The Requirements Apprentice: Automated Assistance for Requirements Acquisition

Howard B. Reubenstein, *Member, IEEE,* and Richard C. Waters, *Senior Member, IEEE*

Abstract—**Requirements acquisition is one of the most important and least supported parts of the software development process. The Requirements Apprentice (RA) assists a human analyst in the creation and modification of software requirements. Unlike most other requirements analysis tools, which start from a formal description language, the focus of the RA is on the transition between informal and formal specifications. The RA supports the earliest phases of creating a requirement, in which *ambiguity, contradiction,* and *incompleteness* are inevitable.**

From an artificial intelligence perspective, the central problem the RA faces is one of *knowledge acquisition.* The RA develops a coherent internal representation of a requirement from an initial set of disorganized imprecise statements. To do so, the RA relies on a variety of techniques, including dependency-directed reasoning, hybrid knowledge representation, and the reuse of common forms (clichés).

The Requirements Apprentice is being developed in the context of the Programmer's Apprentice project, whose overall goal is the creation of an intelligent assistant for all aspects of software development.

Index Terms—**Informality resolution, knowledge acquisition, requirements analysis, reuse.**

I. The Programmer's Apprentice

THE Programmer's Apprentice project [34], [35] is studying how software engineers analyze, synthesize, modify, specify, verify, and document software systems and how these tasks can be automated. Recognizing that it will be a long time before it is possible to fully automate any of these tasks, the near-term goal of the project is the development of a system, called the Programmer's Apprentice, which can act as a software engineer's junior partner and critic, taking over simple tasks completely and assisting with more complex tasks.

Viewed most simply (see Fig. 1) the software-development process has, at one end, the desires of an end-user and, at the other end, a program that can be executed on a machine. The part of the software process closest to the user is typically called requirements acquisition; the part of the process nearest the machine is typically called implementation; the area in the middle is generally described as design. The project is following the strategy of building demonstrations of parts of the Apprentice, working inward from the two ends of Fig. 1.

A key result of the project's research on the right end of Fig. 1 is a demonstration system called the Knowledge-Based Editor in Emacs (KBEmacs) [35], [46], which can automatically implement a program once a software engineer has selected the appropriate algorithmic fragments to use. Work is currently underway on a Design Apprentice [33], [35] that will go beyond this by taking over the low-level aspects of design. This system will be able to critique a high-level design provided by a software engineer and automatically make the low-level design decisions that follow from it.

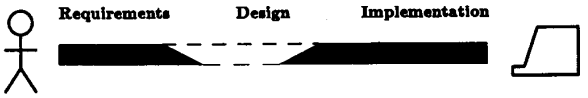


Fig. 1. Incremental approach to the Programmer's Apprentice.

The project's research on the left end of Fig. 1 has resulted in a demonstration system called the Requirements Apprentice (RA) [36], [37], [38], which can assist an analyst in the creation and modification of software requirements. The way the RA fits into the requirements acquisition process is discussed in Section II. The main body of the paper (Section III) discusses two transcripts showing the currently running RA demonstration system in action. This presentation is interspersed with discussions of the techniques used by the RA. In Section IV, the paper concludes with a discussion of the future goals of research on the RA.

II. The Requirements Apprentice

Research on requirements acquisition is valuable for two reasons. First, from the perspective of software engineering, requirements acquisition is perhaps the most crucial part of the software process. Studies (e.g., [3]) indicate that errors in requirements are more costly than any other kind of error. Furthermore, requirements acquisition is not currently supported very well by software tools. Second, from the perspective of artificial intelligence, requirements acquisition is a good domain in which to pursue fundamental questions related to knowledge acquisition in general.

It is useful to distinguish three phases in the requirements acquisition process: *elicitation, formalization,* and *validation.* The elicitation phase usually takes the form of a *skull session,* whose goal is achieving a consensus among a group of users about what they want. In the elicitation phase, the requirements analyst acts primarily as a facilitator, perhaps with the aid of a debriefing methodology, such as WISDM [49], JAD [8], or viewpoint resolution [24]. The end product of the elicitation phase is typically an informal requirement, such as the university library database description in Fig. 2 (taken from [2]). (This requirements sketch has been used as a benchmark for comparing tools at the last four International Workshops on Software Specification and Design and is used for the first transcript in Section III.)

Most work on software requirements tools (e.g., [18], [22], [45]) focuses on the *validation* phase. The main goal of this part of the requirements acquisition process is increasing confidence

Manuscript received March 14, 1990; revised September 11, 1990. Recommended by the Guest Editors for the special section. This work was supported in part by the National Science Foundation under Grant IRI-8616644, in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract N00014-88-K-0487, and in part by the IBM, NYNEX, Siemens, and Microelectronics and Computer Technology corporations.

H. B. Reubenstein is with the MITRE Corporation, Bedford, MA 02139.

R. C. Waters is with the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

IEEE Log Number 9041660.

```
Consider a university library database.  There are two types of users: normal
borrowers and users with library staff status.  The database transactions are:
  (1) Check out a copy of a book.
  (2) Return a copy of a book.
  (3) Add a copy of a book to the library.
  (4) Remove a copy of a book from the library.
  (5) Remove all copies of a book from the library.
  (6) Get a list of titles of books in the library by a particular author.
  (7) Find out what books are currently checked out by a particular borrower.
  (8) Find out what borrower last checked out a particular copy of a book.

These transactions have the following restrictions:
  R1 - A copy of a book may be added or removed from the library only by
       someone with library staff status.
  R2 - Library staff status is also required to find out which borrower last
       checked out a copy of a book.
  R3 - A normal borrower may find out only what books he or she has checked
       out.  However, a user with library staff status may find out what
       books are checked out by any borrower.

The requirements that the database must satisfy at all times are:
  G1 - All copies in the library must be checked out or available for check out.
  G2 - No copy of a book may be both checked out and available for check out.
  G3 - A borrower may not have more than a given number of books checked out
       at any one time.
  G4 - A borrower may not have more than one copy of the same book checked out
       at one time.
```

Fig. 2. Example of an informal requirement (copyright © IEEE 1985).

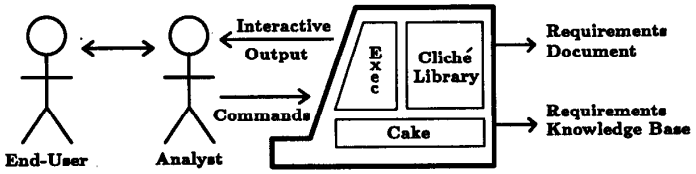


Fig. 3. The Requirements Apprentice.

that a given requirement corresponds to the end-user's desires. In current research approaches, this is achieved by applying simulation, symbolic execution, and various kinds of logical analysis to a formal specification. This work does not, however, address the key question of how a formal specification is constructed in the first place. For example, in both [20] and [47] the discussion begins by simply exhibiting the translation of the informal requirement of Fig. 2 into a formal specification.

The focus of the RA is on the formalization phase that bridges the gap between an informal and formal specification. This is a crucial area of weakness in the current state of the art. For example, it was reported at the Second Workshop on Software Specification and Design [2, p. 107] that some of the greatest problems stemmed from "the process of completing the system analysis work needed to translate the informal specification into the appropriate input for the tools."

A. Interacting with the RA

Fig. 3 shows the role of the RA in relation to other agents involved in the software process. Note that the RA does not interact directly with an end-user, but is an assistant to a requirements analyst. The main benefit of excluding direct interaction with the end-user is that it avoids having to deal with the syntactic complexity of natural language input. Free-form natural language input would be essential for interaction with a naive end-user. However, a professional analyst should have no trouble using a more restrictive command language.

The RA produces three kinds of output. Interactive output notifies the analyst of conclusions drawn and inconsistencies detected while requirements information is being entered. A machine-manipulable Requirements Knowledge-Base (RKB) represents everything the RA knows about an evolving requirement. (The idea of using a central requirements data base goes back at least to PSL/PSA [44].) In the long term, it is intended that the RKB be accessed directly by other tools and components of the Programmer's Apprentice. Finally, the RA can create a more or less traditional requirements document summarizing the RKB. Since many software projects are contractually obligated to provide a document of this form, automatically generating (and regenerating) such a document is a valuable near-term feature of the RA.

Fig. 3 shows that the RA is composed of three modules. *Cake* [13], [32] is a knowledge-representation and reasoning system, which supports the reasoning abilities of the RA (and the rest of the Programmer's Apprentice). Cake provides basic facilities for propositional deduction (including the detection of contradictions), reasoning about equalities, maintenance of dependencies between deduced facts, and incremental retraction of previously asserted facts.

The *executive* handles interaction with the analyst and provides high-level control of the reasoning performed by Cake. The analyst communicates with the executive by issuing commands. Each command provides fragmentary information about an aspect of the requirement being specified. The immediate implications of a command are processed by Cake, added to the RKB, and checked for consistency. If the analyst makes a change in the description (to correct an inconsistency or simply due to a change of mind) the executive incrementally incorporates this modification into the RKB, retracting invalidated deductions and replacing them with new deductions.

The *cliché library* is a declarative repository of information relevant to requirements in general and to domains of particular interest. When creating a requirement, the information unique to the particular problem comes from the analyst. However the bulk of general information about the domain comes from the cliché library.

While a few other requirements tools make significant use of domain knowledge, most do not. For example, the typical tool for the symbolic execution of a form requirements language operates without any knowledge other than the semantics of the language itself. Such a tool can do a lot to help identify problems with what is in a given requirement. However, it is not in a position to say very much about what might be missing from the requirement. In contrast, having specific knowledge about what should be in requirements associated with a particular domain makes it possible for the RA to critique what is not in a requirement as well as what is in the requirement.

An interesting point of comparison with the RA is the Kate system [14]. Fickas has proposed an interactive system that will provide assistance over the entire requirements acquisition process, emphasizing specification design. The system will critique an evolving requirements specification [15] and will eventually be capable of generating examples to support its observations. To make this feasible, the system will rely heavily on knowledge of a particular domain (resource management). In related work, Robinson [39] describes a technique for performing tradeoffs when merging specifications that were written by users with different goals. The basis of this is a domain model of the influence of specification properties on goals.

B. Dealing with Informality

Viewed from an artificial intelligence perspective, research on the RA attacks the general problem of *knowledge acquisition.* In doing so, it concentrates on the issue of *informality* and the process by which informal descriptions become formal ones. (It is interesting to note that while the RA avoids involvement with the surface syntax of natural language, problems of knowledge acquisition are central to the deeper issues of natural language understanding.)

From the knowledge-acquisition perspective in general, the work most similar to the RA has been concerned with providing automated assistance for acquiring new rules for expert systems. Most systems, such as Teiresias [10], Seek [29], and KRITON [11], are limited to fairly simple well-formedness and consistency checking. Other systems, such as KLAUS [16], ROGET [6], and OPAL [27] provide automated assistance for the elicitation and acquisition of new concepts and vocabulary.

The knowledge-acquisition project closest to the goals of the RA is BLIP [48]. BLIP assists a user in a process called *sloppy modeling,* which allows some informality in the user's input. However, while BLIP functions as an intelligent notepad, BLIP is more limited in scope than the RA, because it does not make use of a domain model as a source of knowledge and expectations.

On the question of informality in particular, the RA continues in the tradition of the SAFE project [4]. Balzer, Goldman, and Wile were the first to argue that informality is an inevitable (and ultimately desirable) feature of the specification process. They began by studying actual natural language software specifications, cataloging the kinds of informality they found. At the end of the project, the SAFE prototype system succeeded in automatically producing formal specifications (in the language AP3) from preparsed informal natural language specifications for a number of examples. The primary difference between SAFE and the RA is that while the SAFE work points to the importance of domain knowledge in resolving informality, it lacks the notion of clichés as a way of representing, organizing, and applying this knowledge. In addition, SAFE is an automatic batch system, whereas the RA is an interactive assistant.

The following is a list of general features that characterize informal communication between a speaker and hearer. These features are based on the discussion in [4] and the experience gained from applying the RA to the informal requirement in Fig. 2.

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

These kinds of informality are not a matter of the speaker being lazy or incompetent. Rather, informality is an essential part of the human thought process. It is part of a powerful *debugging* strategy for dealing with complexity, which shows up in many problem solving domains: start with an almost-right description and then incrementally modify it until it is acceptable [42]. Thus, having the RA deal with informality is not just a question of being user friendly—it is a fundamental prerequisite. The following subsections discuss the RA's support for resolving the six kinds of informality listed above.

C. Reuse of Clichés

Expert engineers rarely construct complex artifacts (automobiles, electronic circuits, or requirements specifications) by starting from first principles. Rather, they bring to the task their previous experience in the form of knowledge of the commonly occurring structures in the domain. The term *cliché* is used here to refer to these commonly occurring structures. In normal usage, the word cliché has a pejorative sound that connotes overuse and a lack of creativity. However, in the context of engineering problem solving, this kind of reuse is a positive feature.

Abbreviation in the form of references to clichés is essential for effective communication on any topic. There is ample evidence that it is difficult, if not impossible, to acquire new knowledge unless one already has a large amount of relevant old knowledge. Imagine trying to communicate the requirements for an inventory control system to a person who knows nothing about either information systems or inventories.

A major goal of research on the RA is the codification of clichés in the domain of software requirements. This codification includes both clichés of broad applicability and more specific

clichés in particular application areas. An important benefit of orienting the RA around the use of clichés is that domain-specific knowledge can be provided as data, rather than built into the system. New domains can be covered by defining new clichés.

The importance of codifying domain knowledge is a theme which the RA shares with systems such as Φ-NIX [5] and DRACO [28]. However, neither of these projects focuses on supporting informality.

Notions similar to the cliché idea appear in software engineering in the work of Arango and Freeman [1] (domain models), Harandi and Young [17] (design templates), and Lavi [21] (generic models); and in artificial intelligence in the work of Minsky [25], [26] (frames, concept germs) and Schank [40] (conceptual structures).

D. Disambiguation

The principal source of ambiguity in the analyst's input is that the words used each have many different meanings. For instance, the word *sort* has meanings associated with several different methods of sorting as well as other, unrelated meanings (e.g., sort in the sense of a type). As subsequent information is collect, the exact meaning of a given use of the word sort has to be determined from the context. This is done by *classification* [19], [41]—i.e., "intersecting" the information underlying the words used in the analyst's statements to determine a mutually consistent set of specialized meanings.

This kind of word disambiguation is the RA's principal mechanism for fleshing out the description being acquired. Each time a specialized meaning is identified, significant amounts of additional information are incorporated into the requirement by instantiating the more specialized clichés involved. This makes the requirement more precise, leaving less room for (mis)interpretation.

E. Order Independence

The basic reasoning algorithms of Cake are order independent—they arrive at the same conclusions no matter what order facts are presented to them. As a result, the RA is fundamentally insensitive to the order of presentation. With regard to any particular deduction, the RA simply waits until the final piece of appropriate information appears.

It should be noted that the RA can detect certain obvious omissions of information, e.g., a reference to an object before it is defined. These are maintained in a list of pending issues, which can be viewed as suggestions about what the analyst should present next. However, the analyst has full control of the order of statements.

F. Contradiction Detection/Resolution

Because Cake is not capable of making all possible deductions from a set of facts, the RA is not capable of detecting every contradiction that may be present in the analyst's input. However, the RA examines every aspect of the input and detects every contradiction that follows from sufficiently simple deductions.

Due to an explicit representation of proof structures, Cake can continue to make valid deductions even in the presence of a contradiction. As a result, the analyst has the option of ignoring a contradiction and continuing on. However, any contradictions must be resolved before a requirement can be considered complete.

When the analyst chooses to fix a contradiction, the RA can provide help in a number of ways. To start with, the RA provides an in-depth description of each contradiction including the deductions that lead to it and the premises that underlie it. The simplest way for the analyst to resolve a contradiction is to retract one of the underlying premises. However, this results in a loss of information. Depending on the form of a given premise, the RA can often suggest related premises that might be substituted for a problematical premise. For example, if a premise asserts a type for an object, the type can be replaced with a sibling or parent type.

G. Detecting Incompleteness and Filling in Details

Incompleteness is fundamentally harder to detect than contradiction. Even with a perfect reasoner, there is no way to detect the absence of information that is orthogonal to the current state of a requirement. As a result, the end-user has to be the final arbiter of completeness. Nevertheless, the RA can detect some kinds of incompleteness and can help fill in missing details in the presentation.

Each time a cliché is referred to, a number of expectations are generated in the form of roles that need to be filled in. Some roles have default values associated with them that are used when no other value is specified. Constraints between roles allow some roles to be filled in based on the contents of other roles. Other roles are optional and do not have to be filled in. The RA considers a requirement to be incomplete until every mandatory role is filled in.

H. Support for Evolution

Inaccuracy is the hardest of all to detect. It cannot be detected unless it leads to a detectable contradiction or incompleteness. Further, only the end-user knows how to correct an inaccuracy. However, the RA provides a number of mechanisms that facilitate the rapid evolution of a requirement when inaccuracies are corrected, or when the end-user simply has a change of mind about something.

Because the Cake reasoning system keeps a complete record of every deduction and supports incremental reasoning, it is easy to retract one statement and replace it with another. Because the requirements document produced by the RA is automatically generated from the RKB, it is easy to get a clean and consistent copy of the document after any change. The contradiction and incompleteness detection mechanisms above can be used to assess the effects of a change. Finally, as illustrated at the end of the next section, the RA supports a special *reformulate* command that makes certain kinds of major changes easy to state.

III. A Session with the RA

The following is an annotated transcript showing an interaction with the currently running version of the RA. The capabilities of the RA are the result of interplay between a number of techniques including classification, plausibility checking, and contradiction detection. In the following presentation, sections describing groups of commands to the RA are interspersed with sections describing the techniques that support the processing triggered by the commands.

A. The RA Interface

When looking at the upcoming transcript, it is important to focus on its content, not its form. The content illustrates the

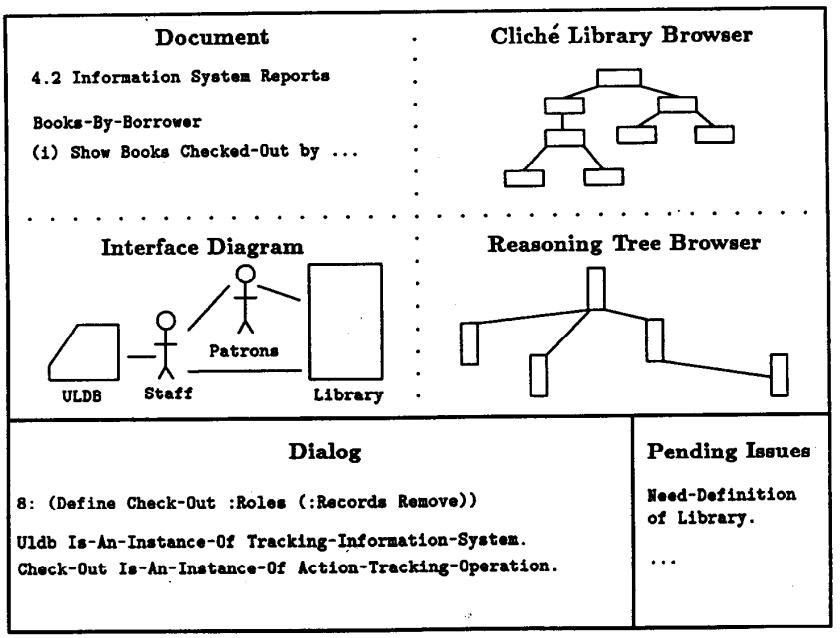


Fig. 4. The proposed interface of the RA.

fundamental capabilities of the RA. However, the form is at best a weak shadow of the full intended RA interface. As shown in Fig. 4, the full interface is intended to display three basic kinds of information. The main part of the screen is used to display information about the requirement as it evolves. A dialog window is used for typing commands to the RA and for displaying the immediate responses of the RA. A third window displays a list of pending issues that need to be resolved.

The main part of the screen is essentially a window into the RKB. The information displayed could take many forms. It could be textual (perhaps rendered in a formal specification language) or diagrammatic. It is intended that many kinds of modifications in the RKB be effected by directly editing these displays.

The main part of the screen can also be used to inspect the contents of the cliché library to gain a better idea of what the RA knows and what words are appropriate to use. A second kind of browser can be used to inspect the reasoning behind the conclusions drawn by the RA.

The RA maintains a list of pending issues that need to be resolved. These issues can be viewed as requests for particular kinds of information. Listing these issues, without forcing them to be immediately resolved, allows the analyst to control the pace of the interaction. However, the requirement cannot be considered complete while there are any pending issues.

In contrast to Fig. 4, the current RA demonstration only supports a minimal debugging interface corresponding roughly to the dialog window. The user can create a document corresponding to the RKB, view the cliché library, display reasoning trees, and list the pending issues. However, there is as yet no support for diagrams and all of the interaction takes place through the dialog window.

As with free-form natural language input, "creature comforts" in the interface are deemphasized not because they are unimportant, but rather because they are largely orthogonal to many of the key issues underlying the RA, and therefore can be temporarily side-stepped.

B. Commands 1–4: Echo/Paraphrase

The RA is implemented in Common Lisp. Each command to the RA is a Lisp expression whose first component specifies the type of command. As a prompt for a new command, the RA prints out an identifying number for the command. The output generated by the RA in response to a command is shown on the lines that follow it.

The commands shown in this transcript were chosen based on the library data base problem in Fig. 2. In Command 1 (see below), the analyst directs the RA's attention to a requirement to be called *library-system.* If there had already been a requirement of this name, the RA would have returned to a consideration of this requirement. In this case, there is no preexisting requirement and a new RKB is created.

```
1. (Find-Requirement Library-System)
Beginning-A-New-Requirement-Called-The
   Library-System.
Library-System Is-An-Instance-Of Requirement.
```

The RA decides what to say in response to a command by monitoring changes in the RKB. Finding things to say is not so much the problem as deciding what *not* to say. The underlying reasoning system typically deduces an avalanche of facts, only a few of which are worth reporting. The RA collects a list of all the new and changed facts matching particular syntactic forms. It then filters these facts further to avoid redundancy. For example, if an object is deduced to belong to several different types, all but the most specific type assertions are pruned away. The RA then displays the remaining facts in an order based on the objects they refer to. (For a further discussion of the problems involved in choosing an appropriate level of explanatory detail see [43].)

A significant portion of the output generated by the RA essentially echoes the analyst's command in a paraphrased form.

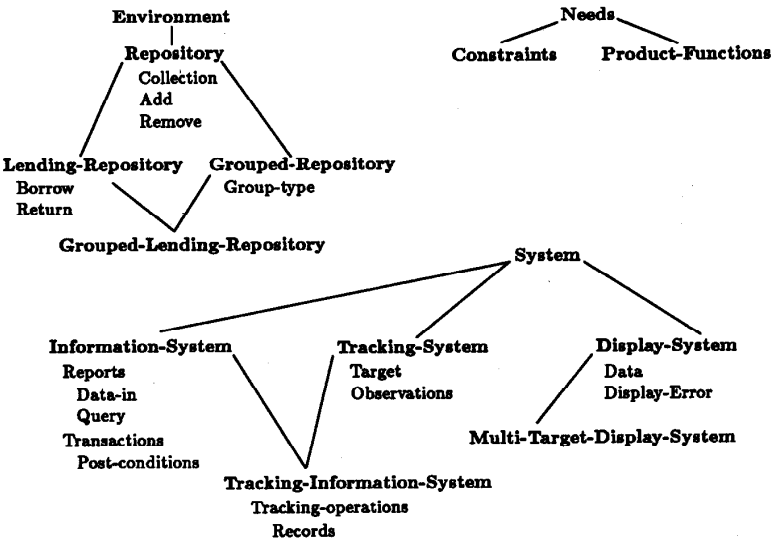


Fig. 5. The structure of a fragment of the requirements cliché library.

This echoing allows the analyst to judge whether or not the RA has interpreted the command correctly. The rest of the output reports conclusions drawn by the RA. In the interest of brevity, the remainder of the transcript suppresses the paraphrastic output and focuses on the deductive output.

In Command 2, the analyst gives a name to the system being specified in the requirement. In a *define* command, the first argument is the word being defined. The word is often followed by one or more keywords (beginning with colons) specifying the type of the word. (As discussed in the next subsection, the requirements cliché library is organized as a hierarchy of types.) The remainder of the arguments to the define command are pairs of keywords and values that specify particular features of the definition. In Command 2, a short synonym (ULDB) is specified, which can be used on input and output when referring to the university library database.

```
2 : (Define University-Library-Database : System
      : Synonym Uldb)
...
```

Command 3 specifies that the environment of the requirement is a library. In the output produced by the RA, the word library appears in quotes. This signifies that although the RA has recorded that the UL is a library, it does not know what a library is. Words are surrounded by quotes unless they are defined by requirements clichés or have been explicitly defined by the analyst in define commands. This is done to indicate that these words must eventually be defined.

```
3 : (Define University-Library : Environment
      : Library: Synonym Ul)
Ul Is-An-Instance-Of "Library".
...
```

Command 4 specifies the prime need in the requirement being constructed. The form (!Logic ···) specifies that its body is a logical formula. The RA's response to this command can best be understood through a discussion of the requirements cliché library.

```
4 : (Need (!Logic (Tracks Uldb Ul)))
Uldb Is-An-Instance-Of Tracking-System.
...
```

C. A Library of Requirements Clichés

Conceptually, a cliché consists of a set of *roles* and *constraints* between them. The roles of a cliché are the parts that vary from one use of the cliché to the next. The constraints specify how the roles interact and place limits on the parts that can be used to fill the roles.

Fig. 5 is a schematic outline of a portion of the RA's current cliché library. The top-level division of the figure into three areas (environment, needs, and system) reflects the theory that requirements are composed of three main kinds of information. The *environment* section describes the context of the requirement—those aspects of the requirement that are not under the control of the end users and are therefore not up for discussion (e.g., the fact that a library of books exists). The *needs* section gives a high-level description of the desires of the end-users (e.g., the need to keep track of which books are in the library). The *system* section contains a high-level specification for a system that meets the needs (e.g., an information system that keeps track of particular pieces of data and furnishes particular kinds of reports).

Three clichés in Fig. 5 that are prominent in the library requirement transcript are: repository, information system, and tracking system. A *repository* is an entity in the physical world. The basic function of a repository is to ensure that items that enter the repository are available for later removal. The repository cliché has a number of roles including: the *collection* of items stored in it, the *patrons* that utilize the repository, and the *staff* that manages the repository. The two key operations on a repository are *adding* and *removing* items.

There are a variety of physical constraints that apply to repositories. For example, since each item has a physical existence, it

```
(Def-Cliche Report (Functional-Requirement)
  (Def-Roles
    (Data-In Data-Spec)
    (Data-Out Data-Spec)
    (Query Logical-Constraint)
    (Accessed-Information (Set-Of Data-Spec))
    Purpose)
  :Preconditions (Information-System (Home-System ?Self))
  :Consequents
    (Truep ?Query)
    (Subset ?Accessed-Information (Stored-Information (Home-System ?Self)))
    (= ?Accessed-Information (Analyze-Information-Accessed ?Query))
  :Equality-Views (?Query)
  :Overview-Text (!Text "The report ?self answers the following query."))

(Def-Cliche Tracking-Information-System-Report (Report)
  (Def-Roles Target)
  :Preconditions (Tracking-Information-System (Home-System ?Self))
  :Consequents (= ?Target (Target (Home-System ?Self)))
  :Overview-Text
    (!Text "The tracking-information-system-report ?self provides
     information about what the state of the ?target is believed to be."))
```

Fig. 6. Two example cliché frame types.

can only be in one place at a time and therefore must either be in the repository or not.

There are several kinds of repositories. Simple repositories merely take in items and then give them out. *Lending repositories* support the lending of items, which are expected to be returned. *Grouped repositories* contain items aggregated into groups of similar items. Example repositories include: storage warehouse (simple repositories for unrelated items), grocery stores (simple repositories for items grouped in classes), and rental car agencies (lending repositories for items grouped in classes).

In contrast to the repository cliché, the *information system* cliché describes a class of programs rather than a class of physical objects. The intent of the information system cliché is to capture the commonality between programs such as personnel systems, bibliographic databases, and inventory control systems.

The central roles of an information system are a set of *reports* that display parts of the data being stored and a set of updating *transactions* that create/modify/delete the data. Each kind of report and transaction is in turn a cliché with its own characteristic roles. Additional roles of an information system include: the *staff* that performs the transactions and the *users* that obtain the reports.

A *tracking system* keeps track of the state of some *target* object. It does this by making *observations*. The observations can either sense the state of the object directly (e.g., the way a radar keeps track of the position of airplanes) or observe the operations that change the state (e.g., the way a set of turnstiles keeps track of the number of people in a building).

The *tracking information system* cliché combines the features of an information system and a tracking system. The main content of this combined cliché is a set of constraints that relate roles of the information system part to the tracking system part. For example, the tracked data becomes the data in the information system. Further, the observations in the tracking system correspond to transactions in the information system.

To operate in the intended manner, the RA must have a considerable amount of background knowledge relevant to the requirement at hand. However, it is unrealistic to assume that this knowledge will ever be complete. In consonance with this, the RA's current cliché library contains extensive knowledge of information systems, tracking systems, and repositories but no knowledge about libraries or library information systems per se.

Returning to a consideration of Command 4, the most interesting aspect of the command is its use of the word *tracks*. The way the RA attaches a meaning to this word in this context is a simple example of what goes on in general when the RA tries to interpret the analyst's statements. Tracks has several specializations in the cliché library and thus the use of the word tracks is ambiguous.

There is nothing directly connected to either ULDB or UL that can be matched up with the use of tracks as a relation between them. However, the meanings of the word tracks that are consistent with what is known about ULDB and UL are all associated with various specializations of the tracking system cliché. As a result, the RA assumes that ULDB is a tracking system. This is reported to the analyst, who could choose to override the assumption.

After processing Command 4, the RA has not succeeded in totally disambiguating what the word tracks means. Rather, it has located an intermediate specialization of the word that reflects an intermediate level of understanding. The cliché library is designed to be a multiple hierarchy with large numbers of intermediate terms that can be used in this way.

D. Representing Clichés

The clichés in the requirements cliché library are represented as frames linked by constraints and arranged in an inheritance hierarchy. Each cliché is represented as a frame type. Each use of a cliché is represented as an instance of the associated cliché frame type. The roles of a cliché are represented by the slots of the associated cliché frame type. The constraints on the cliché are represented by a predicate on these slots. Related ideas on how to apply knowledge representation techniques to software requirements can be seen in the RML language of Greenspan [7].

Two clichés from the RA's current cliché library are shown in Fig. 6. Each is defined using a special Lisp macro `Def-Cliche`. The first argument of this macro is the name of the cliché. The second is a list of the parents of the cliché. The cliché inherits all the properties of all its parents. However, individual properties can be overridden in the definition of the cliché.

The third argument of `Def-Cliche` is the subform `Def-Roles`, which defines the roles of the cliché. Each role specification is either the name of a role or a list of a name and a type. Role names are referred to in the rest of the definition by prefixing them with question marks.

The main body of a `Def-Cliche` specifies constraints on the cliché. These are divided into more than a dozen kinds, which are introduced by keywords (e.g., `: Preconditions`, `: Consequents`). Each kind of constraint has a different kind of processing associated with it. For instance, the preconditions of a cliché have the property that when they are all satisfied, it is safe to assume that the cliché as a whole is applicable. They are used as the primary basis for disambiguation.

Consider the cliché *report* in Fig. 6. Its parent is *functional-requirement*—an abstract cliché that is the parent of reports, transactions, and many other kinds of specifications. The report cliché has five roles, four of which have type constraints. Each of these type constraints acts as a precondition on the cliché. An explicit precondition states that reports are only associated with information systems.

(When an instance of a cliché frame type is being processed, the variable `?Self` is bound to the frame instance as a whole. The form `(Home-System ?Self)` returns the frame instance, if any, that points to the current frame. For instance if a report `R1` is in a slot `S` of some other cliché instance `X`, then `(Home-System R1)` = `X`.)

Most of the constraints on a cliché are represented as `: Consequents` in the corresponding cliché frame type. The consequents are a mixture of logical predicates supported by Cake's standard reasoning mechanisms and special purpose procedures. For instance, the second `: Consequent` of report specifies that the information accessed by the query must be part of the information stored by the associated information system. (The function `Analyze-Information-Accessed` is a special-purpose procedure that inspects a query to determine what information is accessed.)

In addition to preconditions and consequents, cliché frames can have *default constraints* (constraints that are marked as likely candidates for retraction should a contradiction arise) and *default role values* (values that are used if no other value is explicitly specified).

The remainder of a `Def-Cliche` specifies various annotations that are used by the RA. For instance, the `: Equality-Views` annotation in the report cliché specifies that two reports are to be considered equal if they have the same query. The `: Overview-Text` annotation specifies how to construct a brief description of an instance of a cliché in a requirements document.

The cliché *tracking information system report* in Fig. 6 is a specialization of the report cliché, which is associated with tracking information systems. It adds a new role and some additional constraints and annotation.

E. Commands 5–8: Pending Issues

An important goal of the RA is to leave the analyst in control of the interaction. The RA is designed to be noninvasive in that it primarily just processes what the analyst says without much comment, even if it is not able to understand these statements completely. However, the RA maintains a list of issues that need to be resolved. In Command 5, the analyst requests a display of this list. (As illustrated in Fig. 4 it would be better if this list were automatically displayed in some unobtrusive fashion.)

```
5 : (Show-Pending-Issues)
1- Need-Definition of "Library".
2- Need-Further-Disambiguation of Tracks.
3- Need-Definition (Item-States U1).
```

At this point in the transcript, three issues are pending. The word library needs to be defined, and the word tracks needs to be further disambiguated. In addition, the fact that ULDB tracks UL means that there must be some state inside UL that is being tracked. The nature of this state needs to be specified.

Maintenance of the list of pending issues is one of the ways that the RA helps ensure that the requirement will be complete. Words mentioned by the analyst pull in clichés, which contain roles that have to be filled with additional information. Filling in this information makes the requirement more complete but may pull in additional clichés. This process continues until a point of closure is reached where filling in the remaining information does not lead to the inclusion of additional concepts with loose ends.

In Command 6, the analyst defines a library to be a kind of `: Ako` (i.e., a specialization of) a repository that contains books. By means of the repository cliché, the RA knows what the general concept of a repository is. However, it does not know what a book is.

```
6 : (Define Library : Ako Repository : Defaults
      (: Collection-Type Book))
U1.Collection-Type Has-Value "Book".
...
```

Command 6 answers two of the pending issues shown above. As well as defining the word library, it specifies what the state inside the UL is—namely, the set of books contained in it. (The syntax "X.Y" is used to refer to the `Y` role of `X`.)

Command 7 defines a book to be a physical object with certain roles. Command 8 specifies that the ISBN of a book (the International Standard Book Number) is a role containing a single integer value and that this value uniquely identifies the book.

```
7. (Define Book : Ako Physical-Object : Member-Roles
     (Title Author Isbn))
...
8. (Define Book.Isbn : Ako Integer
     : Cardinality Single : Unique-Id T)
...
```

F. Command 9: Disambiguation

Having defined the basic environment of the library system, the analyst uses Command 9 to begin the definition of the functional requirements. Interpreting this command presents a significant challenge to the RA because no type is specified and, while the words *records* and *remove* are both mentioned in the cliché library, neither one has a unique definition. This problem is resolved by locating something that can *record remove* and is meaningful in the context of a tracking system. In particular, if the ULDB is further specialized to a tracking information system, then *check-out* can be understood as a tracking operation (see Fig. 5). In addition, since remove corresponds to one of the fundamental operations that alter the state of a repository, it can be concluded that the ULDB operates by indirect-observation and that check-out can be further specialized as an *action tracking operation*—a transaction that tracks actions that alter the state of a target object.

```
9 : (Define Check-Out : Roles (: Records Remove))
Uldb Is-An-Instance-Of Tracking-Information-
  System.
Uldb.Manner-Of-Observation Is-An-Instance-
  Of Indirect-Observation.
Check-Out Is-An-Instance-Of Action-Tracking-
  Operation.
Check-Out.Object-Type Has-Value Book.
Check-Out.Objects Has-Value (!The (?0) Such-That
  (= (Isbn ?0) $Input)).
Check-Out.Records Has-Value Remove-Repository.
...
```

The information in the action tracking operation cliché, along with the current RKB, permits the RA to deduce a significant amount of information about check-out. For example, since the ULDB tracks the UL and the UL is a repository that stores books, check-out must keep track of objects of type book. A default constraint on tracking operations specifies that, in the absence of anything else being said, the input of check-out should be a unique identifier for the kind of object being tracked and that the object being tracked is the object corresponding to this identifier.

G. Support for Classification

The RA disambiguates/refines the analyst's statements relative to the knowledge in the cliché library through both conservative and heuristic *classification*. The more specific the classification of the terms in a statement, the better the understanding of the statement, since more detailed clichés can be instantiated. Classification can be viewed as a kind of recognition algorithm that opportunistically applies information in the cliché libraries to the information in the RKB as it evolves.

In particular, as soon as a word is associated with a given cliché (say tracking system) the RA begins monitoring the preconditions of each specialization of this cliché (e.g., tracking information system) to see if they are contradicted, partly satisfied, or completely satisfied. Specializations whose preconditions are contradicted are eliminated from consideration. Using standard conservative methods of classification [19], [41], as soon as a set of preconditions is completely satisfied, the corresponding specialized cliché is adopted.

Conservative classification is useful, but cautious in nature. In order to push the acquisition process forward faster, the RA jumps to conclusions in many situations by heuristically adopting a specialized cliché if the percentage of its preconditions that are satisfied is significantly higher than the percentage of preconditions that are satisfied for any of the alternate specializations.

Heuristic classification trades the possibility of making a wrong decision for the added knowledge due to making some decision. Although it sometimes leads to problems that have to be fixed later, heuristic classification more than makes up for the problems it causes by fostering the rapid incorporation of large amounts of relevant information into a requirement. (The dependency information maintained by the Cake reasoning system keeps the cost of wrong guesses low by allowing easy retraction.)

H. Commands 10–11: Detecting an Anomaly

In Command 10, the analyst gives a brief definition of a *check-in* transaction by specifying that it is the inverse of check-out (i.e., it tracks the inverse state change). From this statement, the RA is able to derive quite complete information about check-in. Starting with Command 9, the amount of knowledge in the RKB has reached a critical mass that allows the RA to draw much more extensive conclusions from each statement of the analyst.

```
10 : (Inverse Check-In Check-Out)
"Check-In" Is-An-Instance-Of Action-Tracking-
  Operation.
...
```

Next, the analyst defines an *acquisition* transaction that tracks the addition of an entirely new book to the library. The RA is again able to determine significant amounts of information about this transaction and deduces that there is a problem with the requirement. From what has been said so far, the acquisition and check-in transactions appear identical. They both record the addition (the inverse of removal) of a book to the library. The RA has a built-in expectation that words should not be synonymous unless they are explicitly defined to be synonyms. This bias was introduced on the theory that an accidental alignment of this kind indicates that the analyst has likely left out some critical information or is using some word in an incorrect way.

```
11 : (Define Acquisition : Roles (: Records Add))
...
Conflict #1
Colliding-Definitions-Of "Check-In" And
  Acquisition
```

Fig. 7 shows three kinds of output created by the RA to help the analyst to understand and correct the anomaly. The RA first displays a proof of the problematic equality. An unfortunate aspect of this proof is that, like many machine-generated proofs, it is quite large and not particularly easy to understand (most of the proof is elided in the figure). (Much could and should be done to improve the way the RA presents such explanations. For instance, as described in [23], experimentation is underway on the possibility of using a graphical display of the proof tree.)

The RA then prints a list of premises underlying the conflict. The premises corresponding to statements made by the analyst are printed first, followed by information that comes from the cliché library. Although there can be a large number of these premises, it is often easier to grasp the problem by looking at the premise list than by looking at the proof. The conflict cannot be resolved without retracting or changing one of the premises.

It might be the case that one of the premises is totally wrong and should be discarded. However, more typically, all of the premises are at least partially correct. As shown at the bottom of Fig. 7, the RA can be helpful in this situation by providing suggestions for how a premise might be altered. This is done based on a number of heuristics. For example, if an equality premise specifies that something is equal to a particular value X, it might be the case that what is really desired is some value X' that is a close relative of X. In Premise 1 in Fig. 7, instead of remove, the analyst might have really meant some other operation on repositories. The only other operation on a simple repository is add, which the RA suggests as an alternative. Unfortunately,

```
Explanation:
 1.1 (Equal-Defs Check-In Acquisition) is True by Modus Ponens from
  2.1 Internal Premise:
      (Implies (And (Tracking-Operation Check-In)
                    (Tracking-Operation Acquisition)
                    (= Frame-I2 Frame-I3))
          (Equal-Defs Check-In Acquisition))
  2.2 (And (Tracking-Operation Check-In) (Tracking-Operation Acquisition)
           (= Frame-I2 Frame-I3)) is True by Conjunction from
   3.1 (= Frame-I2 Frame-I3) is True by Equality from
...
Premises:
1. (= (Records Check-Out) Remove)
2. (= (Records Acquisition) Add)
3. (Inverse Check-Out Check-In)
4. (Need (!Logic (Tracks University-Library-Database University-Library)))
5. (Default 7 (= (Objects Check-In)
                 (!The (?O) Such-That (= (Isbn ?O) $Input))))
...
Would you like suggestions for alternate premises?(N or #s) (1).
In the premise (= (Records Check-Out) Remove) you could substitute for the
word Remove the word Add-Repository.
Would you like to make a substitution?(Y or N) No.
```

Fig. 7. RA output describing the basis of Conflict 1.

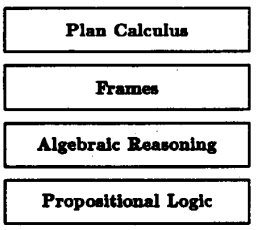


Fig. 8. The layered architecture of Cake.

changing remove to add is certainly not the correct way to fix the conflict since this would cause acquisition and check-out to have the same definition.

In the transcript being shown here, the analyst decides to ignore the conflict for the time being—it goes onto the list of pending issues. This delay in dealing with Conflict 1 illustrates that the RA always allows the analyst to be in control of what is dealt with and when. It is often a good idea to ignore something for a while until clarifying information comes in.

I. A Hybrid Reasoning System

The degree of automation that the RA can provide depends ultimately on its ability to reason about structured objects (clichés, requirements) and their properties. This rests on the capabilities of the Cake hybrid knowledge-representation and reasoning system [13], [32].

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

Fig. 8 shows the architecture of Cake. Note that Cake combines special-purpose representations, such as frames, with general-purpose logical and mathematical reasoning. Each layer of Cake builds on facilities provided by the more primitive layers below.

The propositional layer of Cake provides four principal facilities. First, it automatically performs simple one-step logical deductions (technically, unit propositional resolution). Placing tight limits on the kinds of deductions that are performed automatically is essential to avoid combinatorial explosions.

Second, the propositional layer supports a general mechanism for the pattern-directed invocation of demons. This can be used to simulate various kinds of quantified reasoning. Care is taken to ensure that the order independence of the reasoning is preserved even when demons are used.

Third, the propositional layer detects a certain class of shallow contradictions. (In conjunction with equality reasoning in the algebraic layer or Cake, this is used to detect the anomaly introduced by Command 11.) Importantly, contradictions are represented explicitly in such a way that reasoning can continue without having to resolve the contradiction immediately. This feature is motivated by a desire for the RA to support an evolutionary process wherein the analyst is always in control of the order of events.

Fourth, the propositional layer acts as a recording medium for dependencies (what is often called a truth-maintenance system; see [12]) and thus supports explanation (using the dependency records as a trace of the system's reasoning) and retraction (nonmonotonic reasoning). These facilities are motivated by the observation that when you delegate work to an assistant such as the RA, you must have accountability and the ability to recover from mistakes, in case it does not do what you expect. (Dependencies are used as the basis for the explanation in Fig. 7.)

The algebraic layer of Cake contains special-purpose decision procedures for equality (congruence closure), common algebraic properties of operators (such as commutativity, associativity, and transitivity), partial functions, and the algebra of sets. For example, the congruence closure algorithm determines whether or not terms are equal by substitution of equal subterms. The decision procedure for transitivity determines when elements of a binary relation follow by transitivity from other elements. The algebra of sets involves the theory of membership, subset, union, intersection, and complements. The algebraic layer of Cake also extends the basic propositional logic with the addition of typing (sorts) and limited quantificational facilities.

The frames layer of Cake supports the standard frame notions of inheritance, slots, and instantiation. It is used to represent the

clichés in the requirements cliché library. Frame inheritance is the primary organizing principle in the library. A notable feature of Cake's frame system is that constraints between the slots of a frame can be reasoned about in a general way. This helps support the RA's ability to incrementally acquire information in any order.

The Plan Calculus [31], [35] layer of Cake supports an internal representation for algorithms that is well suited for automated reasoning and other manipulations. This layer is not used by the RA, but figures prominently in other parts of the Programmer's Apprentice.

J. Command 12: Generating a Requirements Document

In Command 12, the analyst defines a report that the ULDB needs to support. The key content of the command is the logical expression that is used to fill the query role of the report. This describes the essential information content of the report and is therefore the heart of a high-level requirement for the report. Detailed information about layout and the like belongs in more detailed specifications, or perhaps in some appendix of the requirement.

```
12 : (Define Books-By-Topic : Report : Roles
       (: Data-In (!Decl ((?Topic Topic)))
       : Data-Out (!Decl ((?Titles (!Set-Of Title))))
       : Query (!Logic (= ?Titles (!The-Set-Of-All (?T Title) Such-That
                                    (!There-Exists (?B Book) Such-That
                                      (And (Mem ?B U1)
                                           (= ?T ?B.Title)
                                           (= ?Topic ?B.Topic))))))))
Book Has-A : Property Called "Topic".
...
```

An interesting aspect of Command 12 is that it refers to a previously undefined property of a book (its topic). The cliché library does not contain any mention of the word topic. However, the context "?B.Topic" indicates that topic can be applied to an arbitrary book. As a result, the RA assumes that topic must be a property of books in general.

The analyst can ask the RA to generate a requirements document at any time. Fig. 9 shows two excerpts from a requirements document generated after Command 12. The introduction summarizes the requirement. The report specification shown at the bottom of the figure is part of the system description section of the requirement. It reflects some simple deductions made by the RA based on Commands 1–12 and is generated based on the information in the tracking information system report cliché shown in Fig. 6.

K. Commands 13–14: Resolving a Conflict

In Command 13, the analyst defines a second report. Like Command 12, Command 13 mentions a word (borrower) not used previously. However, unlike topic, borrower appears in the cliché library. In particular, it is defined in conjunction with the lending repository cliché (see Fig. 5). The heuristic classification of the UL as not just a repository, but as a lending repository allows a meaning to be assigned to the relation borrower.

```
13 : (Define Books-By-Borrower : Report : Roles
       (: Data-In (!Decl ((?B Borrower)))
        : Data-Out (!Decl ((?Books (!Set-Of Book))))
        : Query (!Logic (= ?Books (!The-Set-Of-All
                                    (?A-Book Book) Such-That
                                    (Borrower ?A-Book U1 ?B))))))
U1 Is-An-Instance-Of Lending-Repository
...
```

In initial response to Conflict 1, the analyst asked for suggestions on how to alter Premise 1 (see Fig. 7). Since the analyst expressed an interest in this question, the RA uses Cake demons to continually monitor the premise and displays any new information regarding alternatives when it becomes known. Since lending repositories support two additional operations not supported by simple repositories, two new choices become available after Command 13.

```
Additional choices exist for substitution in
premises for Conflict #1.
In the premise (= (Records Check-Out) Remove) you
could substitute for the word Remove one of
(Add-Repository Return-Lending-Repository
Borrow-Lending-Repository).
Would you like to make a substitution?(N or #) 3.
...
Resolved-Conflict #1.
```

Prompted with the additional information above, the analyst decides to resolve Conflict 1. In particular, the analyst chooses to have check-out record *borrow* instead of *remove*. This causes numerous changes to both check-out and check-in and resolves the problem.

L. Commands 14–16: Support for Evolution

In Command 14, the analyst specifies an *unshelf* transaction that records the permanent removal of a book. Command 15 then

1 Introduction

The environment of the library-system requirement is the University-Library (UL). The UL is a library. A library is a repository for books.

The system being specified is the University-Library-Database (ULDB). The ULDB is a tracking-information system, which tracks the state of the (UL). Three transactions and one report are specified.

The library-system requirement is inconsistent and incomplete. There are five pending issues and one unresolved conflict.

4.2 Reports of ULDB

The reports of an information-system generate information about the data base without altering it. Reports are principally described by an input data signature, an output data signature, and a query that defines the functionality of the report.

4.2.1 Books-By-Topic

The tracking-information-system-report books-by-topic provides information about what the state of the UL is believed to be.

Data-in: ?topic of type topic.
Data-out: ?titles of type set-of title.
Query: ?titles = {?t:title | ∃ ?b:book (?b ∈ UL ∧ ?t = ?b.title ∧ ?topic = ?b.topic)}.
Accessed-information: titles of books and topics of books.
Target: UL.

The purpose slot is empty.

Fig. 9. Excerpts from a requirements document generated after Command 12.

specifies an *unshelf-all* transaction that records the removal of every book with a given ISBN number. In this command, the analyst overrides the default logical expression that specifies the objects being tracked.

```
14 : (Define Unshelf : Roles (: Records Remove))
...
15 : (Define Unshelf-All : Roles
       (: Records Remove
        : Objects (!The-Set-Of-All (?B) Such-That
          (= Isbn ?B) $Input))))
...
Conflict #2
Colliding-Definitions-Of Unshelf And Unshelf-All
```

As with Commands 9–11, the RA is able to deduce the basic features of both of these transactions. As with Command 11, the RA is able to detect a problem. Since the ISBN number is a unique identifier for books (see Command 8), there can never be two books with the same ISBN. As a result, the kind of removal recorded by unshelf-all is no different from the removal recorded by unshelf (see Fig. 10).

On seeing this conflict, the analyst realizes that the commands to this point reflect a type/token confusion between a physical copy of a book, and the logical notion of a book as a class of copies of books with some particular title, author, and ISBN.

(By this time it may seem that the errors made by the analyst in the transcript are a bit implausible. However, most errors look dumb after you have found them. In addition, the type/token confusion above was not merely fabricated as part of this transcript. Although the original problem statement in Fig. 2

```
Explanation:
 1.1 (Equal-Defs Unshelf Unshelf-All)
     is True by Modus Ponens 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)))
...
```

Fig. 10. RA output describing the basis of Conflict 2.

maintains a general distinction between a book and a copy of a book, several parts of it suffer from type/token confusion and use the words incorrectly, see [2].)

To fix the problem, a new word corresponding to a copy of a book must be introduced. In addition, something has to be done about the fact that only some uses of the word book in the commands above refer to the concept of a book. The rest refer to the concept of a copy of a book. The RA provides a special command *reformulate* that can assist with this kind of conceptual realignment.

```
16 : (Reformulate Book Copy-Of-Book)
<6> (Define Library : Ako Repository : Defaults
       (: Collection-Type Book))
= Reformulate=>?(Y or N) Yes.
<7> (Define Book : Ako Physical-Object
       : Member-Roles (Title Author Isbn))
= Reformulate=>?(Y or N) Yes.
```

```
1: (Find-Requirement Aircraft-System)
...
2: (Define Radar :Tracking-System
     :Roles (:Target (!Set-Of Aircraft)))
Radar Is-An-Instance-Of Multi-Target-Tracking-System.
...
3: (Fill-Roles Radar.Observation-Region
     :Max-Radius (Miles 50) :Max-Altitude (Feet 40000))
Radar Is-An-Instance-Of Multi-Target-Direct-Observation-Tracking-System.
Radar.Observation-Region Is-An-Instance-Of Cylindrical-Volume.
...
4: (Fill-Roles Radar.Observation.Error
     :Range-Error (Feet 20)
     :Theta-Error (Degrees 1)
     :Azimuth-Error (Degrees 1))
Radar.Observation.Error Is-An-Instance-Of Polar-Error.
...
5: (Define Radar-Display :Display-System
     :Roles (:Displays Radar.Observation))
...
6: (Fill-Roles Radar-Display.Display.Error
     :Latitude-Error (Feet 300)
     :Longitude-Error (Feet 300)
     :Altitude-Error (Feet 300))
Radar-Display.Display-Error Is-An-Instance-Of Cartesian-Error.
...
Conflict #1
Contradiction:
 (< (Altitude-Error (Error (Observation Radar))) (Feet 300)) is true
 (Not (< (Altitude-Error (Error (Observation Radar))) (Feet 300))) is true
...
```

Fig. 11. Excerpt from a second RA transcript.

```
<8> (Define Book.Isbn : Ako Integer : Cardinality
      Single : Unique-ID T)
= Reformulate=>?(Y or N) No.
<12> (Define Books-By-Topic : Report : Roles ···)
= Reformulate=>?(Y or N) No.
<13> (Define Books-By-Borrower : Report
      : Roles ···)
= Reformulate=>?(Y or N) Yes.
U1.Collection-Type Has-Value Copy-Of-Book.
Check-Out.Object-Type Has-Value Copy-Of-Book.
Check-In.Object-Type Has-Value Copy-Of-Book.
...
```

The reformulate command displays all the commands that contained the word in question and asks the analyst to specify where reformulation should occur. As shown by the analyst's responses above, libraries actually contain copies of books and the basic definition of a book really applies to copies of books. However, the ISBN is the unique identifier of a book, not a copy of a book. Further, the report books-by-topic does indeed want to refer to books. (Otherwise, you would see the same title ten times if the library contained ten copies of a relevant book.)

Once the analyst has identified the commands to be altered, the RA retracts the old commands and asserts the new ones. The changes that result are presented to the analyst for review. The truth maintenance facilities of Cake allow the retraction and assertion to be done in an incremental way so that the new state of the RKB can be determined with a minimum of rederivation. The RA ensures that the rederivation is carried out completely and that every aspect of the RKB is rechecked for consistency. This kind of comprehensive support for change is a particularly important feature of the RA.

In the interest of brevity, the transcript is truncated at this point. The complete transcript is shown in [37], which also contains the complete requirements document produced after the last command is entered. The analyst has to enter 20 additional commands to communicate the full content of Fig. 2 to the RA. The final requirements document generated by the RA contains 24 pages of information derived from the 36 commands in conjunction with the library of requirements clichés.

M. A Second Example Transcript

The range of applicability of the RA is illustrated by Fig. 11, which shows the first six commands of a transcript of the RA being used to create a requirement for part of an air traffic control system. It is excerpted from a much larger transcript presented in [23]. The key feature of the transcript is that while it uses a few clichés not used in the library transcript, it uses several of the same clichés and exactly the same reasoning, disambiguation, and conflict detection mechanisms.

As an example of a cliché used in both transcripts, consider that the radar defined in Command 2 of Fig. 11 is a tracking system. The radar tracks many targets rather than just one, operates by direct observation within a restricted volume rather than by indirect observation, and is not an information system. Nevertheless, the radar shares with the ULDB the key characteristic of tracking the state of other objects.

As an example of a different cliché, consider that the radar-display defined in Command 5 is a *display system*—a system that maintains a continually updated display of some quantity. In addition, the commands in the figure make use of a number of clichés involving different kinds of volumes and errors. For instance, radar observes airplanes within a limited cylindrical volume and has an observational error that is naturally characterizable in polar coordinates.

In Command 6, the analyst specifies that the positions of airplanes must be displayed with an error of no more than 100

feet. This immediately leads to a conflict. Unlike the examples of unexpected synonymy in the library transcript above, the conflict in Fig. 11 is a direct contradiction. One of the constraints on the display system cliché specifies that the accuracy of the display is inherently limited by the accuracy of the data. Here, it is impossible to achieve the required altitude accuracy in the display given the long range and large azimuth error of the radar. One either has to use an improved radar or settle for less display accuracy.

IV. Conclusion

Research on the RA addresses an important gap in much of the work on tools to support requirements analysis. Whereas the requirements acquisition process almost inevitably begins with a vague and informal statement of what is desired, most requirements analysis tools need some sort of relatively formal requirement statement as their input. The RA attacks head-on the problem of fleshing out and cleaning up a vague and informal requirement statement.

Although research on the RA is still in progress, the current version of the system already demonstrates a number of capabilities that can be used to bridge the gap between informal and formal specifications. The analyst is responsible for communicating with the end-user and entering the core information in the requirement. However, the analyst is able to talk in terms of high-level words leaving the task of disambiguating these words to the RA, because the RA has a cliché library that allows it to infer large amounts of information from the analyst's statements. The RA continually checks the requirement for consistency (searching for contradictions) and completeness (based on expectations set up by various clichés). The RA provides specific support for evolutionary modification of the requirement, using the facilities of the underlying reasoning system to ensure that each change is carried out in a consistent way throughout the requirement. At any time, the RA can print out a requirements document reflecting the current state of the requirements knowledge base.

There are three basic directions in which work on the RA can profitably proceed in the future. First, basic research needs to continue on ways for the RA to provide greater support for requirements acquisition. In particular, the expressive power of cliché frames and the RA's reasoning capabilities both need to be increased. In addition, the horizons of the RA should be expanded to encompass more information. For instance, the RA currently only captures the final decisions of the end-user about what should be in the requirement. In the interest of better support for evolution, it would be good to go beyond this and capture the outlines of the reasoning behind these decisions (see [30]).

Second, without the need for any further research, many aspects of the RA can be directly incorporated in practical tools. An example of this is the Knowledge-Based Requirements Assistant (KBRA) system [9] constructed by Sanders Assoc. with the assistance of Charles Rich of the Programmer's Apprentice group. The metaphor of the KBRA is that it manages an engineer's notebook. This notebook contains information in four forms: text (which has some simple lexical analysis applied to it), context diagrams, state transition diagrams, and spreadsheets (where constraints can be defined and propagated). All four kinds of information are stored in a central database, which can be used as the basis for an automatically generated requirements document. A library of reusable requirements components can be used in a cut and paste style. Reasoning processes in the KBRA include inheritance, classification through the use of special-purpose decision tables attached to library components, and limited constraint propagation. The key difference between the RA and the KBRA is that the input to the KBRA is primarily represented as uninterpreted text and diagrams. In contrast, all of the input to the RA is represented in a form where it can be effectively reasoned about. In addition, the RA is capable of significantly more flexible and complex reasoning than the KBRA.

Third, the RA itself has reached the point where it would be appropriate to construct a full-scale prototype that could be tested on full-scale requirements. This would require the construction of a realistic interface along the lines illustrated in Fig. 4 and a full-scale cliché library for some particular domain. Experimentation with real requirements could then be used to quantify the benefits of the RA due to more rapid construction of requirements through the reuse of requirements clichés and the discovery of problems in a requirement earlier in the software engineering process.

Acknowledgment

In addition to the authors, several people have made major contributions to the development of the RA. In particular, C. Rich has contributed to every aspect of the RA and KBRA from their earliest inceptions. C. Rich and Y. Feldman implemented the Cake reasoning system on which the RA is based. In the course of working on the air traffic control example, P. Lefelhocz uncovered a number of problems with the RA and contributed to their solution.

References

[1] G. Arango and P. Freeman, "Modeling knowledge for software development," in *Proc. 3rd Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Aug. 1985, pp. 63–66.

[2] R. Babb *et al.*, "Workshop on models and languages for software specification and design," *Computer,* vol. 18, no. 3, pp. 103–108, Mar. 1985.

[3] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software,* vol. 1, no. 1, pp. 75–88, Jan. 1984.

[4] R. Balzer, N. Goldman, and D. Wile, "Informality in program specifications," *IEEE Trans. Software Eng.,* vol. 4, no. 2, pp. 94–103, Mar. 1978.

[5] D. R. Barstow, "Domain-specific automatic programming," *IEEE Trans. Software Eng.,* vol. 11, no. 11, pp. 1321–1336, Nov. 1985.

[6] J. S. Bennett, "A knowledge-based system for acquiring the conceptual structure of a diagnostic expert system," *J. Automated Reasoning,* vol. 1, no. 1, pp. 49–74, 1985.

[7] A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge representation as the basis for requirements specifications," *Computer,* vol. 18, no. 4, pp. 82–90, Apr. 1985.

[8] A. Crawford, "Joint application design: A new way to design systems," in *Guide Int. Proc.,* Guide Int. Corp., 1982.

[9] A. Czuchry and D. Harris, "KBRA: A new paradigm for requirements engineering," *IEEE Expert,* vol. 3, no. 4, pp. 21–35, Winter 1988.

[10] R. Davis, Interactive transfer of expertise: Acquisition of new inference rules," *Artificial Intell.,* vol. 12, pp. 121–157, 1979.

[11] J. Diederich, I. Ruhmann, and M. Maym, "KRITON: A knowledge acquisition tool for expert systems," *Int. J. Man–Machine Studies,* vol. 26, no. 1, pp. 29–40, Jan. 1987.

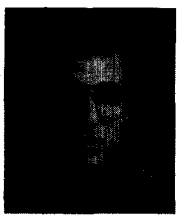
[12] J. Doyle, "A truth maintenance system," *Artificial Intell.,* vol. 12, pp. 231–272, 1979.

[13] Y. A. Feldman and C. Rich, "Principles of knowledge representation and reasoning in the FRAPPE system," in *Proc. Sixth Israeli Symp. Artificial Intelligence,* Dec. 1989.

[14] S. Fickas, "Automating analysis: An example," in *Proc. 4th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Apr. 1987, pp. 58–67.

[15] S. Fickas and P. Nagarajan, "Critiquing software specifications," *IEEE Software,* vol. 5, no. 6, pp. 37–47, Nov. 1988.

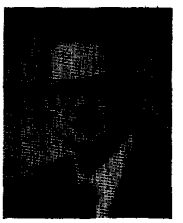
[16] N. Haas and G. G. Hendrix, "Learning by being told: Acquiring knowledge for information management," in *Machine Learning an Artificial Intelligence Approach,* R. Michalski *et al.,* Eds., 1983, pp. 405–421.
[17] M. T. Harandi and F. H. Young, "Template based specification and design," in *Proc. 3rd Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Aug. 1985, pp. 94–97.
[18] W. Johnson and M. Feather, "Representing evolution transformations," in *Proc. 11th Int. Joint Conf. Artificial Intelligence, Workshop Automating Software Design,* Aug. 1989, pp. 125–131.
[19] T. S. Kaczmarek, R. Bates, and G. Robins, "Recent developments in NIKL," in *Proc. 6th Nat. Conf. Artificial Intelligence,* Aug. 1986, pp. 978–985.
[20] R. A. Kemmerer, "Testing formal specifications to detect design errors," *IEEE Trans. Software Eng.,* vol. 11, no. 1, pp. 32–43, Jan. 1985.
[21] J. Z. Lavi, "Improving the embedded computer systems software process using a generic model," in *Proc. 3rd Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Aug. 1985, pp. 127–129.
[22] S. Lee and S. Sluizer, "SXL: An executable specification language," in *Proc. 4th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Apr. 1987, pp. 231–235.
[23] P. M. Lefelhocz, "An experiment in knowledge acquisition for software requirements," MIT Artificial Intelligence Lab., Rep. MIT/AI/WP-330, May 1990.
[24] J. C. Leite, "Viewpoint analysis: A case study," in *Proc. 5th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, May 1989, pp. 111–119.
[25] M. L. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision,* P. H. Winston, Ed. New York: McGraw-Hill, 1975.
[26] ——, *Society of Mind.* New York: Simon & Schuster, 1987.
[27] M. Musem, L. Fagan, D. Combs, and E. Shortliffe, "Use of a domain model to drive an interactive knowledge-editing tool," *Int. J. Man–Machine Studies,* vol. 26, no. 1, pp. 105–125, Jan. 1987.
[28] J. M. Neighbors, "The Draco approach to constructing software from reusable components," *IEEE Trans. Software Eng.,* vol. SE-10, no. 5, pp. 564–574, Sept. 1984.
[29] P. Politakis and S. Weiss, "Using empirical analysis to refine expert system knowledge bases," *Artificial Intell.,* vol. 22, no. 1, pp. 23–48, 1984.
[30] C. Potts, "Requirements analysis, domain knowledge, and design," MCC, Tech. Rep. STP-313-88, Feb. 1989.
[31] C. Rich, "A formal representation for plans in the programmer's apprentice," in *Proc. 7th Int. Joint Conf. Artificial Intelligence,* Aug. 1981, pp. 1044–1052.
[32] ——, "The layered architecture of a system for reasoning about programs," in *Proc. 9th Int. Joint Conf. Artificial Intelligence,* Aug. 1985, pp. 540–546.
[33] C. Rich and R. C. Waters, "The programmer's apprentice: A program design scenario," MIT Artificial Intelligence Lab., Rep. MIT/AIM-933A, Nov. 1987.
[34] ——, "The programmer's apprentice: A research overview," *Computer,* vol. 21, no. 11, pp. 10–25, Nov. 1988.
[35] ——, *The Programmer's Apprentice.* Reading, MA: Addison-Wesley, and Baltimore, MD: ACM Press, 1990.
[36] C. Rich, R. C. Waters, and H. B. Reubenstein, "Toward a requirements apprentice," in *Proc. 4th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Apr. 1987, pp. 79–86.
[37] H. B. Reubenstein, "Automated acquisition of evolving informal descriptions," Ph.D. dissertation, MIT Artificial Intelligence Lab., Rep. MIT/AI/TR-1205, June 1990.
[38] H. B. Reubenstein and R. C. Waters, "The requirements apprentice: An initial scenario," in *Proc. 5th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, May 1989, pp. 211–218.
[39] W. Robinson, "Integrating multiple specifications using domain goals," in *Proc. 5th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, May 1989, pp. 219–226.
[40] R. Schank, "A conceptual dependency representation for a computer-oriented semantics," Stanford Univ., Tech. Rep. AIM-83, 1969.
[41] J. G. Schmolze and T. A. Lipkis, "Classification in the KL-ONE knowledge representation system," in *Proc. 8th Int. Joint Conf. Artificial Intelligence,* Aug. 1983, pp. 330–332.
[42] G. J. Susman, "The virtuous nature of bugs," in *Proc. Conf. Artificial Intelligence and the Simulation of Behavior,* Univ. Sussex, England, July 1974.
[43] W. Swartout, "The GIST behavior explainer," in *Proc. 3rd Nat. Conf. Artificial Intelligence,* Aug. 1983, pp. 402–407.
[44] D. Teichroew and E. Hershey, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.,* vol. 3, no. 1, pp. 41–48, Jan. 1977.
[45] R. B. Terwilliger, M. J. Maybee, and L. J. Osterweil, "An example of formal specification as an aid to design and development," in *Proc. 5th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, May 1989, pp. 266–272.
[46] R. C. Waters, "The programmers' apprentice: A session with KBEmacs," *IEEE Trans Software Eng.,* vol. SE-11, no. 11, pp. 1296–1320, Nov. 1985.
[47] J. M. Wing, "A Larch specification of the library problem," in *Proc. 4th Int. Workshop Software Specification and Design.* Washington, DC: IEEE Computer Society Press, Apr. 1987, pp. 34–41.
[48] S. Wrobel, "Design goals for sloppy modeling systems," *Int. J. Man–Machine Studies,* vol. 29, no. 4, pp. 461–482, Oct. 1988.
[49] "Using the WISDM team method to define system requirements," Western Institute Software Engineering, 1986.



Howard B. Reubenstein (S'85–M'90) received the S.B. and S.M. degrees in computer science in 1985, and the Ph.D. degree in computer science in 1990 for his work on the Requirements Apprentice, all from the Massachusetts Institute of Technology, Cambridge.

He is currently a member of technical staff at MITRE. His research interests include automation of the software development process, requirements acquisition, design replay, and knowledge representation and reasoning systems.

Dr. Reubenstein is a member of the Association for Computing Machinery, the American Association for Artificial Intelligence, and Sigma Xi, as well as the IEEE Computer Society.



Richard C. Waters (M'78–SM'86) received the Ph.D. degree in computer science with a minor in linguistics from the Massachusetts Institute of Technology, Cambridge, in 1978.

Since then he has worked in the MIT Artificial Intelligence Laboratory where he is currently a principal research scientist. He is co-principal investigator of the Programmer's Apprentice project. This project, of which the requirements apprentice is a part, is developing a system that can assist programmers to develop and maintain programs. He is an active consultant in the areas of artificial intelligence and software engineering.

Dr. Waters is currently Program Co-chair for the Enabling Technology and Systems area of AAAI-91, and a member of the program committees of the First International Conference on Requirements Engineering and the Fourth International Symposium on Applications of AI.