MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

Seven Layers of Knowledge Representation and Reasoning in Support of Software Development

Charles Rich, Yishai A. Feldman

TR92-01 February 1992

Abstract

This paper summarizes our experience in the Programmerś Apprentice project in applying knowledge representation and automated reasoning to support software development. We describe a system, called Cake, that comprises seven layers of knowledge representation and reasoning facilities: truth maintenance, boolean constraint propagation, equality, types, algebra, frames, and Plan Calculus. We also include sessions with two experimental software development tools implemented using Cake: the Requirements Apprentice and the Debugging Assistant.

IEEE Transactions on Software Engineering, Vol. 18, No. 6, June 1992, pp. 451-469. Special Issue on Knowledge Representation and Reasoning in Software Development

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, Inc., 1992
201 Broadway, Cambridge, Massachusetts 02139

MITSUBISHI ELECTRIC RESEARCH LABORATORIES
CAMBRIDGE RESEARCH CENTER

Technical Report 92-01 February 1992

Seven Layers of Knowledge Representation and Reasoning in Support of Software Development

Charles Rich Yishai A. Feldman*

Abstract

This paper summarizes our experience in the Programmer's Apprentice project in applying knowledge representation and automated reasoning to support software development. We describe a system, called Cake, that comprises seven layers of knowledge representation and reasoning facilities: truth maintenance, boolean constraint propagation, equality, types, algebra, frames, and Plan Calculus. We also include sessions with two experimental software development tools implemented using Cake: the Requirements Apprentice and the Debugging Assistant.

To appear in IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Development

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories of Cambridge, Massachusetts; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories. All rights reserved.

Copyright © Mitsubishi Electric Research Laboratories, 1992
201 Broadway; Cambridge, Massachusetts 02139

*Department of Computer Science, Tel Aviv University, Israel.

Publication History:–

1. First printing, TR92-01, February 1992.
2. *IEEE Trans. Software Engineering*, Vol. 18, No. 6, June 1992, pp. 451–469.

I. Introduction

This paper summarizes our experience in the Programmer's Apprentice project [1, 2] in applying knowledge representation and automated reasoning to support software development. Our basic conclusion is that there currently exists a collection of knowledge representation and automated reasoning facilities that are both feasible and useful to support the next generation of software development tools.

We support the claim that the facilities presented here are feasible by describing their implementation in a prototype knowledge representation and reasoning system called Cake. We support the claim that the facilities are useful by describing how Cake has been used in the construction of two demonstrations of the next generation of software tools: the Requirements Apprentice and the Debugging Assistant. We do not believe that the facilities presented here are the only ones that will be needed; however, they are a good starting point.

The organization of this paper is as follows. The rest of the introduction discusses general characteristics of future software development tools that motivate Cake, followed by an overview of the system and the key issues in its architecture. Section II presents sessions with two experimental tools implemented on top of Cake, illustrating the need for particular knowledge representation and automated reasoning capabilities. Sections III through IX, which comprise the bulk of the paper, describe each of the seven layers of Cake in detail. Section X concludes with a discussion of future directions.

A. Software Development Tools

Cake was developed to support research toward an intelligent, interactive software development tool, called the Programmer's Apprentice, which will assist software engineers in all phases of the programming process. The following are three key characteristics of such a tool and their implications for knowledge representation and reasoning.

- *Knowledge-Intensive:* Software engineers, like engineers in other disciplines, seldom reason from first principles. Rather, they rely whenever possible on their experience with similar problems. In particular, good software engineers try to reuse parts of solutions with which they are already familiar (either first hand or learned in school). Thus a systems analysis tool that is nothing more than an electronic "blank slate" for box-and-arrow diagrams is not as good as one that comes with a well-thought-out library of generic diagrams for various applications at various levels of abstraction plus knowledge about how to choose the appropriate diagram and adapt it to the problem at hand.

  Supporting knowledge-intensive tools will require powerful facilities for representing structured artifacts, such as programs, specifications, and requirements, at various levels of abstraction

- *Intelligent Assistance:* Rather than simply accepting and executing commands, an intelligent assistant checks the reasonableness of decisions, fills in missing details, and requests advice about how to carry out complex operations. These abilities can contribute to both a software engineer's productivity and the reliability of the final product. Another hallmark of an intelligent assistant is the ability to explain its actions and decisions in terms that the user can understand. This allows the user to check what the tool has done. It also allows the tool to describe the problems it has encountered when it requires advice.

  Supporting intelligent assistance requires flexible and powerful automated reasoning over a wide range of mathematical domains that arise in software development, such as boolean logic, types, and algebra. However, many apparently simple reasoning problems, such as checking whether a set of boolean propositions is consistent, are completely solved only by exponential algorithms. What this means, in our view, is that the appropriate role for automated deduction is not as the "main engine" of software tools (as for example, in the programs-as-proofs/deductive synthesis approach [3]), but rather as a supporting function that increases productivity and reliability.

- *Evolutionary:* Every aspect of software development is colored by the need for continual evolution. Requirements change because the world changes and because it is not possible for requirements analysts or end users to foresee all of the opportunities for a system's use. Designs change because requirements and technology change. Implementations change because designs change and bugs have to be fixed.

  The representations required for knowledge-intensive tools are large and the automated reasoning required for intelligent assistance is expensive. Supporting evolution therefore requires special attention to issues of incremental computation.

B. Overview of Cake

The architecture of Cake comprises the seven layers shown in Fig. 1. The bottom six layers provide the following "generic" knowledge representation and automated reasoning facilities:

- The *truth maintenance* layer stores and maintains dependencies between facts the system has been told or has deduced. It supports incremental recomputation of results and simple explanation (using the dependencies as a trace of the system's reasoning).
- The *boolean constraint propagation* layer performs an efficient but incomplete form of general-purpose logical deduction. It also detects "shallow" contradictions.

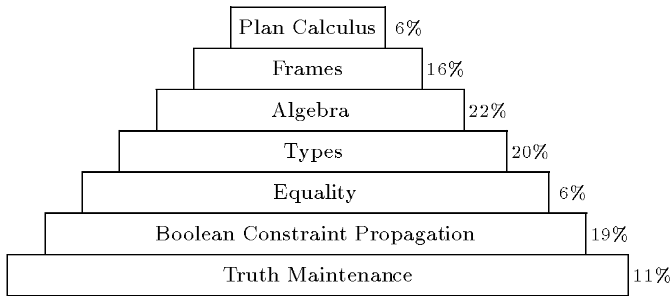

Fig. 1. Seven layer architecture of Cake showing the approximate percentage of the total 17,000 lines of code in each layer.

- The *equality* layer provides a complete reasoner[1] for equality. Full dependency information is recorded so that equalities can be asserted and retracted.
- The *types* layer implements a very flexible version of typed logic. New types can be defined by specialization, intersection, union, and complement.
- The *algebra* layer contains special-purpose decision procedures for common algebraic properties of operators, such as commutativity, associativity, and transitivity. It also provides limited reasoning facilities for sets and quantifiers.
- The *frames* layer implements the standard frame notions of inheritance and slot retrieval within a dependency-directed framework. It also has the capability of reasoning symbolically using frame constraints.

The top layer of Cake, the *Plan Calculus*, is an example of the use of these generic facilities to implement a specialized formalism for software development. The Plan Calculus supports representation and reasoning about programming concepts, such as input-output specifications, data flow, and control flow, in a programming-language independent form.

Each layer in Cake is implemented using only facilities in the layers below. The order of the layers is an outgrowth of attempting to maximize the reuse of already implemented facilities to implement the next layer.

The original kernel of Cake was a truth maintenance system with boolean constraint propagation and equality (the bottom three layers in Fig. 1) developed by McAllester [4] in 1982. This was modified and extended by the authors resulting in a system called BREAD (for "Basic REAsoning Device") completed in 1984. The next three layers of Cake were built on top of BREAD between 1984 and 1987 to provide an additional range of broadly applicable facilities, such as frames and algebraic reasoning. These six layers together are called FRAPPE (for "FRAmes in a ProPositional Environment") [5, 6].

Cake is implemented in Common Lisp. Each question-response interaction in the various examples below takes from a fraction of a second to several minutes on a Sun-4 workstation.

[1] A complete reasoner is a procedure that can prove any true theorem in its language. For an incomplete reasoner, there are true theorems that cannot be proved.

C. Control of Reasoning

Cake presents itself to a software tool developer as an "active database," i.e., a database that monitors certain integrity constraints and automatically invokes certain reasoning processes whenever relevant data appears. A key architectural issue throughout Cake, therefore, is what kinds of reasoning should happen spontaneously (i.e., in immediate response to the appearance of data) as opposed to being explicitly invoked and controlled by some higher-level process. The need for this division stems from the combinatorially explosive nature of many automated reasoning algorithms coupled with the fact that an intelligent assistant must not "go away" for long and upredictable periods of time.

In general, we restrict the spontaneous reasoning in Cake to algorithms that have linear or close to linear cost. Examples of spontaneous reasoning in Cake include maintenance of well-founded support in the truth maintenance layer (linear time) and the congruence closure algorithm ($n \log n$) in the equality layer. The final tuning of what is done spontaneously depends of course on the computing resources available to the system and needs to be re-evaluated whenever new algorithms are discovered. However, we expect the basic need for this division to exist for the forseeable future.

It is also desirable for the spontaneous reasoning algorithms to always give the same answer independent of the order in which information appears. All of the spontaneous reasoning in Cake has this property.

Sometimes, if the complete algorithm for some kind of reasoning is too expensive to be done spontaneously, it makes sense to break it into two parts: one part that solves a limited part of the problem efficiently, and another part that can solve the complete problem under explicit control. The intuition underlying this strategy is that an intelligent assistant should be able to make "obvious" deductions spontaneously, whereas you are willing to wait for the answer to more a difficult question. (The general question of what deductions are "obvious" and how to tailor a reasoning procedure to efficiently produce only those deductions is a deep and interesting one.)

The best example of this approach in Cake is the way propositional reasoning is handled. A complete decision procedure for propositional logic is exponential in the worst case. The boolean constraint propagation layer provides an efficient (linear time) but incomplete kind of spontaneous propositional reasoning, together with a special procedure (`Ask`—see Section IV.B) that can be used to explicitly request more complete reasoning when desired. A similar approach is taken to reasoning about types.

When explicit control of reasoning is required, it can come from several sources. Sometimes there is enough context in the application task that the tool developer can forsee where to insert the appropriate calls to control procedures such as `Ask` (this is true in a number of places

in the two experimental tools described below). Alternatively, context-dependent heuristics may be used. For example, Section VIII.B describes a heuristic for expanding frame type definitions based on the appearance of slot expressions. At worst, responsibility for explicit control of expensive reasoning can be passed through to the end user, in which case a tool degrades into a "guided proof" mode of use.

D. Hybrid Representation and Reasoning

A hybrid knowledge representation and reasoning system is one in which two or more fundamentally different algorithms and data abstractions are used. In this sense, it is fair to say that all practical knowledge representation and reasoning systems are in fact hybrid. For example, no practical system tries to deduce that $2+2=4$ from the axioms of arithmetic; instead, systems like Cake call the addition procedure of the underlying programming language. The motivation for a hybrid architecture is thus to take advantage of specialized methods for many common kinds of reasoning that are much more efficient than a single universal method can ever be.

Each layer of Cake introduces new data abstractions and algorithms. The major lesson we learned from the hybrid architecture of the system is the difficulty of debugging all of the interactions between reasoners on a case-by-case basis. For example, for some pairs of complete reasoners (e.g., equality and transitivity), we wanted to make sure that the resulting combination was a complete reasoner for the union theory. This analysis was complicated by the fact that the relevant facts may come and go in any order via the truth maintenance system. In the case of combining a complete reasoner with an incomplete reasoner, (e.g., equality and boolean constraint propagation), we had difficulty giving a principled description of the resulting incomplete reasoner. We also spent a lot of effort on the interaction between equality and pattern-directed invocation [7]. (A subsidiary moral here is that equality interacts with everything!)

A strong solution to the problems of hybrid architectures would be a simple interface specification that guaranteed completeness whenever a new reasoner was added. Examples of such interfaces exist in restricted settings, such as combining decision procedures for disjoint theories [8] and the addition of sort reasoners to unification-based systems [9]. Neither of these frameworks, however, deals with combining complete and incomplete reasoners, or includes dependencies and changing beliefs as part of the interface.

Barring the invention of a general hybrid architecture with strong completeness guarantees, the only other promising approach for a system like Cake is a framework that provides a standard protocol for inference within each reasoner [10] and for communication between reasoners [11].

E. Retraction and Efficiency

When we first started to implement Cake, we embraced evolution and incremental computation to the fullest extent possible: *every* fact in Cake was retractable and fully supported by dependencies. This led to a system that, although it was very "clean," was also unbearably slow. Upon further reflection, we realized that a lot of the system's resources (time and memory) were being expended on making it possible to retract, for example, the commutativity of addition, or the fact that integers are a subtype of numbers.

The current implementation of Cake therefore has in many places two parallel mechanisms for the same kind of reasoning: an efficient non-retractable version and a more expensive retractable one. For example (see Section VII.A), an operator can either be permanently (i.e., non-retractably) defined to be commutative, or its commutativity can be asserted as a fact that can later be retracted.

The availability of two parallel mechanisms with small granularity (individual facts) provides the tool developer with a spectrum of choices in the trade-off between retraction and efficiency, which we have found very useful. However, it also led to a significant increase in the complexity of the implementation of Cake, since it is really a kind of hybridization—not two mechanisms for two different theories as discussed above, but two mechanisms for the same theory under different conditions. In addition to the effort of implementing all the extra mechanism for the non-retractable version, we also had to bear the cost of making sure that the retractable and the non-retractable versions interact properly.

II. Example Tool Sessions

To illustrate the kind of software development tools that motivate Cake, this section presents sessions with two experimental tools that have been implemented as part of the Programmer's Apprentice project using Cake. See [12, 13] for a description of a third tool based on Cake (the Design Apprentice), which was only partially implemented.

The typographic conventions used in this section and the rest of the paper are as follow. What the user types is shown following the > prompt. What the system prints back is shown indented below. (If what the system prints back is unimportant in the current context, it is omitted to save space.) Editorial comments are shown in italics.

A. The Requirements Apprentice

The Requirements Apprentice (RA) is an intelligent assistant for software requirements acquisition and analysis. The focus of the RA is on the *formalization* phase of software requirements, i.e., the process by which informal descriptions become formal ones. The kinds of informality the RA deals with include: abbreviation, ambiguity, poor ordering, contradiction, incompleteness, and inaccuracy.

The motivation and technical contributions of the RA are well described elsewhere [14]. In this section, we will

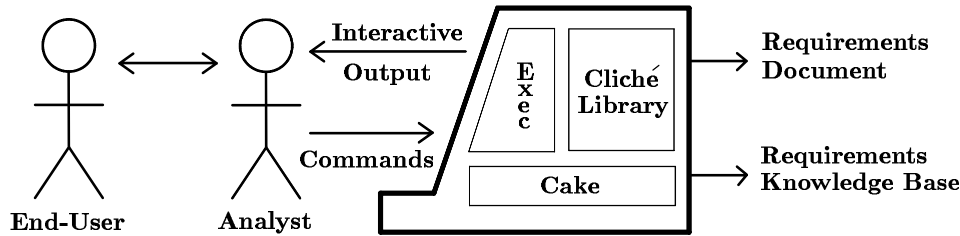

Fig. 2. Architecture of the Requirements Apprentice showing the role of Cake.

concentrate instead on the knowledge representation and automated reasoning facilities upon which it relies. To avoid distraction with the syntactic details of the RA's input language, what the user types in the session below has been replaced by editorial paraphrases. The corresponding literal transcript can be found in [14].

Fig. 2 shows the role of the RA in relation to other agents involved in the requirements process. Note that the RA does not interact directly with an end user, but is an assistant to a requirements analyst.

The RA produces three kinds of output. Interactive output notifies the analyst of conclusions drawn and problems detected as information is being entered. A requirements knowledge base represents everything the RA knows about the evolving requirement. Finally, the RA can create a more or less traditional requirements document summarizing the current state of the knowledge base.

Internally, the RA is composed of three parts: Cake provides the basic knowledge representation and automated reasoning facilities. The *executive* (Exec) contains algorithms and data structures that are specific to the RA and provides explicit control of reasoning for Cake. The *cliché library* contains reusable fragments of requirements and associated domain models, represented as a frame hierarchy. This demonstration system does not emphasize the user interface.

The example session below is based on a requirements benchmark [15] dealing with the specification of a university library database. Although the RA's cliché library contains many concepts that will be relevant, the RA doesn't know anything about libraries per se at the start of the session.

1>*"University Library Database"* (ULDB) *is a system.*

2>*"University Library"* (UL) *is an instance of "library".* UL *is part of the environment.*

The distinction between the system being specified and the environment (which cannot be changed) is part of the RA's basic epistemology.

The simple initial statement of the requirement is that the database "track" the state of the (actual) library. There are several different versions of tracking specifications in the cliché library. The type of tracking system being specified will be refined as the session progresses.

3>ULDB *tracks* UL.
```
  ULDB Is-Instance-Of Tracking-System.
```

At this point, a number of issues are pending. The RA must accept information in any order presented by the analyst (e.g., uses of words before their definition) and come to the same conclusions. The third pending issue below says that the RA needs to know the possible states of the library.

4>*Show pending issues.*
```
  1- Need-Further-Disambiguation of Tracks.
  2- Need-Definition of "Library".
  3- Need-Definition (Item-States UL).
```

A central concept in the remainder of this session is *repository*, represented as a frame in the RA's cliché library with the following slots: the *type of items* stored in it, the *patrons* that utilize it, and the *staff* that manages it. Two key operations on a repository are *adding* and *removing* items.

5>*"Library" is a kind of repository, with items of type "book."*

Notice that the new type *book* is introduced above. The analyst now specifies some properties of this type.

6>*"Book" is a kind of physical object, with slots "title," "author," and "ISBN."*

7>*"ISBN" is a kind of integer.* ISBN *uniquely identifies a book.*

The analyst now begins to define various types of transactions on the database. As more information is accumulated, the RA is able to further refine the type of ULDB to be a tracking *information* system (because it *records* information about the history of the tracked object, rather than just providing the current state).

8>*"Check out" is a transaction that records removing books from the library.*
```
  ULDB Is-Instance-Of Tracking-Information-System.
```

9>*"Check in" is the inverse of check out.*
```
  "Check-In" Is-Instance-Of Action-Tracking-Opn.
```

Using simple algebraic reasoning, the RA is able to deduce quite a lot of information from the statement that check in is the inverse of check out (i.e., it tracks the inverse state change).

The first serious problem detected by the RA in this session arises when the analyst defines two further transactions below.

10>*"Unshelf" is a transaction that records removing books from the library.*

11>*"Unshelf all" is a transaction that records removing books from the library. Unshelf all removes the set of all books with a given* ISBN.
```
    **Conflict #1
    **Colliding-Definitions Unshelf And Unshelf-All
```

Based on what it has been told so far, the RA is able deduce that the *unshelf* transaction and the *unshelf all*

transaction are identical: Since ISBN uniquely identifies a book, there can never be two books with the same ISBN. Therefore the set of books removed by unshelf all must contain exactly one book. The RA has a built-in expectation that terms should not be synonymous unless they are explicitly defined to be synonyms.

In response to the analyst's request for an explanation of the problem, the RA prints out a listing (mostly elided below) of the dependency tree underlying the problematic conclusion. In the part of the tree shown, we can see boolean logic and equality reasoning; the full tree includes deductions from all layers of Cake. Unfortunately, the full tree is much too large and hard to understand to be useful to the analyst in raw form.

```
12>Explain.
   1. (Equal-Defs Unshelf Unshelf-All) is True
      by Implication from:
      1. (Implies
          (And (Tracking-Operation Unshelf)
               (Tracking-Operation Unshelf-All)
               (= Frame-I4 Frame-I5))
          (Equal-Defs Unshelf Unshelf-All))
      2. (And (Tracking-Operation Unshelf)
              (Tracking-Operation Unshelf-All)
              (= Frame-I4 Frame-I5)) is True
         by Conjunction from:
         1. (= Frame-I4 Frame-I5) is True
            by Equality from:
     ...
   **Premises:
   1. (= (Records Unshelf-All) Remove)
   2. (= (Objects Unshelf-All)
         (!The-Set-Of-All (?B)
           Such-That (= (ISBN ?B) $Input)))
   3. (= (Records Unshelf) Remove)
   4. (Default 23
       (= (Objects Unshelf)
          (!The (?O) Such-That
                (= (ISBN ?O) $Input)))
   ...
```

A slight improvement over the full dependency tree is the list of premises shown above. Premises corresponding to statements made by the analyst are printed first, followed by assumptions that come from the cliché library. Unfortunately, even the premise list can become quite long. (In this example, the untruncated list has over twenty entries.)

It is often a good idea to ignore a conflict or contradiction for a while until clarifying information comes along. If the analyst chooses to ignore this conflict, it will be added to the list of pending issues.

At this point, however, the analyst realizes that the requirement thus far reflects a token/type confusion between the notion of a physical copy of a book and the notion of a book as a class—i.e., a set of copies with the same ISBN. (This confusion actually appears in the benchmark on which this session is based.) To fix the problem, a new term denoting "copy of book" must be introduced. In addition, something has to be done about the fact that some uses of "book" in the statements above refer to the old concept and some refer to the new concept.

The RA provides the `Reformulate` command demonstrated below to assist with this kind of incremental change. The command displays all the statements that contained the term in question and asks the analyst to choose where substitution should occur. The RA then retracts the obsolete statements and asserts the new ones, relying on Cake's truth maintenance facilities to preserve all of the appropriate deductions.

```
13>Reformulate using "copy of book" in place of book.
   5. "Library" is a kind of repository, with items of
    type "book."
     **Reformulate? >Yes

   6. "Book" is a kind of physical object, with slots "title,"
    "author," and "ISBN."
     **Reformulate? >Yes

   7. "ISBN" is a kind of integer. ISBN uniquely identifies
    a book.
     **Reformulate? >No
```

In the interests of brevity, the session is truncated at this point.

B. The Debugging Assistant

The Debugging Assistant (DEBUSSI) is an experimental tool that helps a programmer localize a bug, i.e., identify a particular procedure call or section of code that needs to be changed.

As in the previous section, the session presented here is extracted from a longer document [16], omitting most of the motivation and technical details not relating specifically to the use of Cake's facilities. In particular, the localization algorithm, which is the key contribution of DEBUSSI from the standpoint of research in debugging, is only superficially described here.

Fig. 3 shows DEBUSSI's overall architecture. DEBUSSI's main input is source code. As shown in the lower left corner of the figure, a program analyzer translates the source code into the Plan Calculus by analyzing the data and control flow. In addition, DEBUSSI may have access to some incomplete specification information, either in the form of explicitly provided test cases or as a byproduct of earlier requirements and design activities using tools like the Requirements Apprentice and the Design Apprentice.

A program bug manifests itself in Cake as a contradiction between the behavior of the program and its specifications. DEBUSSI commences bug localization when such a contradiction arises. DEBUSSI generates an initial set of suspects (procedure calls) by retrieving the premises underlying the contradiction in Cake's dependency structure.

Some of the suspects will typically be conditionals, or "splits" in control flow. DEBUSSI attempts to exonerate splits by performing a special kind of reasoning-by-cases using Cake's facilities for asserting and retracting premises. (This is an example of explicitly controlled reasoning.)

Dependency analysis and split analysis will usually exonerate some, but not all, of the suspects. At this point DEBUSSI requires more specification information, so it queries the user about one of the remaining suspects (heuristics are used to choose which one). The new information obtained in the query will typically result in new deductions in Cake and therefore a changed dependency structure. Analysis of the new dependencies may exoner-

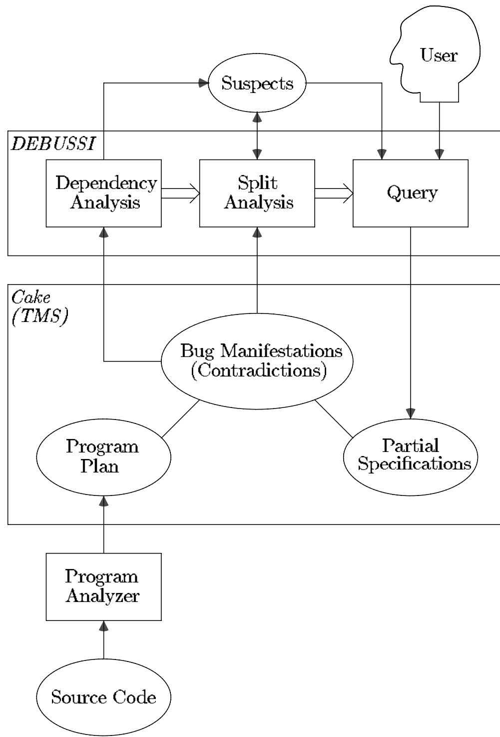

Fig. 3. Architecture of the Debugging Assistant showing the role of Cake.

ate further suspects.

DEBUSSI iterates through the steps of dependency analysis, split analysis, and querying until it finds itself with one suspect or no suspects. If one suspect remains and it represents a call to a system primitive (such as `Car` in Lisp), then DEBUSSI concludes that the bug is due to an incorrect use of that primitive.

If one suspect remains and it represents a call to a user-defined procedure, then DEBUSSI attempts to further localize the bug. DEBUSSI "zooms in" on the procedure call by expanding its definition, and continues debugging the expanded definition. If DEBUSSI is unable to localize the bug after zooming in (it ends having no suspects), then it concludes that the bug is at some undetermined place within the enclosing procedure call.

Unlike the Requirements Apprentice, DEBUSSI is not a knowledge-intensive tool—there is no reusable library component in DEBUSSI. Much other work in this area (e.g., [17, 18]) has explored the use of predefined bug patterns. DEBUSSI is an attempt to support the kind of general-purpose bug localization that people do in unfamiliar situations.

The following transcript illustrates the use of DEBUSSI to find the bug in the unification program in Fig. 4. A much better user interface, which displays the code being debugged with helpful highlighting and underlining was designed for DEBUSSI (see [16]), but is omitted here to save space.

The toplevel procedure in Fig. 4 is `Unify`. Its inputs are two patterns and an environment (an initial list of bindings). Patterns are represented as lists. Variables in

```
(Defun Unify (P1 P2 Env)
  (Cond ((Eq Env 'NoMatch) 'NoMatch)
        ((VariableP P1)
         (If (VariableP P2)
             (Var-Var-Match P1 P2 Env)
           (Extend-If-Possible P1 P2 Env)))
        ((VariableP P2)
         (Extend-If-Possible P2 P1 Env))
        ((ConstantP P1)
         (If (ConstantP P2)
             (If (Same-Constant P1 P2)
                 Env
               'NoMatch)
           'NoMatch))
        ((ConstantP P2) 'NoMatch)
        (T (Unify (Cdr P1)
                  (Cdr P2)
                  (Unify (Car P1) (Car P2) Env)))))

(Defun Var-Var-Match (V1 V2 Env)
  (If (Eq V1 V2)
      Env
    (Let ((B1 (Lookup V1 Env))
          (B2 (Lookup V2 Env)))
      (If (Null B1)
          (If (Null B2)
              (Extend V1 V2 Env)
            (Unify V1 (Binding-Val B2) Env))
        (Unify (Binding-Val B1)
               (If (Null B2)
                   V2
                 (Binding-Val B2))
               Env)))))

(Defun Extend-If-Possible (Var Val Env)
   (Let ((Value-Cell (Lookup Var Env)))
     (If (Null Value-Cell)
         (If (Freeof Var Val Env)
             (Extend Var Val Env)
           'NoMatch)
       (Unify (Binding-Val Value-Cell) Val Env))))

(Defun Freeof (Var E Env)
  (Cond ((ConstantP E) T)
        ((VariableP E)
         (If (Equal Var E)
             Nil
           (Let ((B (Lookup E Env)))
             (If (Null B)
                 T
               (Freeof Var (Binding-Val B) Env)))))
        ((Freeof Var (Car E) Env)
         (Freeof Var (Cdr E) Env))))

(Defun Lookup (Var Env)
  (Cond ((Null Env) Nil)
        ((Equal Var
                (Binding-Var (Car Env))) (Car Var))
        (T (Lookup Var (Cdr Env)))))

(Defun Extend (Var Val Env)
  (Cons (New-Binding Var Val) Env))

(Defun ConstantP (X) (Atom X))

(Defun Same-Constant (X Y) (Eq X Y))

(Defun VariableP (X)
  (And (ConsP X) (Eq (Car X) '?)))

(Defun New-Binding (Var Value) (Cons Var Value))

(Defun Binding-Var (Cell) (Car Cell))

(Defun Binding-Val (Cell) (Cdr Cell))
```

Fig. 4. Common Lisp source code for unification program with bug.

patterns are represented as lists whose first element is a question mark. `Unify` is supposed to return either a list of bindings that unify the two given patterns, or the atom `NoMatch` if the patterns are incompatible. A binding is represented as a dotted pair whose `Car` is a variable. No specification information is provided to DEBUSSI prior to the session below.

The following are three examples of correct test cases for `Unify`.

```
>(Unify '(F (? X) (? Y)) '(F 3 4) Nil)
 (((? X) . 3) ((? Y) . 4))
>(Unify '(F (? X)) '(G 3) Nil)
 NoMatch
>(Unify '(F 1) '(F 1) Nil)
 Nil
```

However, the following test case reveals the presence of a bug.

```
>(Unify '((? X)) '((? X)) Nil)
 ((? X) . (? X))
```

The correct output here is `Nil`. To appreciate the assistance provided by DEBUSSI in the transcript below, the reader at this point is encouraged to examine the code in Fig. 4 to try to find the cause of the incorrect output.

DEBUSSI is invoked by informing it of the correct output for the preceding test case.

```
1>(Correct-Output-Is Nil)
```

DEBUSSI translates the code in Fig. 4 into the Plan Calculus and discovers that the (incorrect) output value in this test case depends (among other things) on the underlined recursive call to `Unify`. To allocate blame, it must ask if the arguments to this call to `Unify` are correct (for this test case). If the arguments are correct, then this procedure is to blame. If they are not, then the problem is earlier in the code.

```
  **Within UNIFY,
  procedure call (Unify Nil Nil '(((? X) . (? X))))
  returned incorrect value: (((? X) . (? X)))
  Was UNIFY called correctly here?
```

Since the correct environment to be returned in this test case is `Nil`, a non-`Nil` environment list, such as the third argument above, should never be created. Therefore the procedure call above is incorrect. The user informs DEBUSSI of this fact as shown below.

```
2>No
  Enter violated condition:
3>(Not (Eq Arg3 Actual3))
```

The incorrect environment list above was produced by the other (non-underlined) recursive call to `Unify`. DEBUSSI therefore queries about whether *it* was called correctly.

```
  **Within UNIFY,
  procedure call (Unify '(? X) '(? X) Nil)
  returned incorrect value: (((? X) . (? X)))
  Was UNIFY called correctly here?
4>Yes
```

The arguments to this procedure call are correct, so DEBUSSI expands `Unify` again (this time with different arguments). In this expansion, DEBUSSI is able to eliminate (using split analysis) all of the suspects except for `Var-Var-Match`.

```
  There is a bug within UNIFY.
  Zooming in...
  There is a bug within VAR-VAR-MATCH.
  Zooming in...
```

DEBUSSI therefore expands the call to `Var-Var-Match` and then queries about one of the procedure calls on which its output depends in this test case, namely `Extend`.

```
  **Within VAR-VAR-MATCH,
  procedure call (Extend '(? X) '(? X) Nil)
  returned incorrect value: (((? X) . (? X)))
  Was EXTEND called correctly here?
```

Since `Extend` is the procedure that builds a bigger environment list, it should never be called in this test case. The user informs DEBUSSI of this fact as shown below.

```
5>No
  Enter violated condition:
6>(Not (Executed Extend))
```

Without any further input from the user, DEBUSSI localizes the bug to the call to `Eq` in `Var-Var-Match`.

```
  Dependency analysis exonerates: Extend.
  **Within VAR-VAR-MATCH,
  procedure call  (Eq '(? X) '(? X))
  returned incorrect value: Nil
  EQ is primitive; localization complete.
```

An experienced Lisp programmer would now immediately recognize the problem: Since variables are represented as lists, two variables need to be compared using `Equal`, not `Eq`. Notice that DEBUSSI required only six small pieces of information from the user to help find this bug.

Because of the way DEBUSSI is implemented on top of Cake, the localization algorithm demonstrated above works equally well for debugging designs (mixtures of incomplete code and partial specifications) as it does for debugging fully completed code.

After this introductory material, we now begin the layer-by-layer description of Cake.

III. Truth Maintenance

The bottommost layer of Cake is a simple, monotonic truth maintenance system [19]. Truth maintenance is the foundation of Cake in the sense that the reasoning procedures in all layers are required to record their dependencies in the single uniform network provided by this layer. Facilities are also provided so that further extensions to Cake can obey this discipline. This architecture is motivated by the evolutionary nature of software development and the need to support intelligent assistance.

The fundamental data abstraction in the truth maintenance layer is a *fact* (also called a *node*). Every fact is in one of two states: *in* (believed) or *out* (not believed).[2]

Examples of facts in the Requirements Apprentice are: "check in is the inverse of check out" (a user-provided fact—see Section II.A, line 9) and "check out takes an ISBN as input" (a deduced fact).

Dependencies form a network with the facts as nodes. Each dependency specifies a set of facts (the *supports*) that has been used to deduce some other fact (the *conclusion*). Each dependency may also include an arbitrary documentation string.

[2] This is a white lie. In order to support the propositional inference algorithm more efficiently, facts in Cake have three states, as described in Section IV.

The truth maintenance system ensures that all (and only) facts with well-founded support are *in*. A fact has well-founded support if it is either a *premise* (a distinguished class of facts whose belief state is set externally) or the conclusion of a dependency all of whose supports have well-founded support.

The truth maintenance system maintains this invariant incrementally whenever a premise is asserted or retracted (state set to *in* or *out*), or when a dependency is added (dependencies may not be removed). The algorithm visits each node in the network at most once (dependency cycles are prohibited). This is therefore an example of a kind of reasoning that is inexpensive enough to run spontaneously.

Another important service provided by the truth maintenance layer is the triggering of attached procedures (demons) whenever the state of a fact changes. This facility is used by other layers to implement various kinds of specialized processing, such as searching for an alternative proof when a fact goes *out*. It is also used in the Requirements Apprentice to monitor when certain important facts come *in* (such as when two definitions are proved equal—see Section II.A, line 11).

A. Using Dependencies

In addition to supporting the incremental processing described above, dependencies are useful for two other purposes.

First, dependencies have been an invaluable aid to us as developers of Cake for debugging it and the tools built on top of it. Because Cake is a highly data-directed system, the flow of control through its procedures is not obvious. When an undesirable conclusion is reached through a long chain of deductions, it can therefore be quite difficult to find the error. By convention, whenever a specialized reasoner (such as equality) records a dependency, it identifies itself in the associated documentation string. The procedure for printing support trees includes these documentation strings in its output, (e.g., "...`by Equality from` ..." in Section II.A, line 12). We have also developed a graphical tool for inspecting support trees.

Second, dependency information can be used to provide explanations to the end user of a software tool built on top of Cake. From our experience with Requirements Apprentice, however, the dependency network is best viewed as only the "raw material" for such explanations. This is partly because of the sheer size of the dependency tree supporting a typical fact. Also, most of the internal steps in these machine generated proofs refer to formal mathematical theories rather than the application domain.

B. Problems

Significant research issues remain regarding how to abstract and present dependency information in a form that is useful to an end user (for related research, see [20]).

Another major problem with the truth maintenance layer has been the lack of a methodology for "garbage collection." In the current implementation, nodes can only be added to the dependency network, never removed. It is clear that this approach will not scale up to large applications, since it leads to the accumulation of useless dependency network, which takes up memory and eventually slows down the system due to paging. (Useless network does not directly slow down the truth maintenance algorithm, since it is not looked at during normal processing.) The lack of garbage collection is especially problematical when a software tool explores many hypotheticals (such as when the Debugging Assistant considers alternative control paths through a program). Recent research on combining efficient context switching mechanisms with truth maintenance may be helpful in a future version of the this layer.

In the final analysis, the most dramatic architectural impact of the truth maintenance layer has been indirect. The necessity of recording correct dependencies caused us to re-implement a number of standard reasoning algorithms in Cake rather than being able to use implementations already written by others. Truth maintenance is not a feature that is easily added to existing code.

IV. Boolean Constraint Propagation

The second layer of Cake provides limited propositional reasoning. Propositional reasoning makes conclusions concerning boolean combinations of facts (i.e., using $\wedge$, $\vee$, and $\neg$), in which the individual facts are considered to be atomic objects. This form of deduction underlies all of the behavior presented in Section II.

Unfortunately, a complete decision procedure for propositional logic requires time in the worst case exponential in the number of propositions (facts), which would be unacceptable as spontaneous reasoning. Instead, Cake spontaneously performs only unit propositional resolutions, in which one of the inputs is a single fact and the other is a clause—a disjunction of facts. For example, from $\neg A$ and $A \vee \neg B \vee C$, Cake can conclude $\neg B \vee C$. This incomplete inference rule seems to correspond reasonably well with what users consider to be "obvious" conclusions when using a tool such as the Requirements Apprentice.

The fundamental data abstraction introduced by the boolean constraint propagation layer is the *clause*, represented as a list of (possibly negated) nodes. Clauses may be added to Cake, but not removed (and are therefore subject to the same garbage collection problem as nodes and dependencies).

The concrete data structures for the boolean constraint propagation layer and the truth maintenance layer have been merged to achieve improved efficiency. Nodes in Cake therefore have three states, called *truth values*: *true*, *false*, and *unknown*, corresponding respectively to a fact being believed, its negation being believed, or neither.

The constraint propagation algorithm implements unit resolution by repeatedly performing the following basic step: Suppose exactly one of the nodes in a clause is unknown, the others being either true or false. Furthermore, suppose that each true node is negated in the clause and each false node is not negated. The state of the unknown node may then be set to false (if the node is negated in

the clause) or true (if it is not negated). This algorithm requires time in the worst case linear in the number of clauses and requires no additional space for new clauses.

This basic algorithm was developed by McAllester [21] and can be thought of as a kind of constraint propagation network [22] (hence the name of the layer: boolean constraint propagation); it is also related to achieving arc consistency in boolean constraint satisfaction [23].

A. Contradictions

The propositional inference algorithm described above also detects "obvious" contradictions. A contradiction has been detected whenever all the nodes in a clause are set to the wrong truth value (i.e., all the negated nodes are true and all the non-negated nodes are false). Contradictions are handled in Cake through the Common Lisp condition system [24]. If the contradiction condition is not trapped by a higher-level control mechanism (as, for example, in reasoning with simplifying assumptions [25] or in refutation proofs as shown in the next section), the user is asked what to do:

```
>(Assert[3] P)               ;make premise
>(Assert (Implies P Q))

>(Assert (Not Q))
 **Contradiction**
 There is a conflict between the premises:
        1. P is True.
        2. (Implies P Q) is True.
        3. (Not Q) is True.
 1: Retract one of the premises
 2: Ignore this contradiction
 3: Return to Common Lisp Top Level

 Select option: >1
 Premise to retract: >3
 Retracting (Not Q) being True...
```

It is important to note that one of the options above is to ignore the contradiction. (The user can come back to deal with it later later). In many conventional theorem proving systems, it is not possible to continue making sound deductions when there is a contradiction present. In Cake, however, (due to keeping track of dependencies) reasoning that does not involve the contradiction can continue without difficulty. This is particularly important for tasks such as requirements acquisition where it is impractical to demand that the user resolve each problem as soon it is discovered.

B. Explicit Control

Cake can be requested to "try harder" to deduce that a given unknown node is *true* (or *false*) by temporarily setting the truth value of the node to *false* (or *true*). If the propagation of this temporary state leads to a contradiction, then the opposite truth value is deduced (this is called a refutation proof). In the limit, this approach can be used to achieve a complete (and therefore exponential) decision procedure for propositional logic by trying all combinations of values for all unknown nodes.

[3] We assume in this paper that procedures such as `Assert` appropriately quote their arguments. In fact, for building tools on top of Cake, these procedures will normally pass around internal node (and other) data structures.

This feature is made available to the implementor of a software tool on top of Cake through use of the `Ask` procedure illustrated below.

```
>(Assert (Implies R S))
>(Assert (Implies (Not R) S))
>(Truth S)           ;retrieve truth value
 :Unknown            ;constraint propagation is incomplete
>(Ask S)             ;try refutation if necessary
 :True
>(Why S)             ;show immediate support
 S is True by Refutation from:
   1. (Implies R S) Is True as a premise.
   2. (Implies (Not R) S) is True as a premise.
```

`Ask` is used throughout the Requirements Apprentice and the Debugging Assistant whenever the task context suggests it is worth trying harder for a particular conclusion.

V. Equality

Reasoning about equality is particularly important for the Programmer's Apprentice, because the formal semantics of the Plan Calculus (Section IX) makes heavy use of equality; for example, data flow arcs in plans denote equalities between terms representing the source and destination points. Equality reasoning is also at the heart of the main error-detection heuristic illustrated in the Requirements Apprentice session.

In the preceding two layers, a fact was an atomic object with no internal structure. The equality layer introduces the *term* data abstraction. A term is a hierachical expression of operators and arguments, down to the primitive terms at the leaves. Examples of terms are: `5`, `(< X (Plus Y 5))`, `Check-In`, `(Inverse Check-Out Check-In)`, and so on (terms are written here in the Lisp prefix notation used by Cake). Starting with this layer, every node in Cake has a corresponding term, but not vice versa. Only terms that are "boolean valued", i.e., that express facts, have corresponding nodes. Other terms, such as `5` and `(Plus Y 5)` are the building blocks of facts. (Cake's internal data structure for terms prints out in the form `#<Term ...>`.)

The equality layer also introduces the equality operator, `=`. Equality is an equivalence (transitive, reflexive, antisymmetric) relation on terms satisfying the congruence closure property: If term t_1 equals term t_2, then any term T containing t_1 is equal to the term obtained by substituting t_2 for t_1 in T.

An equality in Cake is a fact that may be asserted, retracted, and proved like any other. For example:

```
>(Assert (= Y Z))
>(Why (= (F Y) (F Z)))
 (= (F Y) (F Z)) is True by Equality from:
   1. (= Y Z) is True as a premise.
```

Using a number of specially designed internal data structures, the equality layer of Cake provides a complete ground reasoner for equality. What this means is that Cake is guaranteed to spontaneously assign *true* to an equality node (a node associated with a term of the form `(= ...)`) if the equality follows from other true equality

nodes in the system using (only) the semantics of equality specified above.[4]

Congruence closure of equivalence relations is a well studied problem (see, for example, [26]), for which quite efficient ($n \log n$ average time complexity) algorithms have been developed. The algorithm used in Cake is somewhat less than optimal due to some tradeoffs that have been made to facilitate incremental assertion and retraction of equalities and the recording of dependencies.

A. Hybrid Reasoning

The interaction between propositional reasoning (as implemented by boolean constraint propagation) and equality reasoning is a good example of some of the difficulties that arise in building a hybrid system like Cake.

When you combine a complete reasoner for theory $\mathcal{T}$ (e.g., equality) with an incomplete reasoner for theory $\mathcal{U}$ (e.g., propositional logic), you end up with an incomplete reasoner for the union theory (the set of all true facts statable using the symbols of $\mathcal{T}$ and/or $\mathcal{U}$). This can be illustrated simply in Cake by the following example:

```
>(Assert (= U V))
>(Assert (Or P (= V W)))
>(Assert (Or (Not P) (= V W)))
>(Truth (= (F U) (F W)))        ;desired conclusion
 :Unknown
>(Truth (= V W))                ;the missing link
 :Unknown
```

The problem here is that the incompleteness in propositional reasoning has “interrupted” the flow of equality reasoning. Unfortunately, asking the system to try harder (using `Ask`) to prove the desired conclusion above doesn’t help, because the equality layer doesn’t have any special algorithms for reasoning with *dis*equalities:

```
>(Ask (= (F U) (F W)))
 :Unknown
```

The only way to get the desired conclusion is to identify the missing link and try refutation on it:

```
>(Ask (= V W))
 :True
>(Truth (= (F U) (F W)))
 :True
```

Although the appropriate node on which to try harder is obvious in this contrived example, it can be difficult to identify in practice. This illustrates the general conclusion that even if the behaviors of two reasoners are relatively easy to understand individually, it can be hard to understand the net behavior when they are combined.

B. Pattern-Directed Invocation

Another important facility associated with the introduction of terms in the equality layer is pattern-directed invocation: a procedure (demon) is automatically executed whenever a term matching a given pattern is created. Pattern-directed invocation is used heavily to implement the algebra, types, and frames layers of Cake, as well as by tools built on top of Cake.

Pattern-directed invocation also interacts strongly with equality reasoning. In the presence of equality, the proper conditions for invoking a demon become more complicated. However, in this instance, we were able to come up with a very satisfactory hybrid solution.

The following is a simple example of the incompleteness problem that arises when you naively combine pattern-directed invocation with equality. Suppose there is a demon with the pattern `(G 0 *)`, i.e., it is waiting for the creation of a term with operator `G`, first argument `0`, and any second argument.[5] (This demon might have some special knowledge about the behavior of the function `G` when its first argument is zero.) Now suppose that the term `(G A B)` is created and then `A` is asserted equal to zero. The knowledge embodied in the demon above is now relevant, but unfortunately the demon is not invoked because the term `(G 0 B)` has not been created.

A brute-force approach to this problem would be to automatically close the set of terms in the system under substitution of equals (i.e., to create all terms such as `(G 0 B)`). This approach is not feasible, however, because the numbers of terms generated grows quickly with the number of equalities, and is infinite in the case of recursively defined equalities.

We have developed an algorithm [7] that solves this completeness problem by generating just the subset of all possible substitutions necessary to invoke all relevant demons. Furthermore, the algorithm is incremental. New demons, new terms, and new equalities can be added in any order. Equalities can also be retracted.

VI. Types

Types, type hierarchies, and type signatures are a common feature of modern software from requirements specifications down to programming languages. It was therefore natural to provide facilities in Cake for reasoning about types. A powerful way of doing this is to introduce types directly into the syntax of the logical language in which reasoning takes place.

A type[6] in Cake is a predicate on terms. The central algorithm of the types layer assigns a type to every term at the time it is created using a type inference algorithm similar to the one used in typed programming languages, such as ML [27]. (The type is computed recursively based on the type signature of each operator and the types of its arguments.) This type, called the “syntactic type” of the term, is a permanent and unchangeable property of the term. (However, as we will see below, it is also possible to assert and retract additional type information).

There are two advantages to using a typed as opposed to untyped reasoning language. The first is efficiency. Ax-

[4] Note that a complete ground reasoner for theory $\mathcal{T}$ is not the same as a complete decision procedure for the quantifier-free theory of $\mathcal{T}$, since the quantifier-free theory of $\mathcal{T}$ includes propositional logic as a sub-theory. See the discussion of hybrid reasoning below.

[5] Cake also supports pattern variables and nested patterns.

[6] In logic, what we have called a “type” is more properly called a “sort.” The language used by Cake is therefore a “many sorted logic.” In this paper, however, we will continue to use the more software-oriented terminology of “type.”

iomatizations and proofs in a typed language are typically smaller than the corresponding untyped axiomatizations and proofs. Furthermore, in a typed system, much of the taxonomic reasoning in a given problem can be achieved by specialized reasoning mechanisms operating on the type structure, rather than by general purpose deduction.

A second advantage of using a typed language is more incidental. It has generally been the experience of reasoning system builders (confirmed by our work with Cake) that including taxonomic information in the syntax of the language helps to catch errors and manage complexity in the formalization task.

A. A Typed Logic

In Cake we have implemented a very expressive form of typed logic, similar to LLAMA [28]. Its main features (each discussed briefly below) are:

- boolean lattice of types
- polymorphic type signatures
- overlapping argument types
- retractable type assertions

A boolean lattice of types means that in addition to specifying the usual subtype hierarchy, (e.g., `Integer` is a subtype of `Number`, which is a subtype of `Any`), we can also use intersection, union, and complement operators (`*And`, `*Or`, and `*Not`) to form new types. For example, `(*Or List Null)` is the type of an implementation object that is either a list or nil. `(*And Male Student)` might be the type of a requirements object.

The type lattice in Cake is non-retractable. New subtypes may be added dynamically (i.e., extending the lattice "downward"), but the existing subtype relationships may not be changed.

Complete type inference on a boolean lattice is exponential in the worst case. Because type inference in Cake is a spontaneous reasoning process, we intentionally implemented an incomplete version of type inference, which may fail to compute the most specific type when complements are involved. This has not turned out to be a problem in practice.

Polymorphic type signatures are a kind of operator "overloading," in which each operator is assigned a *set* of functionalities. This allows quite sophisticated reasoning to be performed entirely within the type inference algorithm. For example, assuming `Even` and `Odd` are subtypes of `Number`, the type signature for the function `Plus` might include, among others, the functionalities shown below.

```
(Defunction Plus (Number Number Number)
  (:From Odd Odd Even)       ;odd × odd → even
  (:From Even Odd Odd))      ;even × odd → odd
```

Using this information, Cake can, through type inference alone, deduce the parity of complex expressions, such as the following (assuming that the syntactic type of `N1` through `N5` is `Odd`):

```
>(Type-Of (Plus (Plus N1 N2)
                (Plus (Plus N3 N4) N5)))
 Odd
```

In Cake's type inference algorithm, we compute a result type for an expression as long as the type of each argument *overlaps* (i.e., has a non-empty intersection) with the type of the corresponding position in the operator's signature. (In more restrictive logics, the type of the argument is required to be a subtype of the argument position.) Several examples of the utility of this feature are given in [28].

B. Retractable Type Assertions

In addition to the fixed syntactic type of a term, further facts about a term's type may be asserted, retracted, and proved like any others. This has certain advantages and costs.

The main advantage of supporting retractable type assertions is that it allows us to derive type information from non-type information and vice versa using general-purpose deduction. For example, the type `Even` can be defined to be subtype of (in Cake, *specialize*) `Number` as shown below.

```
(Deftype Even (:Specializes Number)
  (:Constraints (= (Remainder ?Even 2) 0)))
```

Given this definition, one can prove that a number `M` is less than zero depending on the fact that its type is `Even`:

```
>(Assert (Even M))
>(Expand-Defn (Even M))          ;discussed below
>(Why (= (Remainder M 2) 0))
 (= (Remainder M 2) 0) is True
 by Definition of Operator from:
   1. (Even M)
```

Conversely, one can prove that a number `N` is of type `Even` by establishing that it is less than zero:

```
>(Assert (= (Remainder N 2) 0))
>(Show (Even N))    ;expand definition if necessary
 :True
```

Retractable type assertions are very useful for exploring requirements or design options. However, this flexibility comes at significant cost. For instance, Cake can conclude that the compound `Plus` term above is of type `Odd` based on retractable type assertions for `N1` through `N5` (i.e., assuming now that their syntactic type is just `Number`). However, performing this deduction with retractable type assertions instead of at term creation time would require roughly an order of magnitude more computational resources (for the extra data structures and processing associated with the extra terms created and the appropriate incremental reasoning).

Given that there is a choice between retractable and non-retractable type reasoning, what we have typically done in using Cake is to set some basic syntactic type information at term creation time, and then to use retractable type assertions to explore options that are natural to the task. For example, at the requirements level, basic distinctions such as whether a term refers to an object, a transaction, or a report would be syntactically fixed. More particular properties, such as whether a particular report is weekly or monthly, would be explored incrementally.

If an error is made in the basic syntactic types, one must re-initialize the system and start again. Typically, there

is a text file of definitions (such as the definition of `Even` above), which one can edit and then re-evaluate.

The type assertion example above also illustrates another kind of reasoning in Cake (in addition to refutation) that needs to be explicitly controlled, namely definition expansion. Cake does not automatically expand all definitions, because this can lead to an explosion of terms (an infinite explosion in the case of recursive definitions). The procedure `Expand-Defn`, given a term, expands the definition of the operator of that term (with the appropriate if-and-only-if logical connection between the term and its expansion), unless it has already been expanded. In the frame layer (Section VIII.B), a heuristic is introduced that automatically calls `Expand-Defn` under certain conditions.

The `Show` procedure illustrated above tries even harder than `Ask` to prove a given node true or false by (in addition to trying refutation) expanding the definition of operators appearing in the corresponding term. `Show` also invokes special backward-chaining proof procedures that can be associated with particular operators.

VII. Algebra

Simple algebraic properties, such as associativity and commutativity, appear everywhere in the formal modeling of data structures and application domains. We have therefore included special-purpose decision procedures in Cake for various of these common algebraic theories. The algebra layer also contains limited inference procedures for sets, partial orders, and total orders (these are built on top of the mechanisms already used in the types layer).

Any operator symbol in Cake can be given (either as its syntactic type or as a retractable assertion) one or more of the following simple algebraic properties: transitivity, associativity, commutativity, reflexivity, symmetry, anti-symmetry, involution, idempotency. Efficient (worst case linear or $n \log n$) complete ground reasoners have been implemented for each of these theories using a variety of special-purpose data structures.

The algebra layer also introduces special mechanisms associated with the type `Undefined` to support reasoning about partial functions and strictness[7] (see [6] for further details).

A. Commutativity

To illustrate the tradeoff between retraction and efficiency that occurs throughout the algebra layer, let us consider commutativity. The commutativity of a function is a fact that can be asserted, retracted, and proved like any other. For example:

```
>(Assert (Commutative G))
>(Ask (= (G A B) (G B A)))
 :True
>(Retract (Commutative G))        ;retract premise
>(Ask (= (G A B) (G B A)))
 :Unknown
```

[7] A function is strict if and only if its value is undefined whenever its argument is undefined.

However, for many algebraic properties—commutativity being one—it is possible to implement some very efficient special-purpose technique if the property is non-retractable. For example, since the users of Cake are not likely to want to explore the possibility of `Plus` not being commutative, `Commutative` is part of syntactic type of the function `Plus`.

Setting the syntactic type of a function to include `Commutative` installs a special procedure that runs during the *interning* of expressions with that function in the operator position. Interning in Cake, by analogy with interning in Lisp, is the process by which an external Lisp expression is mapped to an internal Cake term data structure such that any two `Equal` (i.e., printing the same) expressions map to the identical term. This can be thought of as a generalization of symbol tables from symbols to the uniqueization of tree structures.

The special interning procedure for non-retractably commutative functions sorts its arguments according to a fixed (e.g., lexicographic) term order. The effect of this procedure can be seen below:

```
>(Intern '(Plus A B))
 #<Term (Plus A B)>
>(Intern '(Plus B A))
 #<Term (Plus A B)>
```

Thus the distinction between `(Plus A B)` and `(Plus B A)` is permanently lost. Declaring a function to be non-retractably commutative prevents the creation of many terms plus all of the extra structures and processing to make terms with different arguments equal under the correct conditions.

B. Language Extensions

Special interning procedures are also used in Cake as a way of extending the reasoning language. The interning procedure associated with an operator has the power interpret the arguments to an expression in a non-standard fashion, if desired. For example, one can define a special interning procedure for the operator `!Infix` that parses its string argument as an expression in standard mathematical notation. (By convention, operators in Cake that take special argument syntax begin with an exclamation point.) Thus the expression `(!Infix "X+Y=Z")` would be interned as `#<Term (= (Plus X Y) Z)>`.

The special interning facility plays the same role in Cake as a macro facility in a programming language, i.e., it makes it possible to embed an arbitrary sublanguage. This has turned out to be extremely useful both for extending Cake in various ad hoc ways and also for building systems on top on Cake.

For example, although Cake is basically a propositional reasoner, we found it useful to add some limited capabilities for reasoning with expressions having bound variables, such as lambda expressions and quantifications.

Most of the reasoning procedures in the algebra layer can be viewed as special-purpose implementations of specific quantified statements. For example, asserting the commutativity of `G` above is equivalent to asserting the

statement $\forall xy\ \texttt{G}(x,y) = \texttt{G}(y,x)$. However, it is also sometimes useful to be able to assert and retract arbitrary quantified statements. For this purpose we introduced the special operators `!Forall` and `!Exists`. The special interning procedures associated with these operators parse the syntax of quantified expressions to extract the (typed) bound variables and build the appropriate data structures to support instantiation and various proof procedures using quantifiers.

For example, in the following interaction with Cake, the quantified fact $\forall n : number\ \texttt{F}(n{+}1) > \texttt{F}(n)$ is asserted and then used to prove a proposition by universal instantiation on 5.

```
>(Assert (!Forall (?N Number)
                  (> (F (Plus ?N 1)) (F ?N))))
>(Instantiate-Universal ... 5)
>(Why (> (F 6⁸) (F 5)))
 (> (F 6) (F 5)) is True
 by Universal Instantiation from:
   1. (!Forall (?N Number)
               (> (F (Plus ?N 1)) (F ?N)))
      is True as a premise.
```

Similar language extensions were made during the construction of the Requirements Apprentice, such as the "`(!The ... Such-That ...)`" construction used in line 12, premise 4 (Section II.A).

The use of `Instantiate-Universal` above is another example of where explicit control of reasoning is required in Cake. Reasoning with quantifiers is inherently explosive. We cannot afford to spontaneously instantiate all quantifications in Cake on all applicable terms (applicable terms are those whose syntactic type overlaps with the syntactic type of the bound variable). Instead, it is the responsibility of the tool developer to explicitly instantiate quantifications on individual terms that are of interest.

In building the demonstration of the Requirements Apprentice, there were adequate heuristics within the task at hand to choose appropriate individuals on which to instantiate quantifications. It was not necessary to make control over quantification instantiation visible to the end user.

In summary, the algebraic layer, together with types and equality provide a specification language that, although far from complete, has made it possible to state and reason about a wide range of important properties of software.

VIII. Frames

Frame representation has evolved (and some might say mutated) greatly since Minsky's introduction of the term "frame" to describe a prototypical concept in commensense reasoning [30]. In current usage, a frame is basically a convenient data structure, similar to a Lisp property list, that is used to represent taxonomically organized, structured objects. Since software development is replete with such objects, the next layer of Cake introduces frames as a data abstraction.

[8] A special interning procedure on `Plus` makes `(Plus 5 1)` intern as `#<Term 6>`, and similarly for other arithmetic functions. This use of special interning procedures was introduced in FOL [29] as "semantic attachment."

The major difference between frames in Cake and frames in current expert system shells is that in Cake in addition to inserting and removing values from slots, it is possible to reason abstractly about the properties of frames and slots. This gives Cake a great deal of additional power (with of course, additional cost). This power is used, for example, in the Debugging Assistant to reason about the behavior of a program given a class of input data, rather than just specific input values.

A. Implementing Frames as Types

A frame in Cake is a type. Accessors for the slots (roles) of a frame are functions with the frame type as their domain. As we will see below, this implementation of frames takes maximum advantage of reasoning facilities already implemented in the algebra and types layers, such as specialization, operator type signatures, and strictness (slot functions are strict).

Multiple-valued slots are represented in Cake by making the range of the accessor function be a set. In our experience, single-valued slots are much more common than multiple-valued ones. Making slot accessors be functions rather than binary relations leads to much more efficient reasoning for single-valued slots at a small increase in complexity for multiple-valued ones.[9]

The frame definition example below defines the type predicate `Pair` and two functions, `Left` and `Right`, with domain `Pair`.

```
(Deframe Pair (:Parts Left Right))
```

In Cake unlike conventional frame systems, a distinction is made between the set of slots that uniquely determine a frame (called the *parts* in Cake) and other slots (called *properties*). The parts of a frame obey an extensionality axiom: Two instances are equal if all of their parts are equal. We have found this notion of extensionality to be very important for reasoning about mathematical objects, such as data structures (and in the next section, plans), but less useful for modelling real-world domains.

One way to specialize a frame (create a subtype) is to restrict the types of the slots. For example, the following defines a subtype of `Pair` in which both parts are numbers.

```
(Deframe Pair-Of-Numbers
  (:Specializes Pair)
  (:Parts (Left Number) (Right Number)))
```

Among other things, this definition adds the functionality $\textit{pair-of-numbers} \rightarrow \textit{number}$ to the signature of the functions `Left` and `Right` (taking advantage of the overloading feature described in Section VI.A).

The `Pair-Of-Numbers` frame can be further specialized by adding the constraint that the parts be sorted from left to right, as shown below.

```
(Deframe Sorted-Pair-Of-Numbers
  (:Specializes Pair-Of-Numbers)
  (:Constraints (< ?Left ?Right)))
```

[9] For example, if slots are functions, the semantics of path expressions involving only single-valued slots does not require an existential quantifier.

This defines a new type in essentially the same manner that `Even` is defined in Section VI.B.

B. Symbolic Frame Reasoning

Given the way that frames and slots are implemented as types and functions, symbolic reasoning about slots is provided without the need for additional mechanism. To illustrate, suppose we create the new term `My-Pair` to be an instance of (i.e., having syntactic type) `Sorted-Pair-Of-Numbers` as shown below. (Cake procedures that operate on frames start with `F`.)

```
>(FInstantiate Sorted-Pair-Of-Numbers My-Pair)
 #<Term My-Pair>
```

Now, for example, if we assert that the value of the `Right` slot of `My-Pair` is less than zero, Cake can conclude that the value of the `Left` slot must also be less than zero.

```
>(Assert (< (Right My-Pair) 0))
>(Ask (< (Left My-Pair) 0))
 :True
```

Symbolic reasoning of this kind is a powerful extension to conventional frame systems, in which reasoning can only take place when a slot is assigned a specific value.

This example also introduces a useful heuristic for automatically expanding type definitions. In order to make the conclusion above, the definition of `Sorted-Pair-Of-Numbers` needs to be expanded (to get the constraints), just as the definition of `Even` needed to be expanded in Section VI.B. It is hard to know in general which occurrences of a defined operator are useful to expand. However, frame type definitions have more structure, which supports the following heuristic: If a term representing one of the parts or properties of a frame instance is created during the reasoning process (e.g., `(Right My-Pair)`), it is likely to be useful to expand the corresponding frame type assertion (i.e., `(Sorted-Pair-Of-Numbers My-Pair)`).

This heuristic is implemented using the pattern-directed invocation facility described in Section V.B: Defining a frame installs demons that are triggered by the creation of any term whose operator is one of the parts or properties of the frame. When the demon executes, it creates the corresponding frame type assertion (if necessary) and calls `Expand-Defn` on it.

C. Slot Values and Literals

In Cake assigning a value to a slot is logically equivalent to asserting an equality between the term representing the slot and the term representing the value. Thus, the following procedure call, which sets the value of the `Left` slot of `My-Pair` to `-5`,

```
>(FPut (Left My-Pair) -5)
```

is logically equivalent to

```
>(Assert (= (Left My-Pair) -5)) .
```

To retrieve stored values, however, Cake must have some way of distinguishing the value of a slot from other terms that may be equal to the slot term. (For example, if `(Left My-Pair)` was equal to `(Left Your-Pair)`, you would not want to retrieve either term as a slot value.) This problem is solved by introducing a syntactically distinguished class of terms called *literals*. The key logical property of literals is that they are mutually disequal, i.e., every literal is disequal to every other literal. Numbers and strings are provided in Cake as literals (a set of literals is also a literal). Developers can add their own kinds of literals (e.g., the colors red, green, blue, etc.).

The procedure for retrieving slot values, `FGet`, searches through the set of terms equal to the given slot term (there is a data structure in the equality layer of Cake that makes this search efficient). If it finds a literal (there can be at most one if there is no contradiction), it returns the literal term as the value; otherwise `Nil` is returned to indicate no value (contradictory values are considered to be no value by this procedure). When `FGet` finds a slot value, it also returns the truth maintenance node corresponding to the equality between the slot term and the literal, as shown below.

```
>(FGet (Left My-Pair))
 -5
 #<Node (= (Left My-Pair) -5)>
```

D. Dependency-Directed Frame Programming

A tool developer may simply ignore the node returned by `FGet`, and use the frame layer of Cake to store and retrieve values much like a conventional frame system. (`FGet` uses the multiple return value feature of Lisp. The second return value is ignored unless it is captured with a special form.) However, the integration of the frame layer with truth maintenance provides the option of a more disciplined, dependency-directed methodology.

To illustrate, suppose that a tool retrieves a value (using `FGet`) from some slot (e.g., `(Left My-Pair)` above) and then, based on the results of a test applied to that value, stores some value (using `FPut`) in another slot. In Cake, the stored value can be made to depend on the retrieved value by providing the node returned by `FGet` as an (optional) argument to `FPut`. If this is done, then when the value of the retrieved slot is later retracted, the stored value will be automatically retracted by the truth maintenance system. (The tool can also arrange for other arbitary processing to occur at this time by installing a demon on the node returned by `FGet` that fires when its truth value changes.)

Although we found (for example, in the Requirements Apprentice) that using this style of dependency-directed programming with frames led to a conceptually cleaner handling of incremental change, it unfortunately also led to a lot of syntactic clutter in the code due to the introduction of temporary variables and special forms to capture and accumulate the nodes returned by `FGet` and then provide them as arguments to `FPut`. Cake therefore provides a special scoping macro, `With-Dependencies`, that simplifies the code written in the common case in which all `FPut`'s depend on all the (dynamically) preceding `FGet`'s. Using this macro, dependency-directed frame code can be written in the simple form below.

```
(With-Dependencies
  (Setq X (FGet ...))
  ...
  (If ...X... (FPut ...) (FPut ...)))
```

A notable omission from Cake's frame layer is automatic classification: many frame systems automatically place any newly defined (or updated) frame at the appropriate location in the taxonomic hierarchy. The usefulness of this service for software development is well demonstrated, for example, in the LASSIE system [31]. A special-purpose form of classification was implemented as part of the Requirements Apprentice. In the future, it would make sense to add classification as a generic service in the frame layer of Cake.

IX. Plan Calculus

The first six layers of Cake provide knowledge representation and automated reasoning facilities which, although motivated in choice and implementation details by the software development application, are in fact quite generic. This section describes a layer of Cake specifically targetted toward a software development need: representing and reasoning about the algorithmic structure of programs. More important than the details of the Plan Calculus itself, this section illustrates how the generic facilities of the six layers can be brought together to implement a new specialized formalism.

The Plan Calculus is central to the operation of the Debugging Assistant and the Design Apprentice (it is not used in the Requirements Apprentice). For example, the first step the Debugging Assistant takes when it begins to help a programmer (see the architecture in Fig. 3) is to make an internal representation shift from source code (e.g., Lisp) to the Plan Calculus. (The Plan Calculus representation of the first procedure in the Debugging Assistant example is shown in Fig. 5.) As a result of this representation shift, the dependency information needed by the localization algorithm falls out as a by-product of straightforward test case evaluation.

A key goal of the Plan Calculus is to represent algorithmic structure in an abstract and programming-language independent fashion. A *plan* is therefore a hierarchical flowchart-like graph composed of test and input/output specification boxes connected by data flow and control flow arcs. Data flow is indicated in plan diagrams by plain arrows; control flow is indicated by cross-hatched arrows. Data flow arcs serve to abstract away from the details of programming language mechanisms for achieving the flow of data from producer to consumer, such as nesting of expressions, and the use of intermediate variables. Control flow arcs abstract away from the variety of control flow mechanisms in programming languages, such as goto statements, conditionals, and looping constructs. For a complete description of the Plan Calculus, see [2].

A. History

The first implementation of the Plan Calculus supported a demonstration of knowledge-based program editing, called KBEMACS [32]. This implementation was essentially a direct translation of the diagrammatic notation illustrated in Fig. 5 into convenient internal data structures for manipulating nodes and arcs in a directed graph. All of the "reasoning" in KBEMACS was achieved by special-purpose procedures that operated on these internal data structures.

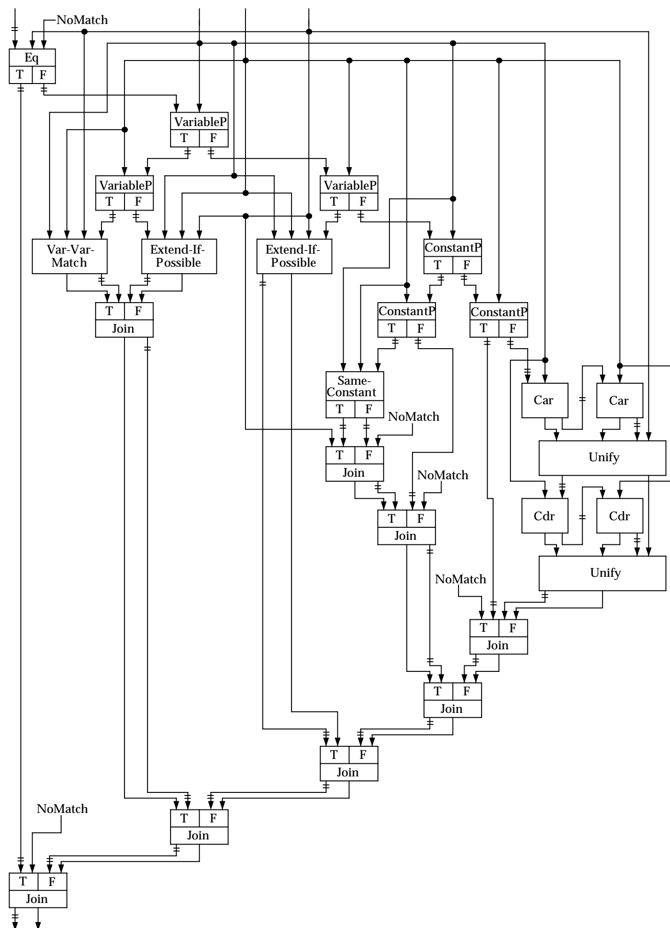

Fig. 5. Plan for the **Unify** program in Fig. 4.

When we considered extending KBEMACS to add error detection and more automation of design decisions, we realized that the ad hoc procedural approach had reached its limits—each new feature required too much new code to be written. Since a formal semantics for the Plan Calculus [33] had just been completed, we decided to try the approach of (re-)implementing the Plan Calculus by "macroexpanding" a plan diagram into its logically equivalent facts and then using automated reasoning (i.e., Cake) on those facts.

We feel this experiment has been very successful. The Plan Calculus layer is actually a very thin veneer on the top of the rest of Cake (about six percent of the code). This is much less code than was required to implement plans in KBEMACS with less functionality.

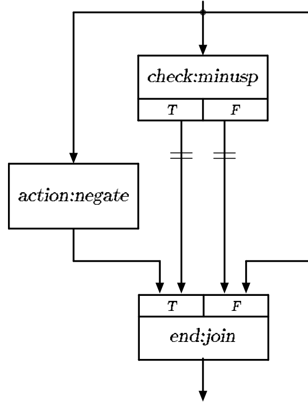

Fig. 6(a). Plan diagram for **Negate-If-Negative**.

```
(Deftest Minusp
  (:Inputs (Input Number))
  (:Condition (< ?Input 0)))

(Defio Negate
  (:Inputs (Input Number))
  (:Outputs (Output Number))
  (:Postconditions (= ?Output (- 0 ?Input))))

(Defplan Negate-If-Negative
  (:Roles (Check Minusp)
          (Action (*Or Negate Undefined))
          (End Join))
  (:Dflow ((Input ?Check) (Input ?Action))
          ((Input ?Check)  (Fail ?End))
          ((Output ?Action) (Succeed ?End)))
  (:Cflow ((Succeed ?Check) (Succeed ?End))
          ((Fail ?Check) (Fail ?End))))
```

Fig. 6(b). Cake definition of **Negate-If-Negative**.

```
>(Expand-Defn (Minusp (Check My-Plan)))
 (And ;slot type restrictions
  (Number (Input (Check My-Plan)))
  ;implicit situation slots
  (Situation (In (Check My-Plan)))
  ((*Or Situation Undefined) (Succeed (Check My-Plan)))
  ((*Or Situation Undefined) (Fail (Check My-Plan)))
  (Implies (Defined (Succeed (Check My-Plan)))
           (Precedes (In (Check My-Plan))
                     (Succeed (Check My-Plan))))
  (Implies (Defined (Fail (Check My-Plan)))
           (Precedes (In (Check My-Plan))
                     (Fail (Check My-Plan))))
  ;test condition
  (Iff (< (Input (Check My-Plan)) 0)
       (Defined (Succeed (Check My-Plan))))
  (Iff (Not (< (Input (Check My-Plan)) 0))
       (Defined (Fail (Check My-Plan)))))

>(Expand-Defn (Negate (Action My-Plan)))
 (And ;slot type restrictions
  (Number (Input (Action My-Plan)))
  (Number (Output (Action My-Plan)))
  ;implicit situation slots
  (Situation (In (Action My-Plan)))
  (Situation (Out (Action My-Plan)))
  (Precedes (In (Action My-Plan))(Out (Action My-Plan)))
  ;postconditions
  (= (Output (Action My-Plan))
     (- 0 (Input (Action My-Plan)))))

>(Expand-Defn (Join (End My-Plan)))
 (And ((*Or Situation Undefined) (Succeed (End My-Plan)))
  ((*Or Situation Undefined) (Fail (End My-Plan)))
  (Data (Output (End My-Plan)))
  (Or (Defined (Succeed (End My-Plan)))
      (Defined (Fail (End My-Plan)))))

>(Expand-Defn (Negate-If-Negative My-Plan))
 (And ;slot type restrictions
  (Minusp (Check My-Plan))
  ((*Or Negate Undefined) (Action My-Plan))
  (Join (End My-Plan))
  ;data flow constraints
  (Implies (Defined (Input (Action My-Plan)))
           (= (Input (Check My-Plan))
              (Input (Action My-Plan))))
  (Implies (Defined (Fail (End My-Plan)))
           (= (Input (Check My-Plan))
              (Output (End My-Plan))))
  (Implies (Defined (Succeed (End My-Plan)))
           (And (= (Output (Action My-Plan))
                   (Output (End My-Plan)))
                (Precedes (Out (Action My-Plan))
                          (Succeed (End My-Plan)))))
  ;control flow constraints
  (Implies (Defined (Succeed (End My-Plan)))
           (And (Defined (Succeed (Check My-Plan)))
                (Precedes (Succeed (Check My-Plan))
                          (Succeed (End My-Plan)))))
  (Implies (Defined (Fail (End My-Plan)))
           (And (Defined (Fail (Check My-Plan)))
                (Precedes (Fail (Check My-Plan))
                          (Fail (End My-Plan))))))
```

Fig. 6(c). Terms created for instance **My-Plan** of **Negate-If-Negative**.

```
(= (Output (End My-Plan)) 3) is True by Equality from:
1. (= (Input (Check My-Plan)) 3) is True as a premise.
2. (= (Input (Check My-Plan)) (Output (End My-Plan))) is True by Implication from:
   1. (Implies (Defined (Fail (End My-Plan)))
               (= (Input (Check My-Plan)) (Output (End My-Plan)))) is True by Conjunction from:
      1. (And (Minusp (Check My-Plan)) ...) is True by Definition of Operator from:
         1. (Negate-If-Negative My-Plan) is True by Syntactic Type.
   2. (Defined (Fail (End My-Plan))) is True by Disjunction from:
      1. (Or (Defined (Succeed (End My-Plan))) (Defined (Fail (End My-Plan))))
         is True by Conjunction from:
         1. (And ((*Or Situation Undefined) (Succeed (End My-Plan))) ...)
            is True by Definition of Operator from:
            1. (Join (End My-Plan)) is True by Conjunction from 2.1.1.
      2. (Defined (Succeed (End My-Plan))) is False by Implication from:
         1. (Implies (Defined (Succeed (End My-Plan)))
                     (And (Defined (Succeed (Check My-Plan)))
                          (Precedes (Succeed (Check My-Plan))(Succeed (End My-Plan)))))
            is True by Conjunction from 2.1.1.
         2. (And (Defined (Succeed (Check My-Plan)))
                 (Precedes (Succeed (Check My-Plan))(Succeed (End My-Plan))))
            is False by Conjunction from:
            1. (Defined (Succeed (Check My-Plan))) is False by Biconditional from:
               1. (Iff (< (Input (Check My-Plan)) 0)
                       (Defined (Succeed (Check My-Plan)))) is True by Conjunction from:
                  1. (And (Situation (In (Check My-Plan))) ...) is True by Definition of Operator from:
                     1. (Minusp (Check My-Plan)) is True by Conjunction from 2.1.1.
               2. (< (Input (Check My-Plan)) 0) is False by Equality from 1.
```

Fig. 6(d). Complete support tree showing how output of **Negate-If-Negative** depends on its roles and constraints when input is 3.

B. Implementing Plans as Frames

Rather than illustrating the implementation of the Plan Calculus with the plan in Fig. 5, we will instead use the small pedagogical example in Fig. 6, which will allow us to exhibit the macroexpansion in full.

Fig. 6(a) shows the diagram for `Negate-If-Negative`, a plan for computing the absolute value of a number: If the number is less than zero, then negate it; otherwise do nothing. `Negate-If-Negative` includes one of each of the three kinds of boxes in plan diagrams: a test specification (`Minusp`), an input-output specification (`Negate`), and a join. Each box in a plan has a unique name (indicated preceding the colon) so that boxes of the same type may be distinguished. (In Fig. 5, only the type of each box was indicated.)

Fig. 6(b) shows the notation used in Cake to define `Negate-If-Negative` and its two component specifications (`Join` is a special builtin type). Both plans and specifications are implemented as frames in Cake; data flow, control flow, preconditions, postconditions, and test conditions are translated into constraints on those frames.

Rather than going into the syntactic details of the notation in Fig. 6(b), let us create an instance of `Negate-If-Negative`, called `My-Plan`, and then examine the semantic content of the terms created when the frame definitions are expanded, paying particular attention how facilities in the six layers of Cake come into play.

```
>(FInstantiate Negate-If-Negative My-Plan)
```

Starting at the top of Fig. 6(c), we see that a test specification frame, such as `Minusp`, has in addition to slots for its inputs and outputs three implicit situation-valued slots called `In`, `Succeed`, and `Fail`.

The type `Situation` is introduced in the Plan Calculus layer to represent the following three phenomena:

- *Temporal Order:* Constraints on the temporal order of program steps are specified via the `Precedes` relation, which is a partial order [algebra layer] on situations.
- *Conditional Behavior:* `Situation` is a subtype of `Defined` and therefore disjoint [types layer] with `Undefined`. Whether a test succeeds or fails is represented by which of the `Succeed` or `Fail` slots is defined. Constraints guarantee that exactly one of the `Succeed` or `Fail` slots is defined and that the `In` situation of the test precedes it.
- *Side Effects:* A mutable object is represented as a strict function [algebra layer] from situations to data values.

Returning to Fig. 6(c), we see that an input-output specifications, such as `Negate`, has two implicit situation-valued slots, called `In` and `Out`, neither of which is conditional. The pre- and postconditions of an input-output specification translate directly into the obvious constraints on the frame.

Joins in the Plan Calculus are best thought of as part of the description of data and control flow—a join box does not represent any computation. A join has two situation-valued slots, `Succeed` and `Fail`, and one data-valued slot, called `Output`. The constraints on these slots fit with the data and control flow constraints of the larger plan (see below).

`Negate-If-Negative` has three slots, one for each of its component boxes. Notice that since the `Action` slot is conditional, its type is a disjunction with `Undefined`.

C. Data and Control Flow

The basic idea of data flow in the Plan Calculus is to specify equality [equality layer] between slots (and slot paths) in a plan. The most general form of a data flow constraint *from* some slot *to* some slot is:

```
(Implies (Defined to)
         (And (= from to)
              (Precedes from-situation to-situation)))
```

This constraint says that if the data is needed at the *to* location (typically an input), then it is equal to the data at the *from* location (typically an output). Furthermore, to preserve causality, the situation in which the data is produced must precede the situation in which it used.

The three data flow constraints in Fig. 6(c) are degenerate versions of the general form above. The first constraint ties together two inputs and therefore does not include `Precedes`. The second constraint is also from an input and involves a join, whose slots are specially handled. The third data flow constraint is from an output to a join.

Control flow constraints are similar in general form to data flow constraints. The difference is that, instead of specifying an equality, control flow requires that the *from-situation* be defined.

As an exercise, the reader may verify that, using propositional reasoning [boolean constraint propagation layer], equality, and the strictness of slot functions [algebra layer], the data flow and control flow constraints in Fig. 6 guarantee that the `Action` is defined (i.e., executed) whenever the `Check` succeeds.[10]

D. Reasoning with Plans

Given an instance of `Negate-If-Negative`, boolean constraint propagation alone is adequate to spontaneously compute the final output value for a given input, as shown below. In more complex plans, however, some explicit control of reasoning is usually necessary to execute a test case.

```
>(FPut (Input (Check My-Plan)) -3)
>(FGet (Output (End My-Plan)))
 3
```

Of course, this is the hard way of computing absolute value! The important thing about implementing plans as above, from the standpoint of the Debugging Assistant, is that all the correct logical dependencies [truth maintenance layer] are established. To illustrate, consider instead putting the value `3` at the input:

[10] Notice that the `Action` is not prevented from executing *more* often than needed, nor is it prevented from preceding the `Check`. This is a choice made by the person defining this plan in order to make it more general.

```
>(FRemove (Input (Check My-Plan)))
>(FPut (Input (Check My-Plan)) 3)
>(Why* (= (Output (End My-Plan)) 3))
 ;show full support tree—see Fig. 6(d)
```

Fig. 6(d) shows the complete dependency tree supporting the output value in this test case.[11] Notice that the output in this case depends on the specifications of the `Check` and `End` slots (underlined), but does *not* depend on the specifications of the `Action` slot. Thus, to obtain the initial suspects for a failing test case, the Debugging Assistant simply retrieves the premises underlying the output fact.[12]

Notice also that the documentation strings appearing in the support tree record the participation of the equality, boolean constraint propagation, and types layers.

Since plans are implemented as frames, the symbolic reasoning illustrated in Section VIII.B also works for plans. For example, removing the input value `3`, let us just assume below that the input to `Negate-If-Negative` is positive. From this assumption Cake can conclude that the output is equal to the input:

```
>(FRemove (Input (Check My-Plan)))
>(Assert (Not (< (Input (Check My-Plan)) 0)))
>(Ask (= (Output (End My-Plan))
         (Input (Check My-Plan))))
 :True
```

This kind of reasoning about the abstract properties of a plan (program) can be very useful in software design. For example, the Debugging Assistant [16] has demonstrated debugging a partially implemented program, i.e., one in which there is a mixture of code and unimplemented specifications. (The sooner one can find problems, the better.)

X. Conclusion

We have described our experience in building and using a knowledge representation and reasoning system, called Cake, to support two experimental software development tools. Our overall conclusion is that the facilities demonstrated in Cake are both feasible and useful to support knowledge-intensive, evolutionary, intelligent assistants for software development.

Three key architectural issues in this work are: control of reasoning, hybrid representation and reasoning, and the tradeoff between retraction and efficiency.

Our approach to the control of reasoning was to identify a portion of the reasoning which was efficient and to let that portion execute spontaneously (without explicit control). The remaining reasoning was subject to explicit control by the tool developer or end user.

Our approach to hybrid representation and reasoning was to divide Cake into separately implemented layers and to analyze their interactions on a case-by-case basis. This was only moderately successful in controlling the complexity of a highly hybrid system.

Our approach to the tradeoff between retraction and efficiency was to provide the tool developer with the choice between the two. This required a lot of implementation effort, but made the performance of the system acceptable (at least for demonstration purposes).

We also came to appreciate through the building of Cake both the power and the curse of a highly data-directed architecture. (In Cake, most things happen not because of one procedure calling another, but because of the appearance of certain terms in the database.) We believe that this approach has allowed us to reap a lot of behavior relative to the amount of code. However, we also had to learn with some pain how to develop various kinds of clever tracing tools to cope with the difficulty of debugging in such an environment.

At the time we began implemented Cake, there was no existing research or commercial system that came anywhere near providing all the capabilities we foresaw needing for the Programmer's Apprentice. Today, the system that is most similar in capabilities to Cake is RHET [34], which is also a research tool. Commercial knowledge representation and reasoning systems are still far from providing the needed facilities, though they are moving in the right direction. For example, most frame-based expert-system shells now incorporate a truth maintenance system.

If we were starting again today to build Cake, we would be tempted to start with a system like JOSHUA [10], because of its emphasis on modularity and extensibility. From our experience thus far, there is no facility in the current Cake system that we would leave out if we were starting again. There are several capabilities we would want to add, such as classification and more powerful (but still not totally general) techniques for quantified reasoning, such as those in [35].

Finally, on the topic of research methodology, we would like to observe that, even though Cake was motivated entirely by the needs of an application task rather than any specific set of research questions in knowledge representation and reasoning systems, it has turned out to be an interesting step in the evolution of such systems. For example, the first two architectural issues discussed above, which emerged naturally out of trying to build Cake, are in fact central to current research in knowledge representation and reasoning. We believe there is a moral here, and it is that there is much to be gained by trying to attack real and difficult problems with these techniques.

Acknowledgement

This article describes research primarily done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology supported in part by the following organizations: National Science Foundation under grants IRI-8616644 and CCR-898273, Advanced Research Projects Agency of the Department of Defense under Naval Research contract N00014-88-K-0487, IBM Corporation, NYNEX Corporation, and Siemens Corporation. The views and conclusions contained in this document are

[11] A special interning procedure interns `(< 3 0)` as `#<Term False>`.

[12] A small techicality: In order for this to work, rather than calling `FInstantiate` to create an instance of the plan to test, the Debugging Assistant must strip the `And` from the front of the plan definition and assert each constraint as a separate premise.

those of the authors and should not be interpreted as representing the policies, expressed or implied, of these organizations.

The Requirements Apprentice was the doctoral thesis of Howard Reubenstein supervised by Richard C. Waters, with Charles Rich as reader. DEBUSSI was the master's thesis of Ron Kuper, supervised by Charles Rich. We thank them for permission to use material from their reports. We also wish to thank Richard Waters for his technical advice and moral support throughout the development of Cake, as well as his thoughtful suggestions on this paper.

REFERENCES

[1] C. Rich and R. C. Waters, "The Programmer's Apprentice: A research overview," *IEEE Computer*, vol. 21, pp. 10–25, Nov. 1988.

[2] C. Rich and R. C. Waters, *The Programmer's Apprentice*. Addison-Wesley, Reading, MA and ACM Press, Baltimore, MD, 1990.

[3] Z. Manna and R. Waldinger, "A deductive approach to program synthesis," *ACM Trans. Programming Languages and Systems*, vol. 2, pp. 90–121, Jan. 1980.

[4] D. A. McAllester, "Reasoning utility package user's manual," Memo 667, MIT Artificial Intelligence Lab., Apr. 1982.

[5] 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, Vision, and Pattern Recognition*, (Tel Aviv, Israel), pp. 77–93, Dec. 1988.

[6] Y. A. Feldman and C. Rich, "Principles of knowledge representation and reasoning in the FRAPPE system," in *Proc. 6th Israeli Symp. on Artificial Intelligence*, (Tel Aviv, Israel), pp. 133–148, Dec. 1989.

[7] Y. A. Feldman and C. Rich, "Pattern-directed invocation with changing equations," *J. Automated Reasoning*, vol. 7, pp. 403–433, 1991.

[8] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Programming Languages and Systems*, vol. 1, pp. 245–257, Oct. 1979.

[9] A. M. Frisch, "A general framework for sorted deduction: Fundamental results on hybrid reasoning," in *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, (Toronto, Canada), pp. 126–136, May 1989.

[10] S. Rowley, H. E. Shrobe, and R. Cassels, "JOSHUA: Uniform access to heterogenous knowledge structures or why joshing is better than conniving or planning," in *Proc. 6th National Conf. on Artificial Intelligence*, (Seattle, WN), pp. 48–52, Aug. 1987.

[11] S. A. Miller and L. K. Schubert, "Using specialists to accelerate general reasoning," in *Proc. 7th National Conf. on Artificial Intelligence*, (St. Paul, MN), pp. 161–165, Aug. 1988.

[12] Y. M. Tan, "Supporting reuse and evolution in software design," Memo 1256, MIT Artificial Intelligence Lab., Oct. 1990. Master's thesis.

[13] R. C. Waters and Y. M. Tan, "Toward a Design Apprentice: Supporting reuse and evolution in software design," *ACM SIGSOFT Software Engineering Notes*, vol. 16, pp. 33–44, Apr. 1991.

[14] H. B. Reubenstein and R. C. Waters, "The Requirements Apprentice: Automated assistance for requirements acquisition," *IEEE Trans. Software Engineering*, vol. 17, pp. 226–240, Mar. 1991.

[15] R. G. Babb, "Workshop on models and languages for software specification and design," *IEEE Computer*, vol. 18, pp. 103–108, Mar. 1985.

[16] R. I. Kuper, "Dependency-directed localization of software bugs," Technical Report 1053, MIT Artificial Intelligence Lab., May 1989. Master's thesis.

[17] W. L. Johnson and E. Soloway, "PROUST: Knowledge-based program understanding," *IEEE Trans. Software Engineering*, vol. 11, pp. 267–275, Mar. 1985.

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

[19] J. Doyle, "A truth maintenance system," *Artificial Intelligence*, vol. 12, pp. 231–272, 1979.

[20] W. Swartout, "The GIST Behavior Explainer," in *Proc. 3rd National Conf. on Artificial Intelligence*, (Washington, DC), pp. 402–407, Aug. 1983.

[21] D. A. McAllester, "An outlook on truth maintenance," Memo 551, MIT Artificial Intelligence Lab., Aug. 1980.

[22] G. J. Sussman and G. L. Steele, "CONSTRAINTS—A language for expressing almost-hierarchical descriptions," *Artificial Intelligence*, vol. 14, no. 1, pp. 1–40, 1980.

[23] A. K. Macworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, pp. 99–118, Feb. 1977.

[24] G. L. Steele Jr., *Common Lisp: The Language (Second Edition)*. Burlington, MA: Digital Press, 1990.

[25] Y. A. Feldman and C. Rich, "Reasoning with simplifying assumptions: A methodology and example," in *Proc. 5th National Conf. on Artificial Intelligence*, (Philadelphia, PA), pp. 2–7, Aug. 1986.

[26] G. Nelson and D. C. Oppen, "Fast decision procedures based on congruence closure," *J. ACM*, vol. 27, pp. 356–364, Apr. 1980.

[27] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard* ML. Cambridge, MA: MIT Press, 1990.

[28] A. G. Cohn, "A more expressive formulation of many sorted logic," *J. Automated Reasoning*, vol. 3, pp. 113–200, June 1987.

[29] R. W. Weyhrauch, "Prolegomena to a theory of mechanized formal reasoning," *Artificial Intelligence*, vol. 13, pp. 133–170, 1980.

[30] M. L. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision* (P. H. Winston, ed.), pp. 211–277, New York, NY: McGraw-Hill, 1975.

[31] P. Devanbu et al, "LaSSIE: A knowledge-based software information system," *Comm. ACM*, vol. 34, pp. 34–49, May 1991.

[32] R. C. Waters, "The Programmer's Apprentice: A session with KBEmacs," *IEEE Trans. Software Engineering*, vol. 11, pp. 1296–1320, Nov. 1985.

[33] C. Rich, "A formal representation for plans in the Programmer's Apprentice," in *Proc. 7th Int. Joint Conf. Artificial Intelligence*, (Vancouver, British Columbia, Canada), pp. 1044–1052, Aug. 1981.

[34] J. F. Allen, "The RHET system," *ACM SIGART Bulletin*, vol. 2, pp. 1–7, June 1991.

[35] D. McAllester, R. Givan, and T. Fatima, "Taxonomic syntax for first order inference," in *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, pp. 289–300, 1989.