
The following are standard lookahead specifications:
(1) `:ARE-CHARS-WAITING` *returns* `T` *if there are pending characters.*
(2) `:NUMBER-CHARS-WAITING` *returns the number of pending characters.*
(3) `:PEEK` *returns* `NIL` *if there are no pending characters and returns the next character (without removing it from the input buffer) if a character is pending.*

```
> Use (2).
```

Figure 7. Excerpt of a target scenario for program design. Programmer input follows the ">" prompt. Response from the Design Apprentice is in italics.

Notice that the statements at the beginning of Figure 7 are much more declarative than KBEmacs commands. They also have a markedly different character than logic-based specifications, such as Larch [23] or Ina Jo [28], or very-high-level programming languages, such as Refine [1] or GIST [4], due to the prominent use of design clichés, such as *interactive display device* and *serial line unit* (SLU). The use of clichés makes this in effect an extensible language.

The K7 specification contains both positive information, which describes how particular roles of this cliché are filled in (e.g., the screen height), and negative information, which states that some aspects of the cliché are not relevant (e.g., the K7 does not support direct cursor positioning).

A particularly interesting part of the driver specification is the implementation guidelines section at the end. The Apprentice will use these guidelines when deciding which algorithms to pick in implementing the driver. For example, the first two guidelines cause the Apprentice to select algorithms with no dynamic storage allocation and which trade time for space. The third guideline instructs the Apprentice to defer inclusion of error checking code until after the prototype version of the driver is written and tested. The key benefit of this postponement is *not* that it saves the Apprentice coding time, but that it saves the programmer thinking time.

The remainder of the interaction in Figure 7 illustrates the Design Apprentice's ability to detect and explain errors made by the programmer. The interactions shown will be supported by the facilities in Cake for propagation of information, dependencies, and contradiction handling.

One kind of interaction revolves around incomplete knowledge. After being requested to write the code for the K7 driver in the middle of Figure 7, the Apprentice starts to make the remaining design decisions. The first request for advice arises when the only way the Apprentice knows to implement one of the K7 operations (`:CLEAR`) conflicts with its default constraints. The programmer in this instance tells the Apprentice to override the constraints, effectively extending the specification of `:CLEAR`. A second request for advice arises when the Apprentice lacks any applicable knowledge for how to implement the `:LINE-NUMBER` operation. Notice that the programmer may at any time postpone answering a question, which enables him to maintain control of the interaction.

Another kind of interaction revolves around inconsistency, which can be of two forms. First, there may be inconsistency between different things the programmer says explicitly (this is not illustrated in the short excerpt here). Second, there may be inconsistency between what the programmer says and knowledge contained in the clichés library (e.g., the constraint violation at the end of Figure 7).

After the problems with `:CLEAR` and `:LINE-NUMBER` have been resolved, the Apprentice generates executable code for the K7 driver, as requested. (Space constraints make it impossible to show the code created. See [47].) To do so, the Apprentice needs to automatically make a number of additional detailed implementation decisions. For example, it has to make reasonable decisions regarding the implementation of the various semaphores and buffers in the driver.

The Design Apprentice differs from program generators, such as Draco [39] and PHInix [7], and procedurally implemented compilers for very-high-level languages such as SETL [53], primarily in the fact that it supports advice-taking from the programmer, and even allows

the programmer to get involved in detailed coding where necessary. This flexibility is made possible by the use of the Plan Calculus to represent the current design, together with the explicit representation of the possible design and implementation decisions in the cliché library.

The Design Apprentice also differs significantly from transformational implementation systems such as Pecos [6], PDS [12], and Refine [1]. Although overlays encode much of the same knowledge as transformations, the use of the Plan Calculus makes it possible to reason about this knowledge more easily. In addition, the derivation of a program in the Plan Calculus typically has many fewer steps than a transformational derivation, since there is no need for transformations that deal with minor syntactic variations.

3.3 Building The Design Apprentice

In the first year of the research, we plan to concentrate on building a comprehensive library of design clichés in the domain of device drivers, using the knowledge representation facilities of CAKE. One source for these clichès are system design textbooks such as [9] and [13].

In the second year of the research, we expect to implement the control and problem-solving structures necessary to support the kind of interactions illustrated in Figure 7. By the end of the second year, we expect to demonstrate a substantial portion of the capabilities of the Design Apprentice.

In the third year of the research, we plan to test our ideas by applying the Design Apprentice to a different domain (to be chosen). This new domain should have some overlap with device drivers (to demonstrate the reuse of clichés), but be different enough to challenge the methodology. By the end of the third year, we expect to complete a prototype that demonstrates all of the capabilities of the full target scenario [47] in both domains.

4 Optimization

A central thrust of the work described in the preceding section is toward designing software using standard, well-understood components (clichés) as much as possible. This approach promises to both reduce the cost of production and increase the reliability of the resulting software. However, any approach based on extensive reuse almost inevitably sacrifices the efficiency of the resulting software. Greater efficiency can almost always be obtained by writing specialized code tailored for the task at hand. A traditional approach to dealing with this kind of problem is to apply various optimization (improvement) techniques. We are investigating a new such technique: *automatic redistribution of intermediate results* [24].

Programs generally call many subroutines and perform many state-changing operations in the course of a computation. On completion, a subroutine returns one or more values and possibly produces some changes in the program's state (via side effects). The term *intermediate result* (or simply *result*) is used to refer to any such value or state change. Intermediate results are used as input to, and to satisfy preconditions of, other subroutines.

One source of inefficiency in programs is computing new results even though previously computed results would suffice. This is particularly true in programs designed using a library of predefined, general abstractions. Such programs can be improved by modifying them to

make certain intermediate results available later in the computation. This "redistribution" can be accomplished in many ways, such as explicitly storing results for later retrieval (using variables or a special-purpose data structure), passing extra parameters between subroutines, and moving code that can use a result into the local context of the result's initial computation. A common feature of these program modifications is they break down the abstraction barriers between the reusable components.

The concept of redistribution of intermediate results unifies and extends a number of standard optimization techniques. In particular, common compiler techniques such as subexpression elimination, dead code removal, loop fusion, and code motion [3] are all examples of redistribution. An essential limitation of optimization techniques in compilers, however, is that they operate only at the source language level. This means that they cannot apply the same optimizations to user-defined (or library) abstractions. For example, although an optimizing Lisp compiler may be able to conclude that a certain list copy operation is unnecessary, it cannot apply the analogous reasoning to eliminate the redundant copying of a user-defined abstraction.

Memoization [37] (explicitly saving values for later use) and tupling [41] (the combination of two initially separate functions into a single, tuple-valued function to share intermediate results) are more complex examples of redistribution that, to date, are typically only applied manually. In addition, although it is quite different in spirit, much of the effect of finite differencing [40] (incrementally updating a result rather then recomputing it completely on each cycle of a loop) can be obtained by means of redistribution.

4.1 An Example

As an illustration of the concept of optimization by redistribution of intermediate results, consider the following very small example involving a program designed using a library of set abstractions. In particular, suppose that we have chosen an implementation of the set abstraction in which sets are represented as lists of their elements, subject to the invariant that the list has no duplicates. Three of the library-defined operations associated with this implementation are shown below:

```
(DEFUN SET-ADD (ELEMENT SET)
  (IF (MEMBER ELEMENT SET) SET (CONS ELEMENT SET)))

(DEFUN SET-MEMBER? (ELEMENT SET)
  (MEMBER ELEMENT SET))

(DEFUN SET-SIZE (SET)
  (LENGTH SET))
```

Now consider the following program that uses these operations. Assume that the elided portions of the program include no invocations, either directly or via further subroutines, of the SET-SIZE operation. Furthermore, assume the program has no side effects, and that it only returns from the last expression in the body.

```
(DEFUN MY-PROGRAM (...)
  ... (SET-ADD ...) ...
  ... (SET-MEMBER? ...) ...
  (THE INTEGER ...))
```

The redistribution opportunity here relies on the observations that (1) the only operation in this example that depends on the representation invariant "no duplicates in list" is SET-SIZE, and (2) the test in SET-ADD has the sole purpose of enforcing this invariant. Inasmuch as MY-PROGRAM does not invoke SET-SIZE, and its only result (a value of type INTEGER) cannot involve a set, the maintenance of the invariant is not needed. Therefore, the input to the test in SET-ADD may be eliminated and its input redistributed directly to the consumers of its output. (This has the effect of converting SET-ADD from linear time to constant time.) In other words, the program may be rewritten as follows:

```
(DEFUN SET-ADD-1 (ELEMENT SET)
   (CONS ELEMENT SET))

(DEFUN MY-PROGRAM (...)
  ... (SET-ADD-1 ...) ...
  ... (SET-MEMBER? ...) ...
  (THE INTEGER ...))
```

Although this example is overly simplified (due to lack of space), it exhibits two key features of the redistribution of intermediate results, both of which make it a more tractable subcase of optimization in general.

First, redistribution does not change the basic algorithm of a program. For example, changing the implementation of a sorting routine from bubblesort to heapsort involves more than redistribution. Although we do not yet know how to precisely characterize this notion of "not changing the basic algorithm," we believe it has to do with preserving the structure of the design explanation—or ultimately, the proof of correctness—of the program.

Second, it is almost always more efficient to share an intermediate result instead of recomputing it. In general, knowing whether a program's efficiency will improve after an optimization is very difficult, involving a deep understanding of the performance properties of the algorithm as a whole. In the case of redistribution, however, this can usually be determined by a much more limited analysis (such as making sure the overhead of adding an auxiliary data structure does not exceed the cost of recomputation).

4.2 Automating Redistribution

We will demonstrate that redistribution of intermediate results can be automated by implementing the technique using the representation and reasoning facilities of CAKE. As part of this demonstration, we will provide to the optimizer, in addition to a program, a record of the design of the program in the Plan Calculus, such as will be produced by the Design Apprentice.

Automating the optimization technique can be broken into three subtasks: identifying opportunities for redistribution, determining whether the redistribution is advantageous, and implementing the redistribution. Of these three, the first subtask is the most difficult to automate, since *a priori* it requires a search in the space of all pairs of a result and a use of a result in the program. In our initial investigations, we will ignore the question of determining whether a redistribution is advantageous. Implementing the redistribution is straightforward: in the Plan Calculus representation of a program it is often no more than rerouting a data flow arc.

The search for potential redistribution pairs can be filtered in a number of ways. For example, one need only consider pairs in which the target result is in a control environment reachable from the source result. (This is easily computed in the Plan Calculus.) One can also filter out pairs that involve disjoint data types.

For each remaining pair, the question that needs to be answered is whether the earlier result satifies the weakest preconditions on the later use. (A special case of this is when an intermediate result is identical to a later result, i.e., when the same thing is literally computed twice.) A major focus of the research will be to discover constraints on the structure of the design record that make answering this form of question tractable. For example, if the design record is equivalent to a complete proof of correctness of the program, the fact that certain clauses in a postcondition are not used in the proof may suggest that the relevant results may be replaced by weaker ones. We also need to develop techniques for using incomplete specifications, while still guaranteeing that the optimization is correctness-preserving.

If it is not possible to fully automate the identification of redistribution opportunities, we can also apply the assistant approach to this subtask. For example, the programmer may be able to help prune the search. In the limit, the programmer might ask the system to redistribute a specific result to a specific target location, with the system taking care of the details of implementing the redistribution. Alternatively, the system may be able to query the programmer for a specific bit of additional specification information needed to justify a particular potential redistribution (similar to Miller's proposal for an "interactive compiler" [36]).

In the first year of the research, we will concentrate on developing a rich corpus of hand-worked examples of the technique applied to a wide range of programs. A major goal of this phase will be to gain further insight into the structure of the design record required to support automation of the technique. Going hand-in-hand with these example programs will be a library of reusable abstractions and their implementations.

In the second year of the research, we will begin implementing the demonstration system. We expect to work first on modules for automatically implementing redistribution, such as by adding global variables, hash tables, etc., and secondly on the techniques for pruning the search. For convenience, we will use Lisp as the source language; however, our system will be to a large extent language-independent due to the use of the Plan Calculus as the internal representation.

In the third year of the research, we will concentrate on strengthening the ability of the system to prove properties of programs within the framework of the design record. One of our final measures of success will be to demonstrate actual runtime speedup on realistic programs resulting from automatic redistribution of intermediate results.

5 Debugging

No technology will ever completely eliminate error in the software development process. The best we can hope for is to detect bugs as early as possible and provide support for correcting them. For example, the Requirements Apprentice described in [50, 42] is directed towards detecting bugs that can creep in at the very earliest point in the process, when an informal requirement is first formalized. In this section, we first report on an intelligent assistant

for localizing bugs in completed programs. We then describe further research to generalize and extend the system to support incremental change (evolution) throughout the design and implementation.

5.1 Localizing a Bug

We have recently completed a prototype debugging assistant, called Debussi [29], that assists a programmer in the task of determining the smallest region of a program that manifests a bug. This task is only part of the debugging process (other tasks are detecting an bug and repairing it), but it is a particularly time-consuming one for people to perform in large software systems.

Formally speaking, there are only two kinds of bugs that can occur in a program: the wrong subroutine (or primitive) is called at some point, or there is something wrong with the pattern of data and control flow in the definition of some subroutine. Debussi's goal is to narrow the bug to either a single erroneous call, or a single subroutine definition in which the data/control flow is erroneous.

The way Debussi works is best described by means of an example. The top part of Figure 8 shows a simple unification program that contains a bug. (Again, for convenience we are using Lisp as the source language, but the implementation of Debussi is to a large extent language-independent.) As illustrated by the test cases in the middle part of the figure, the toplevel function `UNIFY` compares two patterns. In these patterns, variables are represented by lists whose first element is "?". If it is not possible to unify the patterns, the symbol `NOMATCH` is returned; otherwise an association list is returned specifying the necessary variable bindings. Note that a return value of `NIL` means that the patterns unify without binding any variables.

The third part of Figure 8 shows an interaction with Debussi that leads to the localization of a bug. This interaction begins with a test case that manifests a bug. The two patterns shown should unify with a `NIL` binding list. Notice that the programmer here is performing the task of detecting the existence of a bug in the first place. There is a large body of literature on automating the functional testing of programs (see [8]), including our earlier work on a Testing Assistant [11].

The command `(OUTPUT-SHOULD-HAVE-BEEN NIL)` informs Debussi that the preceding test case should have resulted in a different value. Debussi responds by translating the program `UNIFY` into the Plan Calculus and using CAKE to produce a proof (using a mixture of symbolic and concrete execution) of why the output is wrong. The dependencies in this proof are used to localize the bug.

As a starting hypothesis, Debussi assumes that the data flow and control flow structure of `UNIFY` is correct, i.e., that the bug is in one of the calls. If this assumption fails to further localize the bug, it will then be reasonable to assume that something is in fact wrong with the structure of `UNIFY`.

A first analysis of the dependencies produced by CAKE reveals that that the calls to `EXTEND-IF-POSSIBLE` and `FREEOF?` cannot be responsible for the bug. However, in the toplevel execution of `UNIFY`, there are fifteen calls that do participate in the computation of the incorrect result: five predicates that together decide that the fifth `COND` clause is applicable,

```
(DEFUN UNIFY (P Q &OPTIONAL (FRAME NIL))
  (COND ((EQ FRAME 'NOMATCH) 'NOMATCH)
        ((AND (VAR? P) (VAR? Q)) (VAR-VAR-MATCH P Q FRAME))
        ((VAR? P) (EXTEND-IF-POSSIBLE P Q FRAME))
        ((VAR? Q) (EXTEND-IF-POSSIBLE Q P FRAME))
        ((AND (CONSP P) (CONSP Q))
         (UNIFY (CDR P) (CDR Q) (UNIFY (CAR P) (CAR Q) FRAME)))
        ((EQ P Q) FRAME)
        (T 'NOMATCH)))

(DEFUN VAR? (X) (AND (CONSP X) (EQ (CAR X) '?)))

(DEFUN VAR-VAR-MATCH (V W FRAME)
  (IF (EQ V W) FRAME
    (LET ((BV (ASSOC V FRAME :TEST #'EQUAL))
          (BW (ASSOC W FRAME :TEST #'EQUAL)))
      (IF (AND (NULL BV) (NULL BW)) (CONS (CONS V W) FRAME)
        (UNIFY (IF BV (CDR BV) V) (IF BW (CDR BW) W) FRAME)))))

(DEFUN EXTEND-IF-POSSIBLE (V VAL FRAME)
  (LET ((BV (ASSOC V FRAME :TEST #'EQUAL)))
    (COND (BV (UNIFY (CDR BV) VAL FRAME))
          ((FREEOF? V VAL FRAME) (CONS (CONS V VAL) FRAME))
          (T 'NOMATCH))))

(DEFUN FREEOF? (VAR E FRAME)
  (COND ((ATOM E) T)
        ((VAR? E)
         (AND (NOT (EQUAL VAR E))
              (LET ((B (ASSOC E FRAME :TEST #'EQUAL)))
                   (OR (NULL B) (FREEOF? VAR (CDR B) FRAME)))))
        (T (AND (FREEOF? VAR (CAR E) FRAME)
                (FREEOF? VAR (CDR E) FRAME)))))

          ----------------------------------------

> (UNIFY '(F (? X)) '(G 3)) --> NOMATCH
> (UNIFY '(F (? X)) '(F (G (? Y)))) --> (((? X) . (G (? Y))))
> (UNIFY '(F 3 4) '(F 3 4)) --> NIL

          ----------------------------------------

> (UNIFY '((? F) 4) '((? F) 4)) --> ((? F) . (? F))
> (OUTPUT-SHOULD-HAVE-BEEN NIL)
```

Pursuing suspect: `(UNIFY '(? F) '(? F)) --> ((? F) . (? F))`
Are the arguments to `UNIFY` *correct?* YES

Expanding structure of `UNIFY`.

Expanding structure of `VAR-VAR-MATCH`.

The bug is the call to `(EQ V W)` *in* `VAR-VAR-MATCH`.

Figure 8. An bug localized by Debussi.

four function calls that compute data used by the predicates, and six function calls that compute the result. Any one of these calls could be responsible for the bug.

The number of suspects can be further narrowed by appealing to the following pruning method, which is closely related to the constraint suspension technique for hardware troubleshooting described in [15]. Assuming that the current manifestation is due to a single bug (this is not the same as assuming there is only one bug anywhere in the program, which is certainly not very plausible for a large system), if every possible output of the suspect leads to an erroneous result (including abnormal termination of the program), then that suspect can be exonerated (removed from consideration). In other words, it is not possible to correct the bug by changing this suspect.

For example, in the execution of the failing test case, the predicate `(VAR? P)` returns `NIL`. To exonerate this suspect, Debussi considers what would happen if the predicate returned a non-nil value instead. Since this also leads to an erroneous overall result, this predicate is exonerated. In the example in Figure 8, this kind of pruning exonerates all five suspect predicates. Furthermore, this allows the four calls computing values used by the predicates to be exonerated as well, since their values are used only by the predicates.

Note that if the single-bug assumption turns out to be wrong in a particular instance, Debussi may end up in the anomalous state of having no suspects left. In this situation, it can either give up and notify the programmer, or try applying the same pruning method to pairs of suspects, triples, and so on (this is liable to be too expensive for all but the smallest programs). However, Debussi may also succeed in finding a bug using the simple pruning, even when there is more than one, depending on the exact details of the situation.

In terms of CAKE, what this pruning technique is doing is forcing it to look for an alternative proof of the failure of the test case that does not involve the given suspect. Since CAKE is a limited reasoner, it is possible that the first proof it found had unnecessary dependencies. For example, in CAKE's initial symbolic execution, the result of a program depends on the value (nil or non-nil) of every predicate that is tested on the execution path. To exonerate a predicate, we see if assuming it produced the opposite value still leads to an erroneous result.

To choose between the six remaining suspects, Debussi solicits additional information from the user. Debussi chooses to ask first about the recursive calls to `UNIFY` using the heuristic that user-defined functions are more likely to be associated with bugs than built-in functions. Starting with the first (in execution order) of these calls, Debussi determines whether the bug is in the recursive call to `UNIFY` by asking the user whether the inputs provided are correct (see Figure 8). The user responds that they are correct. Since the output of this call is already known to be incorrect (it is the same as the result of the toplevel execution of `UNIFY`), this localizes the bug. Debussi continues by expanding this call.

Things now proceed much as above except that, after predicate pruning, only one suspect `(VAR-VAR-MATCH P Q FRAME)` remains. No questions need to be asked about the inputs and output of this call, because they are connected directly to the inputs and output of `UNIFY` and are therefore known to conflict (under the assumption that the data flow and control flow in `UNIFY` is correct).

Debussi then proceeds to look in detail at the definition of `VAR-VAR-MATCH`. One of the initial suspects at this level is the predicate `(EQ V W)`, which returns `NIL`. Resimulating with

this predicate returning T, Debussi discovers that UNIFY returns the correct value for the test case. It therefore informs the user that the bug is this call (see Figure 8). Notice that starting with the user's initial declaration of a failing test case, Debussi only had to ask one question to localize the bug.

The repair for this bug (which is the responsibility of the user) is to call EQUAL instead of EQ. It is interesting to note that this bug was actually made by one of our students—we suspect it was due to the fact that he had implemented a special reader macro, which allowed him to type ?F for a variable instead of (? F)

Debussi differs from most other automated program debugging systems, such as [2, 25, 32, 38, 52, 54], which are built around libraries of patterns for correct and/or incorrect programs. This knowledge-based approach renders these other systems more powerful in the area where they are knowledgeable, but much less widely applicable. Many of these other systems can also utilize their knowledge to suggest how to repair a bug. Debussi only addresses the problem of localization.

Due to its reliance on reasoning from first principles, Shapiro's system [55] for debugging Prolog programs comes closest to Debussi. To localize a bug, Shapiro's system performs a binary search of the computation history, asking the user about the correctness of the various intermediate results. This can sometimes work well, but fails to take advantage of opportunities to prune the set of suspects.

5.2 Assistance for Design Evolution

Debussi has demonstrated the effectiveness of applying the Plan Calculus and CAKE's dependency-based reasoning facilities to assisting in the debugging of completed programs using concrete test cases. We believe this approach can be extended to cover more of the debugging process, i.e., bug detection and repair, and to deal with bugs earlier in the software process, i.e., during design. More generally, Debussi illustrates the beginnings of a dependency-based methodology for managing change (evolution) throughout the software process.

In the first year of the research, we will experiment with using Debussi to localize bugs in partially designed programs. This extension should be fairly straightforward, since the Plan Calculus is a wide-spectrum representation. Along with this, we will generalize the notion of "test case" to include any partial specification. For example, a test case might say that if the input to a routine is positive then its output is negative. The deductive capabilities of CAKE will be crucial to using this kind of symbolic information.

In the second year of the research, we will investigate the use of clichés in debugging. This is an area that we have explored in earlier work [54], but which we may now be able to improve upon significantly using the automated recognition techniques described in Section 2. We plan to consider both explicit "bug clichés," and the use of near-miss cliché recognition to detect bugs. It may also be useful to associate bug clichés with design clichés in the library.

In the third year of the research, we will investigate intelligent assistance for bug detection and repair. For example, in the area of bug detection, it may be useful to associate test generation information with design clichés. In the area of repair, we will explore the use of incremental reasoning techniques to verify that a repair is successful.

6 Conclusion

The preceding sections have covered a wide range of research topics, with a primary goal of bringing out their synergy. Work in each of these four areas shares not only a philosophical outlook—the assistant approach and clichés—but are also using the same experimental medium—the Plan Calculus and CAKE. This facilitates communication about shared problems and makes it possible to quickly exploit opportunities to share results.

Specific examples of expected synergy include the following (some of which have already been pointed out above): Recognition and design can be viewed as inverse functions that share the same library and representations. Design will eventually include debugging techniques. Optimization techniques will compensate for inefficiencies introduced by cliché-based design. Optimization may make use of recognition techniques to identify opportunities for optimization. Debugging may make use of recognition techniques to recognize bug clichés.

References

[1] L. M. Abraído-Fandiño. An overview of Refine 2.0. In *Second Int'l. Symp. on Knowledge Eng.–Software Engineering*, Madrid Spain, April 1987.

[2] A. Adam and J. Laurent. LAURA, A system to debug student programs. *Artificial Intelligence*, 15:75–122, 1980.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] R. M. Balzer. A 15 year perspective on automatic programming. *IEEE Trans. on Software Engineering*, 11(11):1257–1267, November 1985.

[5] D. R. Barstow. An experiment in knowledge-based automatic programming. *Artificial Intelligence*, 12(1 & 2):73–119, 1979. PhD thesis. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[6] D. R. Barstow. *Knowledge-Based Program Construction.* North-Holland, New York, 1979.

[7] D. R. Barstow. A perspective on automatic programming. *AI Magazine*, 5(1):5–27, Spring 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[8] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1983.

[9] T. Biggerstaff. *Systems Software Tools.* Prentice-Hall, 1986.

[10] D. Brotsky. An algorithm for parsing flow graphs. Technical Report 704, MIT Artificial Intelligence Lab., March 1984. Master's thesis.

[11] D. Chapman. A program testing assistant. *Comm. of the ACM*, 25(9):625–634, September 1982.

[12] T. E. Cheatham. Reusability through program transformation. *IEEE Trans. on Software Engineering*, 10(5):589–595, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[13] D. Comer. *Operating Systems Design: The XINU Approach.* Prentice-Hall, 1984.

[14] D. S. Cyphers. Automated program explanation. Working Paper 237, MIT Artificial Intelligence Lab., August 1982.

[15] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1–3):347–410, December 1984.

[16] J. Domingue. Itsy: An automated programming advisor. Technical Report 22, The Open University, Human Cog. Res. Lab, Milton Keynes, England, June 1987. PhD thesis.

[17] H. Ehrig, M. Nagl, and G. Rozenberg, editors. *2nd Int. Workshop on Graph-Grammars and Their Application to Computer Science.* Springer-Verlag, New York, Haus Ohrbeck, Germany, October 1982. Lecture Notes In Computer Science Series, Vol. 153.

[18] G. Faust. Semiautomatic translation of COBOL into HIBOL. Technical Report 256, MIT Lab. of Computer Science, March 1981. Master's thesis.

[19] Y. A. Feldman and C. Rich. Bread, Frappe, and Cake: The gourmet's guide to automated deduction. In *Proc. 5th Israeli Symp. on Artificial Intelligence*, Tel Aviv, Israel, December 1988.

[20] Y. A. Feldman and C. Rich. Principles of knowledge representation and reasoning in the FRAPPE system. In *Proc. 11th Int. Joint Conf. Artificial Intelligence*, Detroit, MI, August 1989. Submitted.

[21] S. F. Fickas and R. Brooks. Recognition in a program understanding system. In *Proc. 6th Int. Joint Conf. Artificial Intelligence*, pages 266–268, Tokyo, Japan, August 1979.

[22] C. Green and D. R. Barstow. On program synthesis knowledge. *Artificial Intelligence*, 10(3):241–279, November 1978. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[23] J. V. Guttag, J. J. Horning, and J. M. Wing. The Larch family of specification languages. *IEEE Software*, September 1985.

[24] R. J. Hall. Program improvement by automatic redistribution of intermediate results. Working Paper 305, MIT Artificial Intelligence Lab., May 1988. PhD proposal.

[25] W. L. Johnson and E. Soloway. PROUST: Knowledge-based program understanding. *IEEE Trans. on Software Engineering*, 11(3):267–275, March 1985. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[26] V. Jonckers. Exploring algorithms through mutations. In *7th European Conference on Artificial Intelligence*, pages 556–568, Brighton Centre, England, July 1986. Vol. 1.

[27] E. Kant. Understanding and automating algorithm design. *IEEE Trans. on Software Engineering*, 11(11):1361–1374, November 1985.

[28] R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Trans. on Software Engineering*, 11(1):32–43, January 1985.

[29] R. I. Kuper. Automated techniques for the localization of software bugs. Technical Report 1053, MIT Artificial Intelligence Lab., May 1988. Master's thesis.

[30] J. Laubsch and M. Eisenstadt. Domain specific debugging aids for novice programmers. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 964–969, Vancouver, British Columbia, Canada, August 1981.

[31] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3), May 1986.

[32] F. J. Lukey. Understanding and debugging programs. *Int'l Journal of Man-Machine Studies*, 12:189–202, 1980.

[33] R. Lutz. Program debugging by near-miss recognition and symbolic evaluation. Technical Report CSRP.044, Univ. of Sussex, England, 1984.

[34] R. Lutz. Diagram parsing — A new technique for artificial intelligence. Technical Report CSRP.054, Univ. of Sussex, England, 1986.

[35] B. P. McCune and J. S. Dean. Advanced tools for software maintenance. Technical Report 313, Rome Air Development Ctr., Griffiss AFB, NY 13441, December 1982.

[36] J. Miller. An interactive compiler: Tools and techniques for program transformation. Proposal to National Science Foundation, 1988.

[37] J. Mostow and D. Cohen. Automating program speedup by deciding what to cache. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 165–172, Los Angeles, CA, August 1985. Vol. 1.

[38] W. R. Murray. Heuristic and formal methods in automatic program debugging. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, Los Angeles, CA, August 1985.

[39] J. M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Trans. on Software Engineering*, 10(5):564–574, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[40] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):402–454, July 1982.

[41] A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *Proc. 1984 ACM Symp. on Lisp and Functional Programming*, pages 273–281, 1984.

[42] H. B. Reubenstein and R. C. Waters. The Requirements Apprentice: An initial scenario. In *Proc. 5th Int. Wrkshp on Software Specs. and Design*, Pittsburgh, PA, May 1989.

[43] C. Rich. A formal representation for plans in the Programmer's Apprentice. In *Proc. 7th Int. Joint Conf. Artificial Intelligence*, pages 1044–1052, Vancouver, British Columbia, Canada, August 1981. Reprinted in M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, Springer Verlag, 1984 and in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[44] C. Rich. Inspection methods in programming. Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD thesis.

[45] C. Rich. The layered architecture of a system for reasoning about programs. In *Proc. 9th Int. Joint Conf. Artificial Intelligence*, pages 540–546, Los Angeles, CA, 1985.

[46] C. Rich. Inspection methods in programming: Clichés and plans. Memo 1005, MIT Artificial Intelligence Lab., December 1987. Submitted to *Artificial Intelligence.*

[47] C. Rich and R. C. Waters. The Programmer's Apprentice: A program design scenario. Memo 933A, MIT Artificial Intelligence Lab., November 1987.

[48] C. Rich and R. C. Waters. Automatic programming: Myths and prospects. *IEEE Computer*, 21(8):40–51, August 1988.

[49] C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11):10–25, November 1988. Also published as MIT AI Memo 1004.

[50] C. Rich, R. C. Waters, and H. B. Reubenstein. Toward a Requirements Apprentice. In *Proc. 4th Int. Wrkshp on Software Specs. and Design*, Monterey, CA, April 1987.

[51] P. M. Ritto. Parallel flow graph matching for automated program recognition. Working Paper 310, MIT Artificial Intelligence Lab., July 1988.

[52] G. R. Ruth. Analyis of algorithm implementations. Technical Report 130, MIT Project Mac, 1973.

[53] J. T. Schwartz et al. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, 1986.

[54] D. Shapiro. SNIFFER: A system that understands bugs. Memo 638, MIT Artificial Intelligence Lab., June 1981. Master's thesis.

[55] E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[56] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. on Software Engineering*, 10(5):595–609, September 1984. Reprinted in C. Rich and R.C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[57] R. M. Stallman. EMACS: The extensible, customizable self-documenting display editor. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, OR, June 1981.

[58] R. C. Waters. KBEmacs: A step towards the Programmer's Apprentice. Technical Report 753, MIT Artificial Intelligence Lab., May 1985.

[59] R. C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. on Software Engineering*, 11(11):1296–1320, November 1985. Reprinted in C. Rich and R. C. Waters, editors, *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986 and in T. Ichekawa, editor, *Language Architectures and Programming Environments*, MIT Press, in preparation.

[60] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. on Software Engineering*, 14(8):1207–1228, August 1988.

[61] L. M. Wills. Automated program recognition. Technical Report 904, MIT Artificial Intelligence Lab., September 1986. Master's thesis.

CS-TR Scanning Project
Document Control Form

Date : 4/19/95

Report # AIM-1100

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☐ Technical Report (TR) ☒ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 28(34-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☐ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP : (1) UN #'ED TITLE PAGE	
(2-28) PAGES #'ED 1-27	
(29-31) SCANCONTROL, DOD(2)	
(32-34) TRGTS (3)	

Scanning Agent Signoff:
Date Received: 4/19/95 Date Scanned: 4/20/95 Date Returned: 4/27/95

Scanning Agent Signature: Michael W. Cook

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Rec'd 8/3/89

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AIM 1100	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER: AD-A206 698	
4. TITLE (and Subtitle): Intelligent Assistance for Program Recognition, Design, Optimization, and Debugging	5. TYPE OF REPORT & PERIOD COVERED: memorandum
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Charles Rich and Richard C. Waters	8. CONTRACT OR GRANT NUMBER(s): N00014-88-K-0487
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, Virginia 22209	12. REPORT DATE: January 1989
	13. NUMBER OF PAGES: 28
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, Virginia 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

automatic programming
debugging
design
optimization
recognition
Programmer's Apprentice

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We describe research in four related areas, based on the following theoretical principles: the assistant approach (incremental automation) and the exploitation of cliches (using knowledge of common engineering practice). Each investigation involves the construction of a prototype system to provide intelligent assistance for a person performing the task: A recognition assistant will help reconstruct the design of a program , given only its source code. A design assistant will assist

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 Cont'd...

a programmer by detecting errors and inconsistencies in his design choices and by automatically making many straightforward implementation decisions. An optimization assistant will help improve the performance of programs by identifying intermediate results that can be reused. A debugging assistant will aid in the detection, localization, and repair of errors in designs as well as completed programs. These prototypes will be constructed using two shared technologies: a programming language independent formal representation for programs and programming knowledge (the Plan Calculus) and an automated reasoning system (CAKE), which supports both general logical reasoning and special-purpose decision procedures.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**