MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 1004 November 1987

The Programmer's Apprentice Project: A Research Overview

by

Charles Rich and Richard C. Waters

Abstract

The goal of the Programmer's Apprentice project is to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify and document programs. This research goal overlaps both artificial intelligence and software engineering. From the viewpoint of artificial intelligence, we have chosen programming as a domain in which to study fundamental issues of knowledge representation and reasoning. From the viewpoint of software engineering, we seek to automate the programming process by applying techniques from artificial intelligence.

Submitted to the Joint IEEE Software/IEEE Expert Special Issue on The Interactions Between Expert Systems and Software Engineering.

This article describes research done 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 National Science Foundation under grant IRI-8616644, in part by the IBM Corporation, in part by the NYNEX Corporation, in part by the Siemens Corporation, and in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

1 Introduction

The goal of the Programmer's Apprentice project is to develop a theory of how expert programmers analyze, synthesize, modify, explain, specify, verify and document programs. This research goal overlaps both artificial intelligence and software engineering. From the viewpoint of artificial intelligence, we have chosen programming as a domain in which to study fundamental issues of knowledge representation and reasoning. From the viewpoint of software engineering, we seek to automate the programming process by applying techniques from artificial intelligence.

The project consists of two intermingled lines of activity. One line of activity, described in Section 2, is concerned with developing knowledge representation and reasoning techniques motivated by features of the programming task. The second line of activity, described in Sections 3 and 4, focuses on the building of prototype systems to demonstrate the feasibility of various kinds of programming automation.

1.1 The Assistant Approach

One approach to solving current software problems is to try to totally eliminate programmers through *automatic programming.* As typically conceived, automatic programming calls for the end user to write a complete specification for what he wants; a completely automatic system then implements this specification. In sufficiently narrow applications, such as report generation, automatic programming has been applied successfully. However, in broader domains fully automatic programming is not a realistic near-term goal.

The fundamental difficulty with the fully automated approach is the trade-off between, on the one hand, the generality of a specification language, and on the other hand, its ease of use and automatic implementation (compilation). Specification languages have been designed that are easy to use and relatively easy to compile, but only for narrow applications. When general-purpose languages are considered, however, the results have been less satisfactory. Writing a complete specification in a general-purpose specification language is not much easier (and sometimes much harder) than writing a program. Furthermore, there has been little success in developing automatic systems that can implement acceptably efficient programs from such specifications.

An alternate approach to the software problem is to intelligently 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, such as 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 software development without getting bogged down in the mundane details that currently use up most of every programmer's time.

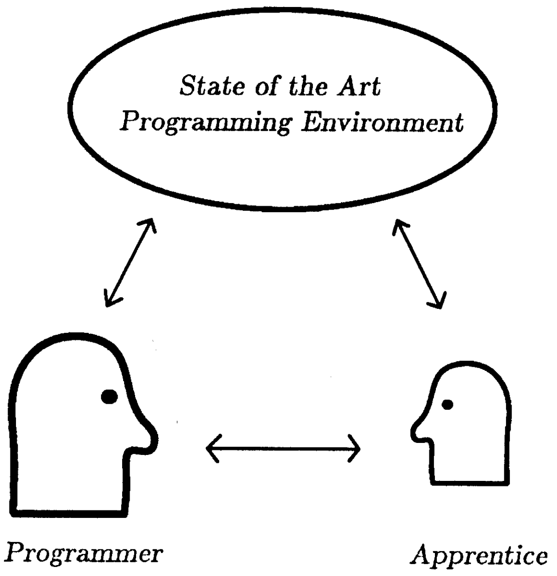

Figure 1. The Programmer's Apprentice is a new agent in the software process. It uses the tools in the programming environment and interacts with the programmer like a human assistant.

Experience has shown that this approach can be very successful. However, it is difficult to apply widely, because it is hard to find enough qualified people who are willing to function as assistants. Our goal is to provide *every* programmer with a support team in the form of an intelligent computer program, called the Programmer's Apprentice.

A key distinction between the Programmer's Apprentice and the conventional software development paradigm is that the Apprentice is an active agent in the software process, rather than a passive tool (see Figure 1). This provides the programmer with a familiar interaction model, namely cooperation with an assistant, through which to introduce new kinds of automation.

An Additive and Incremental Approach

A second feature of the assistant approach is that it is strictly additive. Both the programmer and the Apprentice have access to the existing programming environment. There is no need to re-invent the facilities that are already available in state-of-the-art environments. Thus, the programmer is not prevented from doing ordinary things in ordinary ways.

Finally, the development of the Apprentice is incremental. 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 amount the Apprentice can do will increase. This means that payoff from the research does not have to wait until some part of the task can be totally automated.

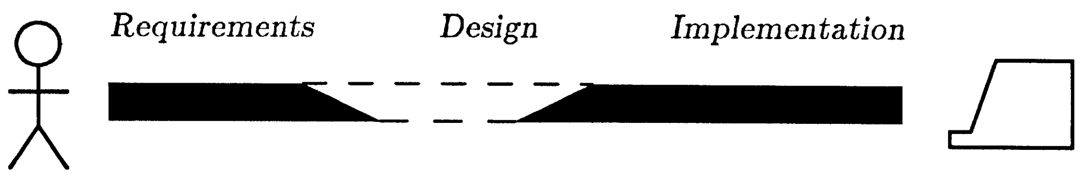

Figure 2. Prototypes of the Programmer's Apprentice are being built working inward from the two boundaries of the software development process.

1.2 Building Prototypes of the Apprentice

Viewed most fundamentally, the software development process has, at one end, the desires of a user and, at the other end, a program that can be executed on a machine. The part of the software process closest to the user is typically called requirements acquisition; the part of the process nearest the machine is typically called implementation; the area in the middle can be generally described as design. The goal of the Programmer's Apprentice is to provide support for all parts of the process between these endpoints. This is essential in order to achieve dramatic improvements in programmer productivity.

Nevertheless, since the overall project is very large, we need to build prototypes of the Apprentice incrementally. Rather than trying to precisely define the boundaries between requirements acquisition, design, and implementation *a priori* (as for example, in the traditional waterfall model) and building prototypes separately in each of these areas, we have adopted the strategy of working inward from the two boundaries, as shown in Figure 2. This strategy allows us to explore and discover where the appropriate boundaries should be drawn as we proceed. Another advantage of this strategy is that our prototypes are always anchored at one end to an externally-defined boundary. Because of this, such a prototype can be useful by itself, as opposed to a prototype of a part of the Apprentice which "floats" in the middle of the process.

The major part of our work to date has focused on program implementation, culminating in the completion of a prototype Implementation Apprentice (also called KBEMACS). The principal benefit of the Implementation Apprentice is that it allows a programmer to construct a program rapidly and reliably by combining algorithmic fragments stored in a library. An additional benefit is that it provides a basis for intelligent program modification and maintenance. The principal limitations of the Implementation Apprentice are its narrow coverage of the programming process and weak reasoning abilities. An example of the performance of the prototype Implementation Apprentice is given in Section 3.

The Design Apprentice is the direct successor of the Implementation Apprentice. In comparison with the Implementation Apprentice, the Design Apprentice will have increased reasoning abilities and will be able to assist in a greater portion of the programming process, starting with the detailed, low-level parts of design, and growing upwards towards high-level design. In particular, the Design Apprentice will be able to detect errors and inconsistencies in the programmer's design decisions, which the Implementation Apprentice could not do. It will also be able to automatically make many straightforward implementation choices. Part of a target scenario for the prototype Design Apprentice scenario is given in Section 3.

The Requirements Apprentice is intended to assist a systems analyst in the creation and modification of software requirements. Unlike current requirements tools, which assume a formal description language, the focus of the Requirements Apprentice is on the boundary between informal and formal specifications. The Requirements Apprentice will support the earliest phases of creating a requirement, in which incompleteness, ambiguity, and contradiction are inevitable features. Part of a target scenario for the prototype Requirements Apprentice is given in Section 3.

2 Knowledge Representation and Reasoning Techniques

The first step in automating any task is to develop a computational model of the kinds of knowledge and reasoning used in that task. The next step is to equip the system with the store of specific knowledge needed to perform the task. In particular, any system which attempts to automate the programming task (either fully or partially) must have the following major components:

- A *representation* for structure and function in programs.
- A library of the commonly used methods (*clichés*) in various areas of programming.
- Techniques for *reasoning* about evolutionary change in programs.

The remainder of this section discusses these components in turn.

2.1 The Plan Calculus

The cornerstone of the Programmer's Apprentice is a formal representation for programs and programming knowledge, called the *Plan Calculus*. This formalism was developed early in the project and has been described in detail in a number of other publications [9,4,5,11,8]. We will only summarize its key properties here.

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 language has both a graphical notation, an example of which is shown in Figure 3, and

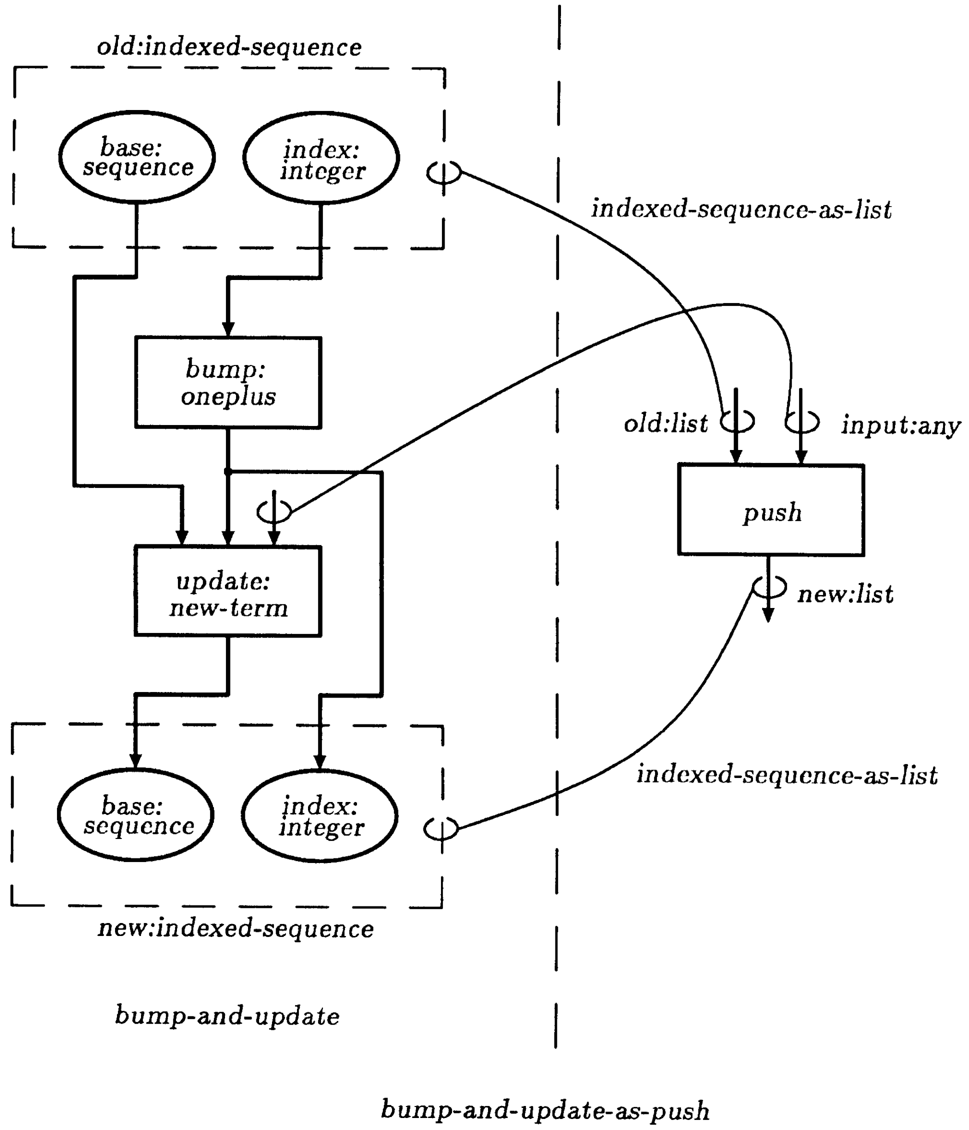

Figure 3. An example of program implementation knowledge represented in the Plan Calculus.

internal format used for automated reasoning. The Plan Calculus provides a canonical, easy to manipulate, and programming-language independent representation for many kinds of programming knowledge.

Figure 3 illustrates the flavor of the Plan Calculus. The structure depicted in this figure is called an *overlay*. Overlays are used, among other things, to represent the relationship between a specification and an implementation. The specification is shown on the right side of the overlay. The left side of the overlay gives an implementation plan. Alternative implementations of the same specification are represented by multiple overlays with the same right side.

For example, Figure 3 shows how to implement a *push* operation on a list, when the list is implemented as an *indexed-sequence*, i.e., a sequence with an associated index pointer. The right side of the overlay simply shows the push operation. (The formal preconditions and postconditions of the operation are specified in a logical language not shown in plan diagrams.) The left side of the overlay shows the corresponding plan for incrementing the index pointer and storing a new term at that location in the sequence.

The Plan Calculus gives the Apprentice a "mental language" that abstracts away from those aspects of algorithms and data structures which have to do only with how they are expressed in a particular programming language. As a result, the technology underlying the Programmer's Apprentice is inherently programming-language independent.

2.2 Using Clichés is Good Engineering

Expert engineers rarely construct complex artifacts (automobiles, electronic circuits, or software systems) by starting from first principles. Rather, they bring to the task their previous experience, in the form of knowledge of the commonly occurring structures (combinations of the primitives) in the domain.

We use the term *cliché* to refer to these commonly occurring structures. In normal usage, the word cliché has a pejorative sound that connotes overuse and a lack of creativity. However, in the context of engineering problem solving, this kind of reuse is a positive feature.

Formally, a cliché consists of a set of *roles* and *constraints*. The roles of a cliché are parts that vary from one occurrence of the cliché to the next. The constraints specify fixed elements of structure (parts that are present in every occurrence), and are used 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.

The Plan Calculus is designed to represent clichés as well as individual programs. A central aspect of research on the Programmer's Apprentice is the codification of clichés. We began by building a library of the clichés in the core area of routine symbolic programming, involving lists, arrays, graphs, and so on [4]. We are now codifying clichés in more specialized application areas, such as information systems and device drivers.

Clichés are organized into libraries by means of taxonomic hierarchies (clichés specialize and extend other clichés) and overlays (clichés implement other clichés). Cliché libraries are used to support both program synthesis and recognition of clichés in previously written programs.

2.3 A Hybrid Reasoning System

Ultimately, the degree of automation which the Apprentice can provide depends on its ability to reason about programs and their properties. Experimentation has shown that a combination of special-purpose techniques and general-purpose logical reasoning is required.

Special purpose representations and associated algorithms are essential in order 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 strict control, as the "glue" between inferences made in different special purpose representations.

A hybrid knowledge representation and reasoning system, called CAKE [6,7], has been developed, which will be a component of both the Design Apprentice and the Requirements Apprentice. Figure 4 shows the architecture of CAKE. Note that CAKE combines special-purpose representations, such as frames and the Plan Calculus, with general-purpose logical reasoning.

Figure 5 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 5.)

The propositional layer of CAKE provides three principal facilities. First, it automatically performs simple "one-step" deductions (lines 1–3). Second, it acts as a recording medium for dependencies, and thus supports explanation (line 3) and retraction (lines 4–5). Third, this layer detects contradictions (lines 6–7). Contradictions are represented explicitly in such a way that reasoning can continue with other information not involved in the contradiction. This is important for allowing a user

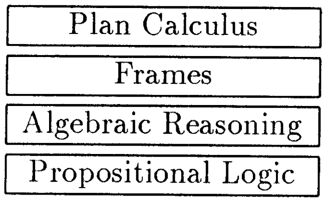

Figure 4. The hybrid knowledge representation and reasoning system (CAKE) has a layered architecture.

```
 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.
     s-A, Resume:   Ignore this contradiction.
     s-B:           Retract one of the premises.
 7>  s-B Retract one of the premises.
     Premise to retract: 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 5. An actual transcript illustrating the reasoning capabilities of the propositional, algebraic, and frame layers of CAKE.

to postpone dealing with problems.

Many of the capabilities illustrated in the target scenarios in Section 3 will be directly supported by facilities in the propositional layer of CAKE. For example, the explanations given by the Apprentice in the scenarios are essentially printouts of dependency information. Retraction is illustrated in the scenarios when the user corrects errors that the Apprentice has detected. The simple "one-step" deductions performed by the propositional layer provides the propagation of information illustrated in the scenarios.

The algebraic layer of CAKE is composed of special-purpose decision procedures for congruence closure, common algebraic properties of operators, such as commutativity, associativity, and transitivity, partial functions [2], 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. (The propositional layer and the congruence closure algorithm of the algebraic layer are derived from McAllester's Reasoning Utility Package [3].)

Reasoning about equality is invoked several times in the Requirements Apprentice scenario to detect potential problems. Other algebraic properties, such as transitivity, commutativity, etc., come up everywhere in formal modeling tasks.

The frames layer, which is built using facilities from many of the layers below, supports the conventional frame notions of inheritance (`:Specializes` in line 18), slots (`:Roles` in lines 19–21), and instances (line 22). A notable feature of CAKE's frame system is that constraints are implemented in a general way. For example, the definition of an Interface (line 21) has constraints between the roles of the frames filling its roles. When an instance of this frame is created (line 22) and a particular value (777777) is put into one of its 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).

The frames layer of CAKE provides the basic structuring and taxonomic facilities upon which the whole Apprentice architecture is based. It is used both for the library of clichés and for user-defined data structures.

While the overall CAKE system is primarily targeted at the task of representing and reasoning about programs, the propositional, algebraic and frame layers of CAKE are of much broader applicability. We have packaged these three layers into a separate utility, called FRAPPE (for FRAmes in a ProPositional Environment), which should be useful in other engineering applications in which reasoning about structured objects with constraints needs to be combined with dependency-directed reasoning.

3 Programmer's Apprentice Scenarios

This section elaborates the concept of the Programmer's Apprentice via three scenarios. The first scenario illustrates the capabilities of the prototype Implementation Apprentice, which has been completed. The second and third scenarios are target scenarios, which are guiding work on the Design Apprentice and Requirements Apprentice prototypes.

3.1 The Implementation Apprentice

The completion of the Implementation Apprentice (also called KBEMACS [16,15,17]) represents a major milestone in the project. The system has been demonstrated on a number of significant size examples, one of which is shown in Figure 6. The most prominent feature of this example is an order-of-magnitude productivity enhancement. Given the five commands at the top of the figure, the Implementation Apprentice produces the 55 line Ada program at the bottom.

The prototype Implementation Apprentice demonstrates a number other of important capabilities that are part of the overall Programmer's Apprentice concept:

- *Editing in terms of algorithmic structure.* Commands to the Apprentice allow the programmer to operate directly on the algorithmic structure of a program. This means that conceptually straightforward algorithmic changes to a program can be achieved by a single command, even when they require widespread textual changes in the program.

- *User-extendable library of clichés.* A programmer can define new clichés as easily as new subroutines. This has two benefits. First, a programmer can extend the Apprentice into new domains of programming. Second, a programmer can tailor the Apprentice to fit better with a given style of programming.

- *Escape to the surrounding environment.* The programmer is free to use any of the standard Lisp Machine programming tools. In particular, text-based (or syntax-based) editing can be applied to the program at any time. New capabilities are added to the programming environment without removing any existing ones.

- *Programming-language independence.* The internal operation of the Implementation Apprentice is substantially language independent. Although the system was originally constructed to operate on Lisp programs, relatively little effort was required in order to extend the system to operate on Ada programs as well.

- *Reasoning by means of constraints.* The Implementation Apprentice uses simple constraint propagation in order to determine some of the consequences of

```
Define a simple_report procedure 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.
```

$$\Downarrow$$

```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, "report.txt");
   loop
      begin
         NEW_LINE; PUT("Enter UNIT Key: "); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT("Invalid UNIT Key"); NEW_LINE;
      end;
   end loop;
   TITLE := "Report of Repairs on Unit " & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE);
   NEW_LINE(2); SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, UNIT_KEY); REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT("Page: "); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT("   Date     Defect    Description/Comment"); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT("Data Base Inconsistent");
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

Figure 6. Five commands to the Implementation Apprentice produce 55 lines of Ada code.

a programmer's decisions. This reduces the number of things which have to be specified by the programmer and enhances the reliability of the programs produced.

- *Taking care of details.* One of the ways in which the Apprentice can be particularly helpful to a programmer is by taking care of mundane details. For example, the Implementation Apprentice can generate most of the variable declarations in an Ada program.
- *Program modification.* The Implementation Apprentice has two specific commands (Replace and Share) which support program modification. In addition, constraints and editing in terms of algorithmic structure facilitate modification.
- *Program documentation.* The Implementation Apprentice can create a comment explaining the implementation structure of a program.

The last four of these capabilities are not demonstrated as strongly as the first four. For example, constraints are represented procedurally by functions which operate directly on the internal representation of the Plan Calculus, rather than in a more declarative, specification-like form. Similarly, each class of details is taken care of by a separate, *ad hoc* procedure rather than by general reasoning. The Implementation Apprentice cannot reason about side-effects in any general way. In addition, the program modification and documentation facilities only scratch the surface of what could be done. The CAKE system will provide the reasoning power to strengthen these capabilities in the Design Apprentice, as well as add new ones.

3.2 General Issues in the Target Scenarios

Before presenting the following two target scenarios, it is important to set the scene by discussing what the Apprentice knows before they begin. In each case, the Apprentice is assumed to have an intermediate level of prior knowledge. An intermediate level of knowledge is chosen because it best typifies the environment the Apprentice is expected to work in.

If it were possible for the Apprentice to know everything about what the user is going to do, then it would be preferable to have a program generator which constructed the desired programs completely automatically. This kind of total knowledge is possible only in very restricted applications. The Apprentice is intended to be applicable to a broad range of programming applications.

Alternatively, if the Apprentice knew much less than what is illustrated in the scenarios, the user would have to say too much. In particular, the user would have to either define the missing clichés or make do without them. In either case, the productivity and reliability benefits of the Apprentice would be significantly eroded. A major focus of this research is therefore to make sure that the Apprentice knows enough to be useful.

A second general issue in the two target scenarios is the user interface. Since the interface has not been fully designed, the interactions in the scenarios are intended to illustrate its major features, without being overly specific about details.

For example, input typed by the user is shown in simple English. Our prototypes will not, however, support natural language input. A stylized command language will be provided which supports the necessary semantic content illustrated in the scenario, but not the same degree of syntactic flexibility. Natural language understanding is a major research area in its own right, which is, by and large, independent of the central issues in building the Apprentice.

Finally, note that several errors have intentionally been introduced into what the user says in each scenario. The particular errors chosen may or may not appear plausible to particular readers. However, large numbers of errors *are* made during the programming process (most of which look quite stupid in retrospect). The errors introduced here were chosen to illustrate the capabilities of the Apprentice to detect and help correct errors in general.

3.3 The Design Apprentice

The target scenario for the Design Apprentice shows the detailed design of a device driver program. This type of program was chosen as a domain for two reasons. First, device drivers are of significant practical importance. Second, they are a good example of the kind of domain in which the Apprentice approach is most applicable—namely domains in which there are many similar programs, but in which each program is likely to have some unanticipated idiosyncrasy.

What the Design Apprentice Knows: Design Clichés

The Design Apprentice's knowledge is embodied in its clichés for typical specifications, typical designs, and typical hardware. Examples of specification clichés in the device driver domain include *initialization*, *reading* from a device, *writing* to a device, *opening* a device, and *closing* a device. Each of these clichés is annotated with information about what roles and constraints are mandatory, likely, or only possible.

Examples of design clichés in the device driver domain include *driver* and its specializations, such as *printer driver* and *interactive display driver*. The driver cliché is a very abstract cliché, which contains information that is common to all drivers. The printer driver cliché specifies things such as the fact that printer drivers only support writing operations and the fact that complex output padding is sometimes required after characters that cause large movements of the printing head. Examples of low-level algorithm clichés in the device driver domain include *semaphores*, *busy-waiting*, and *watermark processing*.

Examples of hardware clichés that the Design Apprentice knows include *serial line unit*, *printer*, and *interactive display device*. A serial line unit is a standard kind of bus interface used by many different kinds of hardware devices. It specifies a stereotyped cluster of four buffer and control registers, which are used to operate the device.

Capabilities Illustrated in the Design Apprentice Scenario

Figure 7 and Figure 8 are an excerpt from a longer scenario [12], which illustrates three major capabilities of the Design Apprentice, which distinguish it from the Implementation Apprentice:

- a declarative (specification-like) input language,
- detection and explanation of errors made by the programmer,
- automatic selection of reasonable implementation choices.

Figure 7 begins with the programmer giving a specification. This could be done interactively line-by-line, or prepared with a text editor and submitted to the Apprentice all at once. The specification consists of two parts. The first part describes the hardware (here an imaginary device called the K7). The second part describes a driver program for the K7.

The specification of the K7 uses the cliché interactive display device. 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).

The last five lines of the K7 specification make use of the serial line unit (SLU) cliché. The phrase "except that" indicates that the programmer is modifying the SLU cliché rather than just filling in its roles. The exception description makes use of a number of technial terms (i.e., "XCSR", "initialize the device", "blank the screen", etc.), which are defined for SLU's. This vocabulary makes it possible for the programmer to describe the exception succinctly.

The second part of the specification in Figure 7 concerns the software to be written, namely the driver program. As with the K7 specification, most of the specification says how various roles of the relevant cliché are filled in (the cliché in this case is interactive display driver).

A particularly interesting part of the driver specification is the implementation guidelines section at the end. The Apprentice uses these guidelines to decide which algorithms to pick when 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 "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 keyboard has three keys.
      key              character
    ACKNOWLEDGE   ACK  #O006
    YES             Y  #O131
    NO              N  #O116
  The bus interface is a standard SLU except that:
    Writing a 1 in Bit 1 of the XCSR initializes the device.
      Initializing the device blanks the screen and homes up the cursor.
      Completion of initialization is signaled the same as transmission of a character.
      Sending characters to the K7 and initializing 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: I do not know how to support :CLEAR.
Note:
  (1) :CLEAR blanks the screen.
  (2) The only K7 operation that blanks the screen is initialization.
Unfortunately:
  (3) Initializing the K7 homes up the cursor.
  (4) A device should not be initialized except by the INIT function.
> :CLEAR should home up the cursor.
> Support :CLEAR by initializing the device.
Request for Advice: I do not know how to support :LINE-NUMBER.
Note:
 (1) :LINE-NUMBER returns the cursor line.
 (2) No K7 operation returns the cursor line.
> Postpone.
Constraint Violation: There should be a lookahead operation.
Note: The following standard operations provide lookahead:
  (1) :ARE-CHARS-WAITING returns T if there are pending input characters.
  (2) :NUMBER-CHARS-WAITING returns the number of pending input characters.
  (3) :PEEK returns NIL if there are no pending input characters and
      returns the next character (without removing it from the input
      buffer) if a character is pending.
> Use (2).
```

Figure 7. Hypothetical dialog between a programmer and the Design Apprentice, excerpted from a longer target scenario illustrating the detailed, low-level design capabilities of the Apprentice. Programmer input follows the > prompt.

```
(defstruct K7                                   ; CONTROL BLOCK DEFINITION
  addr idle-sem in-buffer in-sem out-buffer out-sem
  sending? watermark pending)

(defun K7-control (dcb code &rest ignore)       ; UPPER-LEVEL DRIVER FUNCTIONS
  (without-interrupts
   (case code
     (:CLEAR (flush-fast-q (K7-out-buffer dcb))
             (setf (sem-count (K7-out-sem dcb))
                   (fast-q-size (K7-out-buffer dcb)))
             (sem-wait (K7-idle-sem dcb))
             (K7-clear dcb))
     (:RESET (flush-fast-q (K7-in-buffer dcb))
             (setf (sem-count (K7-in-sem dcb)) 0)
             (K7-control dcb :CLEAR))
     (:LINE-NUMBER not yet implemented)
     (:NUMBER-CHARS-WAITING (sem-count (K7-in-sem dcb))))))

(defun K7-getc (dcb)
  (without-interrupts
   (sem-wait (K7-in-sem dcb))
   (code-char (fast-deq (K7-in-buffer dcb)))))

(defun K7-putc (dcb char)
  (without-interrupts
   (if (not (K7-sending? dcb)) (sem-wait (K7-idle-sem dcb)))
   (sem-wait (K7-out-sem dcb))
   (fast-enq (K7-out-buffer dcb) (char-code char))
   (when (not (K7-sending? dcb))
     (setf (K7-sending? dcb) T)
     (K7-send dcb))))

(defun K7-receive (dcb)                         ; LOWER-LEVEL DRIVER FUNCTIONS
  (without-interrupts
   (fast-enq (K7-in-buffer dcb) (%get (RBUF (K7-addr dcb))))
   (sem-signal (K7-in-sem dcb))))

(defun K7-send (dcb)
  (let ((count (sem-count (K7-out-sem dcb))))
    (cond ((= count (fast-q-size (K7-out-buffer dcb)))
           (setf (K7-sending? dcb) nil))
          (T (%store (fast-deq (K7-out-buffer dcb)) (XBUF (K7-addr dcb)))
             (cond ((> count (K7-watermark dcb))
                    (sem-signal (K7-out-sem dcb)))
                   ((= (incf (K7-pending dcb)) (K7-watermark dcb))
                    (setf (K7-pending dcb) 0)
                    (sem-signal (K7-out-sem dcb) (K7-watermark dcb))))))))

(defun K7-clear (dcb)
  (%store (logand slu-int-enable K7-init) (xcsr (K7-addr dcb))))
```

Figure 8. A portion of the driver code to be written by the Design Apprentice in response to dialog with programmer in Figure 7

The remainder of the interaction in Figure 7 illustrates the Design Apprentice's ability to detect and explain errors made by the programmer. These errors can be roughly divided into errors of omission (incompleteness) and errors of commission (inconsistency).

Incompleteness can be of two kinds. First, the specification may be missing some expected information, which if provided, would allow the Apprentice to finish the implementation. In this case, the Apprentice requests the needed information and proceeds. (The request concerning :CLEAR is an example of this case.) Second, the Apprentice may simply not have enough knowledge to implement a given specification. In this case, the programmer is asked to provide specific implementation instructions. (The request concerning :LINE-NUMBER is an example of this case. Note that the programmer postpones answering this question. This ability is important because it enables the programmer to control the interaction.)

Inconsistency can also be of two kinds. First, there may be inconsistency between different things the programmer says explicitly. Second, there may be inconsistency between what the programmer says and the knowledge contained in the clichés he invokes. (The last interaction in Figure 7 is an example of detecting an inconsistency between what the programmer says and a cliché.)

After the problems with the initial specification have been resolved, the Apprentice writes code for the K7 driver, portions of which are shown in Figure 8. This illustrates the Design Apprentice's ability to automatically make reasonable implementation choices. For example, the Apprentice has chosen watermark processing to increase the throughput of the small output buffers. (In watermark processing, no characters are entered into the output buffer unless a certain minimum amount of space—the watermark—is available.) The Apprentice has also made a number of reasonable decisions regarding the implementation of the various queues and semaphores needed in the driver.

Subsequent steps in the scenario (omitted here due to space limitations) illustrate that the programmer can use the Design Apprentice to change design decisions, introduce and remove instrumentation, add error checking, and make low-level additions to the code.

3.4 The Requirements Apprentice

Research on requirements acquisition is important for two reasons. From the perspective of artificial intelligence, it is a good domain in which to pursue fundamental questions related to knowledge acquisition. From the perspective of software engineering, studies have indicated that errors in requirements are more costly than any other kind of error. Furthermore, requirements acquisition is not currently well supported by software tools.

It is useful to distinguish several phases in the requirements acquisition process. The earliest phase usually takes the form of a "skull session," whose goal is to achieve consensus among a group of users about what they want. The requirements analyst's main role in this phase relies on his personal skills as a facilitator. The end product of this phase is typically an informal requirement.

Most current work on software requirements tools focuses on the later, *validation* phase. The main goal of this part of the process is to increase confidence that a given formal specification actually corresponds to the end user's desires. In current research approaches, this is achieved by applying simulation, symbolic execution, and various kinds of analysis to a formal specification. This work does not, however, address the key question of how a formal specification comes to exist in the first place.

Informality is an Essential Part of Human Thought

The focus of the Requirements Apprentice is on converting informal specifications into formal ones. This is a crucial gap in the current state of the art. For example, it was reported at the Second Workshop on Software Specification and Design that, in testing various requirements tools, the greatest problems stemmed from "the process of completing the system analysis work needed to translate the informal example specification into the appropriate input for the tools".

The following is a list of general features that characterize informal communication between a speaker (e.g. an end user) and hearer (e.g. an analyst):

- *Abbreviation.* Special terms (jargon) are used. The hearer is assumed to have a large amount of specific knowledge which explains the terms.
- *Ambiguity.* Statements can be interpreted in several different ways. The hearer has to disambiguate these statements based on the surrounding context.
- *Poor Ordering.* Statements are presented in the order they occur to the speaker, rather than in an order that would be convenient for the hearer. The hearer needs to hold many questions in abeyance until later statements answer them.
- *Incompleteness.* Aspects of the description are left out. The hearer has to fill in these gaps by using his own knowledge or asking questions.
- *Contradiction.* Statements which are true in the main are liable to be contradictory in detail. This reflects the fact that the speaker has not thought things out completely.
- *Inaccuracy.* For a variety of reasons, some of the statements are simply wrong.

These kinds of informality are not a matter of the speaker being lazy or incompetent. Informality is an essential part of the human thought process. It is part of a powerful debugging strategy for dealing with complexity, which shows up in many problem solving domains: Start with an almost-right description and then incrementally modify it until it is acceptable. Thus, dealing with informality is not just a question of being user-friendly—it is a fundamental prerequisite.

One of the goals of research on the Requirements Apprentice is to elaborate the initial characterization of informality given above, and to develop strategies and heuristics for removing these features from informal requirements.

What the Requirements Apprentice Knows: Requirements Clichés

Another goal of research on the Requirements Apprentice is the codification of clichés in the domain of software requirements. Such a taxonomy would be valuable even if it only existed as a textual handbook for use by a human analyst. Three examples of requirements clichés used in the Requirements Apprentice scenario are: *repository*, *information system*, and *tracking system*.

A repository is an entity in the physical world. The basic function of a repository is to ensure that items which enter the repository will be available for later removal. There are a variety of physical constraints which apply to repositories. For example, since each item has a physical existence, it can only be in one place at a time and therefore must either be in the repository or not.

There are several kinds of repositories. Simple repositories merely take in items and then give them out. A more complex kind of repository supports the lending of items, which are expected to be returned. Another dimension of variation concerns the items themselves. The items may be unrelated or they may be grouped into classes. Example repositories include: a storage warehouses (simple repository for unrelated items), a grocery store (simple repository for items grouped in classes), and a rental car agency (lending repository for items grouped in classes).

In contrast to the repository cliché, the information system cliché describes a class of programs rather than a class of physical objects. The intent of the information system cliché is to capture the commonality between programs such as personnel systems, bibliographic data bases, and inventory control systems. Roles of an information system include an *information schema*, a set of *transactions* which can create/modify/delete the data, a set of *reports* which display parts of the data, *integrity constraints* on the data, a *staff* which manage the information system, and *users* which utilize the information system.

A tracking system is a specialized kind of information system which keeps track of the state of a physical object (called the *target*). The target object is assumed to have a (possibly complex) state and to be subject to various physical operations which can modify this state. The information in the tracking system describes the state of the target object. The transactions modify this information to reflect changes in the target's state.

There are several kinds of tracking systems. A tracking system may follow several targets instead of just one. A tracking system may keep a history of past states of the target. A tracking system may operate based on direct observations of the state of the target or based on observations of operations on the target. Finally, a tracking system may participate in controlling the operations on the target, rather than merely observing them. Example tracking systems include: aircraft tracking systems (which track multiple targets based on direct observations of their position) and inventory control systems (which track a repository based on observations of operations which modify its contents and often exercise some control over what can be given out to whom.)

Capabilities Illustrated in the Requirements Apprentice Scenario

Figure 9 and Figure 10 are an excerpt from a longer scenario [13,10], showing an analyst using the Requirements Apprentice to develop the requirements for a library database system.

The first four commands in Figure 9 begin the process of building a requirement by defining the terms *library* and *copy of a book*. Based on these four commands and the contents of the tracking system and repository clichés, the Apprentice augments it's internal model of the evolving requirement in a number of ways.

For example, since the state of a repository is the collection of items it contains (in this case, copies of books), the Apprentice uses the constraints in the tracking system cliché to derive an information schema that provides fields for the three properties listed for copies of books. Also based on the constraints in the tracking system cliché, an expectation is created within the Apprentice that a set of transactions will be defined corresponding to the typical operations on a repository.

Note that the new terms are far from fully defined at this point. They are incomplete because many roles remain to be filled in. They are ambiguous because it is not yet clear which kind of tracking system is intended, or which kind of repository a library is. Since incompleteness and ambiguity are inevitable during the early stages of constructing a requirement, the Apprentice refrains from complaining at this point. It accepts information and performs inferences on a "catch as catch can" basis. However, if requested, the Apprentice can produce a list of currently unresolved issues (see the bottom of Figure 10) and can guide the analyst in finishing the requirement.

The next two commands in Figure 9 begin the definition of the transactions to be supported by the database. The key term in these commands is *removal*. Removal is one of the standard operations supported by a repository, i.e., taking an item out of the repository and giving it to a user of the repository.

The first command to trigger a response from the Apprentice is the analyst's attempt to define the add transaction. The Apprentice complains that, as currently defined, adding a copy of a book is exactly the same as returning a copy of a book. This contradicts the Apprentice's expectation that new terms should not be synonymous with old terms unless explicitly declared to be so. The analyst does not have to deal with this problem immediately—he could go on to other things leaving the problem pending. In this case, however, he decides to deal with the problem right

```
> Begin a requirement for a system called "LIBDB".
> LIBDB is a tracking system which tracks a "library".
> A library is a repository for "copies of books".
> A copy of a book has the properties:
   title - a text string,
   author - a person's name,
   ISBN number - a unique alphanumeric key.
> Display the transactions subsection.
> The "check out" transaction tracks the removal of a copy of a book.
> The "return" transaction is the inverse of check out.
> The "add" transaction tracks the addition of a copy of a book.
  Potential Problem: The add transaction is exactly the same as the return transaction.
> Explain the problem.
  (1) The add transaction is exactly the same as the return transaction.
  Because:
    (2) The add transaction tracks the addition of a copy of a book.
    (3) The return transaction tracks the addition of a copy of a book.
    Because:
      (4) The check out transaction tracks the removal of a copy of a book.
      (5) The return transaction is the inverse of the check out transaction.
> Retract (4).
> The check out transaction tracks the borrowing of a copy of a book.
  Note: The return transaction now tracks the return of copy of a book
      to a lending repository.
> The "remove" transaction tracks the removal of a copy of a book.
> The "remove all" transaction tracks the removal of every copy of a book.
  Potential problem:
    The term every suggests that copies of books are members of classes.
    If this is not the case, then the remove all transaction is exactly
    the same as the remove transaction.
> Book is a class with properties previously defined for copy of a book.
> Redefine a copy of a book to be an instance of a book.
> A copy of a book has the property:
   copy number - a number unique within the class.
```

Figure 9. Hypothetical dialog between an analyst and the Requirements Apprentice, excerpted from a longer target scenario illustrating the full capabilities of the Apprentice. Programmer input follows the > prompt.

Table of Contents

1. Introduction
   - 1.1 Purpose
   - 1.2 Scope
   - 1.3 Definitions, Acronyms, and Abbreviations
   - 1.4 References
   - 1.5 Overview
2. General Description
   - 2.1 Product Perspective
   - 2.2 Product Functions
   - 2.3 User Characteristics
   - 2.4 General Constraints
   - 2.5 Assumptions and Dependencies
3. Specific Requirements
   - 3.1 Functional Requirements
     - 3.1.1 Transactions
     - 3.1.2 Reports
   - 3.2 External Interface Requirements
   - 3.3 Performance Requirements
   - 3.4 Design Constraints
   - 3.5 Attributes
     - 3.6.1 Security
     - 3.6.2 Error handling

1.1 Purpose

The LIBDB system is a tracking system which tracks a library. A library is a repository for copies of books.

LIBDB records information about the title, author, and ISBN number of the copies of books in the library. Transactions are supported for tracking what happens to the repository. Reports are provided for obtaining information about the copies of books in the library.

3.1.1.1 Check Out

The "check out" transaction tracks the borrowing of a copy of a book from the library.

INPUTS: ISBN number and copy number.
OUTPUTS: none.
PRECONDITIONS: The copy of the book uniquely identified by the inputs must be in the roster of copies of books, which are in the library.
EFFECT ON THE INFORMATION STORE: The input is borrowed from the roster of copies of books which are in the library.
UNUSUAL EVENTS: If the input is not in the roster of copies of books which are in the library, then the information system is inconsistent with the state of the repository. A notation is made in the error log.
USAGE RESTRICTIONS: none.

Unresolved issues:
Should historical record keeping be added?
Should checking of user validity be added?
Should checking of staff member validity be added?

Figure 10. Portions of a requirements document to be produced by the Requirements Apprentice. This document results from the dialog in Figure 9

away.

Studying the explanation generated by the Apprentice at his request, the analyst realizes that he made an error earlier in the definition of check out. Checking out a book does not correspond to removing it from the repository, but rather to borrowing it from the repository. The analyst corrects the error by redefining the check out transaction.

The remaining portion of the scenario illustrates the Apprentice detecting what turns out to be a fundamental epistemological confusion on the part of the analyst—namely between books as a class and books as instances. The analyst decides that title, author and ISBN number are better thought of as properties of a class of books, with each copy as an instance. This conceptual reorganization is propagated by the Apprentice throughout the requirement.

The Apprentice Can Produce Standard Requirements Documentation

In current practice, the end product of requirements acquisition is typically a single written document which is produced by the analyst and used both by the end user (for validation) and by the system designer (as the starting point for design). Using the Requirements Apprentice, the essential end product of the requirements acquisition process will be a machine-manipulable representation of the requirement inside the Apprentice. In the long term, this internal representation will be accessed directly by other tools and components of the Programmer's Apprentice. In the short term, this information will be used to answer queries and to generate various documents for the requirements analyst, the end user, and the system designer.

An example of the kind of document the Apprentice can generate is shown in Figure 10. The structure of this document conforms to the ANSI/IEEE requirements document Standard 830-1984. Although parts of the document in Figure 10 show connected English sentences, most of the output generated by the Apprentice will look more like the point-form outlines. As in the case of understanding free-from English input, the syntactic aspects of good English are not a key concern here. What is important is deciding *what* to say by, for example, choosing the appropriate level of detail.

4 Other Applications

4.1 Automated Program Recognition

When confronted with the source code for a program, programmers are able to *recognize* what is happening in the program. They can identify the algorithms being used and construct a plausible top-down design for the program. If this human ability could be automated, it would have many important applications.

Within the Programmer's Apprentice, recognition would enhance program synthesis by making it possible to recognize opportunities for optimization. Recognition would also make it possible for the Apprentice to accept code that has been totally or partially edited by hand and maintain it as if it had been written using the Apprentice. (The current Implementation Apprentice is limited in its ability to do this because it supports only very weak recognition.)

As a separate module, automated program recognition can be applied to the computer-aided teaching of programming. Most current program tutoring systems use *ad hoc* techniques to recognize errors in code written by novice programmers. The effectiveness of such teaching systems would be greatly enhanced by more powerful automated recognition.

Another important application of program recognition is *program translation* [18]. In order to perform high quality translation from one language to another, you must come to an understanding of what the program does in its entirety. Attempting to translate a program on a line-by-line basis, if it is possible at all, leads to output that is awkward, because it is written in the style of the input language rather than the style of the output language. The output is also often inefficient, because global patterns have not been recognized, and the appropriate special forms in the output language have therefore not been used. Program recognition makes high quality translation possible by providing a way to understand what the program does as a whole.

In addition to its practical applications, program recognition is a worthwhile problem to study from a theoretical standpoint in artificial intelligence. It is a step towards modeling the way in which human programmers understand programs based on their accumulated programming experience.

A Prototype Program Recognition System

Wills [21] has implemented a prototype automatic program recognition system, based on a parsing approach. The system first translates the input program into the Plan Calculus and then applies a graph parsing algorithm developed by Brotsky [1]. The grammar used in the parsing is derived from a library of basic clichés for routine symbolic programming.

Wills' current prototype takes Common Lisp programs as input (but because it is based on the Plan Calculus, the approach is inherently programming-language independent). As a way of communicating the results of the recognition, the system produces a kind of program documentation (see Figure 11).

An important virtue of Wills' recognizer is that it recognizes when two programs are algorithmically equivalent, even though they greatly differ syntactically. For example, the system recognizes that the following program is equivalent to the program in Figure 11, even though it uses very different control and binding constructs. Exactly the same explanation is generated for the two programs, except that the cited variable names are different.

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

$$\Downarrow$$

```
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 lexicographically. 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 11. Wills' system analyzed the undocumented Common Lisp code above and automatically produced an explanation of its implementation in terms of a library of clichés.

```
(DEFUN HASH-TABLE-MEMBER (STRUC ELEM &AUX BKT)
  (SETQ BKT (AREF STRUC (HASH ELEM)))
  (LOOP DO
    (WHEN (NULL BKT) (RETURN NIL))
    (LET ((ENTRY (CAR BKT)))
      (IF (STRING<= ENTRY ELEM)
          (IF (EQUAL ENTRY ELEM) (RETURN T) (SETQ BKT (CDR BKT)))
          (RETURN NIL)))))
```

At the current time, Wills' recognizer can analyze programs containing nested expressions, conditionals, single- and multiple-exit loops, and some data structures. The recognizer cannot handle side-effects other than assignment to variables, nor can it analyze programs containing recursion, arbitrary data abstraction, or functional arguments. Extending the recognizer to handle these language features is one of our current goals.

A crucial question to be answered about Will's recognizer is how well it will perform when it is scaled up to operate on full-sized programs given a full-sized cliché library. An automated system which could "reverse engineer" 100–200 line programs would be an invaluable aid in maintaining the huge and persistent corpus of extant code. Even though these extant systems are millions of lines long, most individual routines are not over a few hundred lines.

The theoretical performance of Brotsky's parsing algorithm, upon which Wills' recognizer is based, is polynomial in the program size and linear in the library size. However, we expect there to be significant problems to solve before this level of performance can actually be achieved. To apply the recognizer to realistic programs, we will also need to add facilities to handle side-effects, recursion, data abstraction and some restricted cases of functional arguments.

4.2 Synchronizable Series Expressions

As part of the evolution of the Plan Calculus, Waters [14] undertook a study of the kinds of loops programmers actually write. This study revealed that approximately 80% of the loops programmers write can be constructed by combining a few common kinds of looping patterns. Based on this observation, Waters has developed a new type of functional language construct, called Synchronizable Series Expressions [19].

The advantages (with respect to conciseness, readability, verifiability and maintainability) of programs written in a functional style are well known. A simple example of the clarity of the functional style is provided by the Common Lisp expression at the top of Figure 12. This expression computes the sum of the squares of the integers from `M` to `N`. `Eup` (for Enumerate up) enumerates a series of integers. `Msquares` (for Map squares) computes the square of each integer in a series. `Rsum` (for Reduce sum) computes the sum of a series of integers.

```
(Rsum (Msquares (Eup M :TO N)))

                ⇓

(PROG (INDEX SQUARE SUM)
      (SETQ INDEX M)
      (SETQ SUM 0)
   LP (IF (> INDEX N) (GO E))
      (SETQ SQUARE (* INDEX INDEX))
      (SETQ SUM (+ SUM SQUARE))
      (SETQ INDEX (+ INDEX 1))
      (GO LP)
    E (RETURN SUM))
```

Figure 12. The Common Lisp implementation of Synchronizable Series Expressions transforms simple and easy-to-understand series expressions into highly efficient loop code.

The key conceptual feature of this expression is that it makes use of a series of values as intermediate quantities. For example, `Msquares` takes as its argument (conceptually at least) a series of integers and returns a series of the corresponding squares. This makes it possible to use functional composition to express a wide variety of computations that are usually written as loops.

Most programming languages support series of one form or another. However, with the notable exception of APL, series expressions are not used nearly as often as they could be. The main reason for this is that, as typically implemented, series expressions are extremely inefficient. Since loops can usually compute the same result much more efficiently, the overhead engendered by using series expressions is quite properly regarded as unacceptable in many situations.

A solution to the problem of the inefficiency of series expressions is to transform them, before evaluating them, into equivalent loops. For example, the expression at the top of Figure 12 can be transformed into the loop at the bottom.

Unfortunately, not all series expressions can be completely transformed into efficient loops. As a result, most approaches transform some expressions and only partially transform others. In this situation, programmers are still reluctant to use series expressions, because it is hard for them to know in advance which expressions will be efficient, and which will not.

An alternate approach is to guarantee that every series expression will be efficient, by restricting the class of expressions which can be written. Designing a successful set of restrictions requires a trade-off between *expressiveness* and *simplicity*. On one hand, the restrictions must permit a usefully large class of expressions. On the other hand, the restrictions must be straightforward enough that programmers can easily determine whether or not a given computation satisfies them.

```
X := SUM(SQUARES(INTEGERS(M,N)));
```

$$\Downarrow$$

```
declare
   INDEX,SQUARE,SUM : INTEGER;
begin
   INDEX := M;
   SUM := 0;
   loop
     if INDEX>N then exit;
     SQUARE := INDEX*INDEX;
     SUM := SUM+SQUARE;
     INDEX := INDEX+1;
   end loop;
   X := SUM;
end;
```

Figure 13. Hosting Synchronizable Series Expressions in the Ada language will allow the simple and easy-to-understand Ada statement above to be transformed into efficient loop code.

The restriction embodied in Synchronizable Series Expressions seems to strike a successful balance. The basic concept of the restriction is *synchronizability*: it must be possible to use each value in every series immediately after it is created. A Common Lisp macro package [20] has been implemented which supports Synchronizable Series Expressions by automatically transforming them into efficient loops before evaluating them. Using this package, programmers incur *no* runtime overhead as compared to writing the loops themselves. The package has been in use in the Laboratory for over a year.

The order-of-magnitude productivity benefit of using Synchronizable Series Expressions instead of loops can be appreciated by comparing the two segments of code in Figure 12. However, our experience this past year suggests that the benefits go beyond this. Presently, most programs contain at least one loop, and most of the interesting computation occurs in loops. This is unfortunate, because loops are generally acknowledged as being the hardest parts of a program to understand. If series expressions were used whenever possible, most programs would not contain any loops.

The concept of Synchronizable Series Expressions is inherently language-independent. Figure 13 illustrates how the facility could be be hosted in the Ada language. Note that hosting Synchronizable Series Expressions would not require any extensions to Ada. It is possible to write an Ada package which supports (albeit inefficiently) synchronizable series operations as ordinary Ada functions. In addition, an optimizer (also written in Ada) can be written, which converts Synchronizable Series Expressions into efficient loops.

References

[1] D. Brotsky. *An Algorithm for Parsing Flow Graphs.* Technical Report 704, MIT Artificial Intelligence Lab., March 1984. M. S. Thesis.

[2] Y. A. Feldman and C. Rich. Reasoning with simplifying assumptions: A methodology and example. In *Proc. 5th National Conf. on Artificial Intelligence*, Philadelphia, PA, August 1986.

[3] D. A. McAllester. *Reasoning Utility Package User's Manual.* Memo 667, MIT Artificial Intelligence Lab., April 1982.

[4] C. Rich. *Inspection Methods in Programming.* Technical Report 604, MIT Artificial Intelligence Lab., June 1981. PhD Thesis.

[5] 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.

[6] C. Rich. Knowledge representation languages and predicate calculus: How to have your cake and eat it too. In *Proc. 2nd National Conf. on Artificial Intelligence*, Pittsburgh, PA, August 1982.

[7] 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.

[8] C. Rich. Inspection methods in programming: Clichés and plans. *Artificial Intelligence*, 1988. Submitted.

[9] C. Rich and H. E. Shrobe. Initial report on a LISP Programmer's Apprentice. *IEEE Trans. Software Engineering*, 4(6), November 1978. Reprinted in D. Barstow, E. Sandewall, and H. Shrobe, editors, *Interactive Programming Environments*, McGraw-Hill, 1984.

[10] C. Rich and R. C. Waters. *Towards a Requirements Apprentice: On the Boundary Between Informal and Formal Specifications.* Memo 907, MIT Artificial Intelligence Lab., July 1986. NSF Proposal.

[11] C. Rich and R. C. Waters. Formalizing reusable software components in the Programmer's Apprentice. In T. Bigerstaff and A. Perlis, editors, *Software Reuse*, Addison-Wesley, 1987. Also published as AIM-954.

[12] C. Rich and R. C. Waters. *The Programmer's Apprentice: A Program Design Scenario.* Memo 933A, MIT Artificial Intelligence Lab., November 1987.

[13] 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.

[14] R. C. Waters. A method for analyzing loop programs. *IEEE Trans. Software Engineering*, 5(3):237–247, May 1979.

[15] R. C. Waters. *KBEmacs: A Step Towards the Programmer's Apprentice.* Technical Report 753, MIT Artificial Intelligence Lab., May 1985.

[16] R. C. Waters. The Programmer's Apprentice: A session with KBEmacs. *IEEE Trans. Software Engineering*, 11(11):1296–1320, November 1985.

[17] R. C. Waters. KBEmacs: Where's the AI? *AI Magazine*, 7(1), Spring 1986.

[18] R. C. Waters. Program translation via abstraction and reimplementation. *IEEE Trans. Software Engineering*, Summer 1987. (To appear).

[19] R. C. Waters. Efficient interpretation of synchronizable series expressions. *ACM Sigplan Notices*, 22(7):74–85, July 1987. *Proc. ACM SIGPLAN '87 Symposium on Interpreters and Interpretative Technique.*

[20] R. C. Waters. *Synchronizable Series Expressions: User's Manual for the OSS Macro Package.* Memo 958, MIT Artificial Intelligence Lab., November 1987.

[21] L. M. Wills. *Automated Program Recognition.* Technical Report 904, MIT Artificial Intelligence Lab., September 1986. M.S. Thesis.