MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 290 September 1986

A Requirements Analyst's Apprentice: A Proposal

by Howard Reubenstein

Abstract

The Requirements Analyst's APprentice (RAAP) partially automates the modeling process involved in creating a software requirement. It uses knowledge of the specific domain and general experience regarding software requirements to guide decisions made in the construction of a requirement. RAAP assists the analyst by maintaining consistency, detecting redundancy of description, and analyzing completeness relative to a known body of requirements experience. RAAP is a tool to be used by an analyst in his dealings with the customer. It helps him translate the customer's informal ideas into a requirements knowledge base. RAAP will have the ability to present its internal representation of the requirement in document form. Document-based requirements analysis is the state of the art. A computer-based, knowledge-based analysis system can provide improvement in quality, efficiency and maintainability over document-based requirements analysis and thus advance the state of the art towards automatic programming. RAAP takes a new approach to automating software development by concentrating on the modeling process involved in system construction (as opposed to the model translation process). By supporting the intelligent creation of perspicuous models, it is hoped that flaws will become self revealing and the quality of software can be improved. Assistance is proved for the creation of "correct" models and for the analysis of the implications of modeling decisions.

Table of Contents

Acknowledgments

I would like to thank Dick Waters and Chuck Rich for reading drafts of this proposal and providing many useful suggestions on form and content. I would also like to thank Dick for many long discussions and for keeping me headed in the right direction.

1. Overview

The Requirements Analyst APprentice (RAAP) is a system which aids an analyst in the modeling process used to create a system requirements definition. During the creation and enhancement of a requirement, RAAP provides for the integration of new information, the maintenance of consistency, and reuse of previous analyses. The end product of the interaction with RAAP is, among other things, the automatic generation of a readable requirements document.

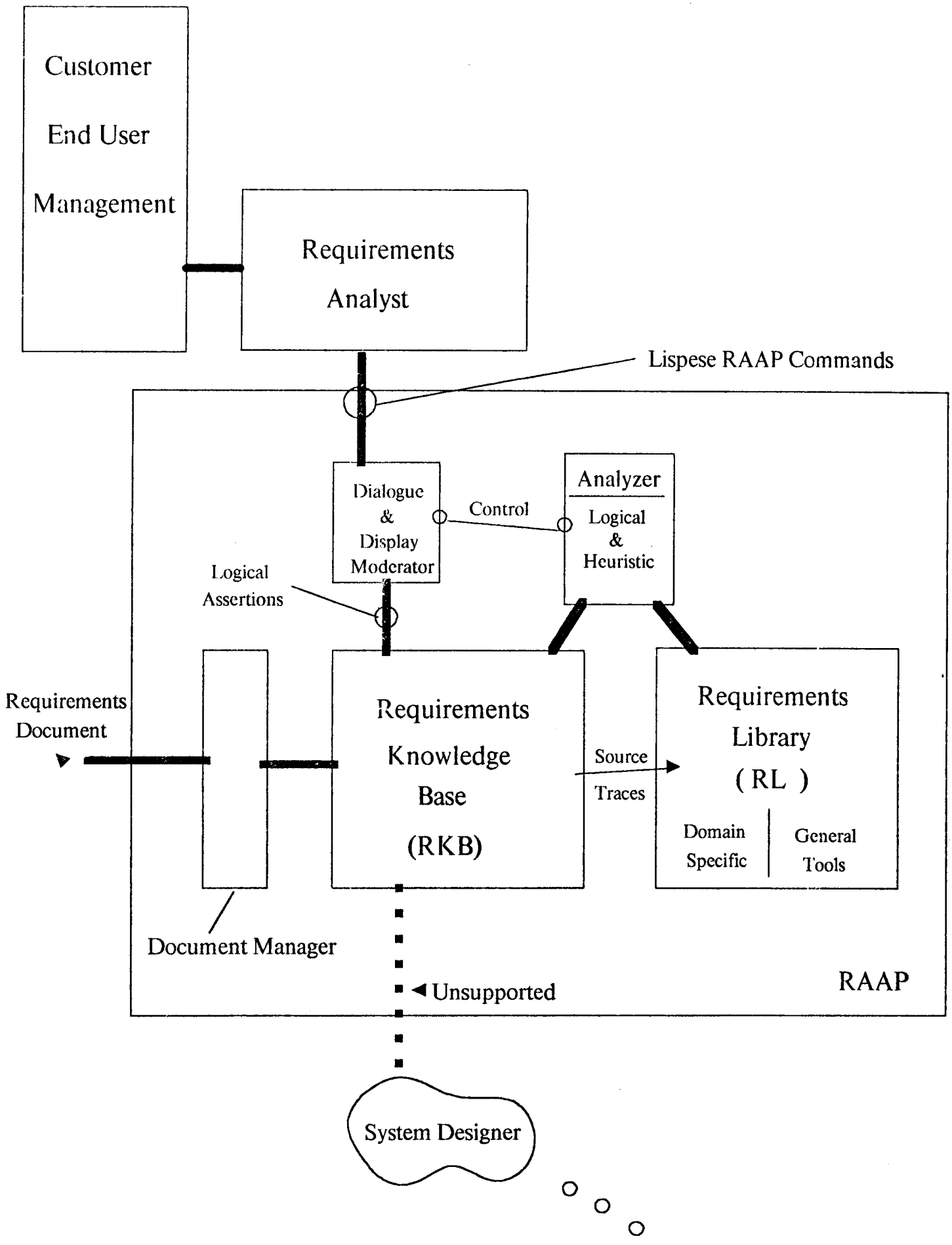
(See next page)

Figure 1-1: Top Level System Picture

Figure 1-1 shows that the analyst uses RAAP as an assistant, as a tool, in his interaction with the customer. Input to RAAP is from a set of formal commands that support creation of the requirement in the Requirements Knowledge Base (RKB). The analyst's input may also include references to reusable aggregations of information in the Requirements Library (RL). Direct interaction between RAAP and the customer is specifically excluded since providing such support would require solving problems in natural language processing and user modeling that are orthogonal to the issues of knowledge integration and deductive power that are addressed here. The interaction with the system designer is also unsupported in the sense that RAAP will not include special purpose tools to facilitate design. This would be a natural augmentation, but as shall be seen, the process of requirements analysis provides quite an adequate source of complex problems. RAAP will be useful to the designer in that it will provide a knowledge base of information that can be accessed, cross-referenced, and analyzed from a variety of viewpoints, i.e. the designer will be able to more easily get at information needed to support his job.

Data Stores

The RKB is a repository of all the relevant information about the current requirement for the system under analysis. In current software engineering practice, the transfer of information from the customer to the analyst results in the creation of a requirements document. In RAAP this mode of interaction is replaced by the creation of a requirements knowledge base, the RKB. Still, it is necessary to support the creation of a requirements document for the purposes of review and dissemination. The Document Manager will provide for the automatic creation of such a document


Customer
End User
Management
Requirements
Analyst
Lispese RAAP Commands
Dialogue
&
Display
Moderator
Control
Analyzer
Logical
&
Heuristic
Logical
Assertions
Requirements
Document
Requirements
Knowledge
Base
(RKB)
Source
Traces
Requirements
Library
(RL)
Domain
Specific
General
Tools
Document Manager
Unsupported
RAAP
System Designer

from the RKB. Eventually the role of the requirements document might be lessened by the integration of RAAP into a fully automated programmer's assistant. There is promise that a computer-based RKB can replace the systems analyst's and builder's need for a requirements document. The computer-based system can deal with small perturbations automatically, perform requirements analyses that are impractical using a paper based system, and track dependencies. However, given RAAP's current isolation in the software lifecycle a requirements document is the most concrete evidence that RAAP is performing correctly. In any case it seems likely that management will need a summary document in order to evaluate less codifiable system concerns and thus support for document creation is essential.

The RL contains reusable semantic attachments for terms associated with the problem domain and the requirements modeling process. The RL distinguishes between domain specific knowledge and general requirements information. The general requirements information can be thought of as modeling tools that are portable across domains. These must be instantiated with values and properties specific to objects in the domain of interest which can be provided by the domain specific component. This division is not strict. There can be domain specific reusable modeling techniques and general reusable descriptions of specific objects. Examples of each type of cliche and their use will be presented in the scenario in Section 2. Strictly speaking the main difference between general requirements information and domain specific knowledge is that RAAP will be "delivered" with an empty domain specific knowledge box, but the general information box will contain tools, perhaps with special supporting code, to be (re)used in any analysis.

The Analyzer maintains consistency in the RKB. It also aids the Dialogue Manager in finding redundancy in user input. Though redundant description is not incorrect, it may be indicative of misconception on the part of the analyst as to the nature of the model he has described. The Analyzer also interacts with the RL, using it to set a context for completeness checking. The RL also drives the logical inference of the Analyzer by providing additional context specific reasoning methods and reusable detailed object descriptions. The function of the Analyzer will be described in more detail later and demonstrated in the scenario.

2. A Scenario

The domain of the scenario is a library information system. The understanding RAAP brings to this problem is described in the next section by informal definitions of some useful terms, referred to as cliche definitions. The scenario is annotated, where appropriate, with descriptions of additional knowledge assumed to support the described behavior. The cliches presented describe an information system in detail and some library specific concepts. The knowledge of how to integrate these descriptions is missing. The scenario demonstrates the integration of these descriptions by the instantiation of a general requirements cliche, the information-system, with domain specific knowledge.

2.1 The Knowledge RAAP Starts With

General Requirements Cliches

A simple cliche is **Overlapping-partition**. This has three roles the **Universe-set**, **Set-a**, and **Set-b**. The definition includes the knowledge that Set-a union Set-b equals the Universe-set and that Set-a intersect Set-b is not empty. A cliche represents a shared, reusable aggregate of information as a set of roles structured by constraints.

The cliche central to the scenario is the **Information-system** (IS). An information system stores and organizes data, provides access to it, and provides tools to change it. The concept of an information system is a top level system structuring tool available to requirements analysts. It is useful enough to motivate projects, such as the User Software Engineering Methodology [36], to support the development of exactly this class of systems.

An Information-system has the following multi-valued roles: **Relations**, **Operations**, **Reports**, and **Data-constraints**. The definition of each of the roles follows:

Data-constraints are predicates that must be true of data stored in the Information-system. RAAP understands that these are logical assertions and can analyze their implications.

Figure 2-1 shows an instance of a **Relation**. A Relation holds the data "in" the system. A Relation's **Schema** consists of a list of the table's column headers and the **Name** in the corner of the table. The Relation also has a Name equal to the Schema's Name. A **Tuple** is one row of the table. The Relation also has **Unique-keys** which are specified by either the name of a **Field** (the column headers in the figure), e.g. Id #, or by a unique key function, e.g. the ordered pair of the "Name" and "Dept" Fields.

(See next page)

Figure 2-1: The Employees Relation

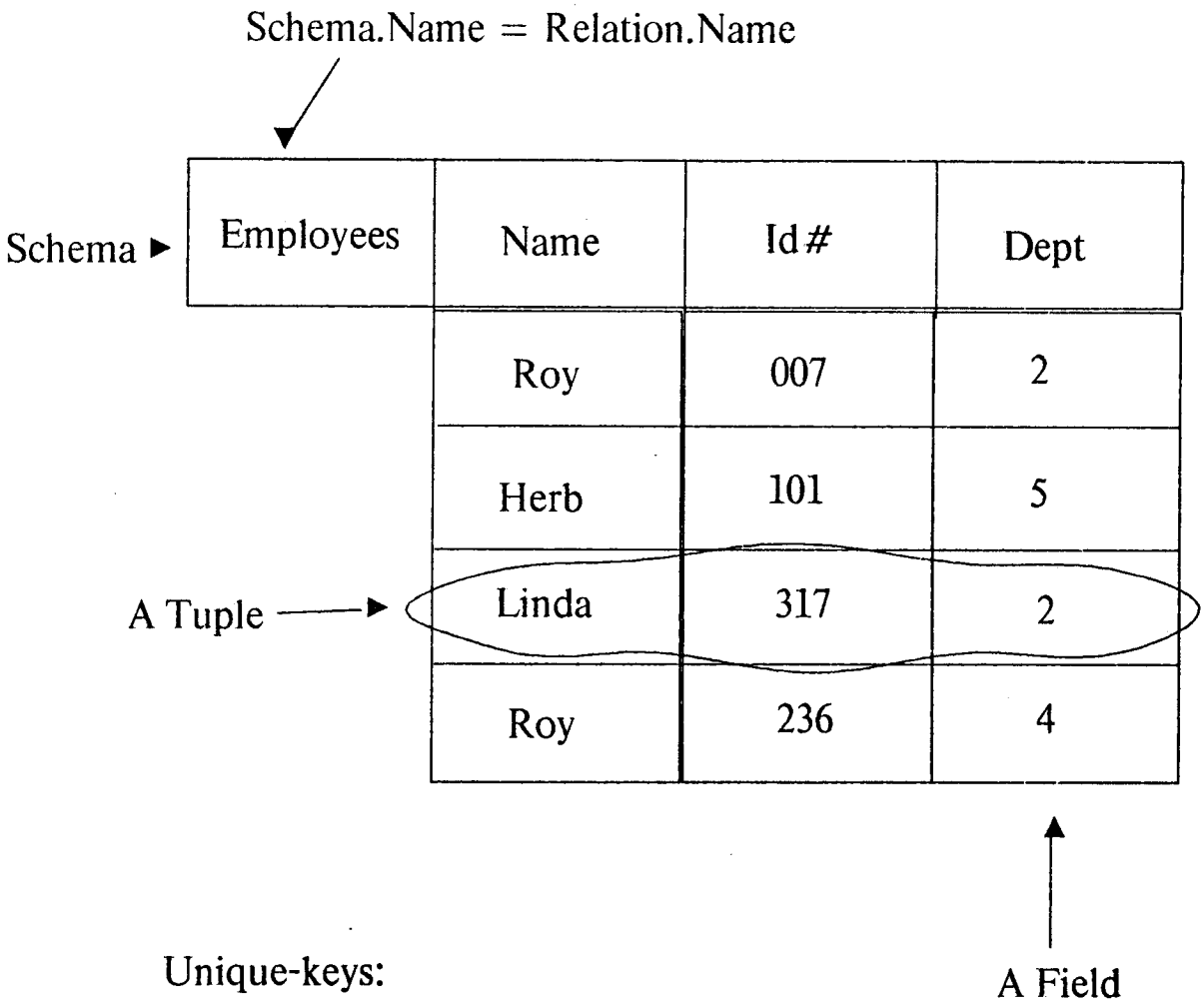
The **Operations** role has four multi-valued sub-roles: **Query**, **Remove**, **Add**, and **Alter**, which store all the data associated with the Operations role. Queries return a set of tuples based on some selection predicate. Remove and Add Operations work on entire single tuples. Alter Operations work on a particular Field in a Tuple.

"Executive" operations such as adding and removing relations are not modeled here. In the scenario, the analyst specifies what relations are stored in the Information-system. In this case the notion of an Information-system is construed to be one where the classification of the type of information stored in the system is not something alterable by the users of the system, rather it is determined by the requirements analyst.

A detailed description of the **Remove** Operation includes the following roles:

- Relation-name
- IS-name
- User-name
- Selection-predicate
- Predicate-data

The roles, in this case, become input parameters in an instantiation of a Remove Operation as a *basic RAAP object* of type *operation*. The definition of an operation called Remove-generic-thing as a Remove Operation is as follows:



Unique-keys:

1. Id#
2. (Name,Dept)

Remove-generic-thing is-a *operation*

Begin

Input = Relation-name, IS-name, User-name, Selection-predicate, Predicate-data

Preconditions = the tuples referenced by the Selection-predicate applied to the Predicate-data are in the Relation named by Relation-name in the Information-system named by IS-name.

Post-conditions = the referenced tuples are not in the Relation; they were removed by the User named by User-name.

Output = None

Exceptions =

- Condition: Not-In, Action: Nop
- Condition: Unauthorized, Action: Bad-attempt(User-name)

End

The components of the above definition comprise RAAP's description of system operations. Each can be used to support various inference processes. For example, pre and post-conditions are stored in a logical assertion form and could be propagated to determine the effect and validity of a sequence of operations.

The fourth role of the Information-system cliche, **Reports**, provides for the presentation of data in the Information-system. The template for a report definition is:

- **Name** - a handle that permits invocation of the report
- **Description** - text that explains the purpose of the report perhaps printed under the header
- The definition of what data is to be computed:
  - **Input** - data used in the determination of the data to be presented
  - **Using-relations** - relations to be accessed in computing the output
  - **Using-fields** - fields in the above relations to be used
  - **Compute-data** - description of the data to be computed
  - **Data-computation-definition** - how to compute the Compute-data

The definition of what is to be printed and how:

- **Print-header** - the text to be printed at the top of the report
- **Output-sort** - a function specifying how the report is to be sorted

- **Print-data** - which of the Compute-data to print

- **Boundary-case** - characterization of what it means for the Compute-data to be empty

General requirements cliches will be part of the requirements analysis tool kit that RAAP provides. There are likely to be relatively few of them compared to the amount of possible domain specific cliches. They should be as free of design decisions as possible. Some other candidates are: Formula-computation-system, Message-passing-object-system, User-input-interaction, User-output-interaction, Events, Object-identification, and Object-initialization.

Domain Cliches

Two examples of domain specific cliches that support behavior in the scenario are given below. The definition of book is a bit of domain specific knowledge used to provide information about the data stored in the information system. The definition of library fines is a small example of the sort of problem specific information that RAAP can use to support domain specific dependency analysis.

A **Book** is an object. Its known intrinsic properties are: **Title**, **Author**, and **ISBN** (an identification number). A Book is not yet understood in the context of the functionality of a library and thus its definition does not include the operationally derived properties of borrower and due-date. Neither is its use as an object in an Information-system understood, thus it does not yet have a Unique-key.

Fines is a role of a Library cliche. It has a **Fine-function** role that computes the **Fine-amount** based on the input of the **Current-date** and the Book's **Due-date**. The **Billing-function** takes as input the Book's **Borrower**, the Book, and the **Fine-amount**. Its definition is a choice between **Bill-by-mail**, whose definition requires knowing the User's address, and **Bill-by-daemon**, i.e. when the Borrower next uses the library. The structure of the Fines cliche also joins the two roles, Fine-function and Billing-function, in a manner describing the invocation of the fine determination and collection process. (Note, concepts like Current-date may also be defined in the RL. If not they are treated as uninterpreted primitives.)

2.2 The Library Example

The following scenario is based upon a library database example published in [12]. A reproduction of the published problem follows on page 8 and the scenario follows it. This problem is something of a benchmark by virtue of the fact that 18 tools were evaluated against it at the Second Workshop on Models and Languages for Software Specification and Design [39].

The library domain is familiar enough that people will have associations for the terms used in the

description. This is something of a disadvantage in that the analyst may unconsciously fill in gaps in the description with his own semi-expert domain knowledge. On the other hand the requirements analysis process is interactive and the goal of RAAP is not to process some preformed "correct" requirement, rather it is to assist in the definition of what ultimately becomes an acceptable requirement. In this sense the library requirement should be thought of as a guide to the analyst. The familiarity of the domain allows him to play both analyst and customer. The scenario will demonstrate features of RAAP that help keep the analyst on the right track. In the sense that the requirement is underdescribed it is important that the source of missing information be identified, i.e. from cliches or from the analyst's head. No tool can be expected to pull missing facts out of nowhere. In the scenario there will be examples of deduction of missing information, extraction of information from a cliche, and plausible guesses at missing information.

The Scenario Text

The RAAP user interface is currently unspecified. It is assumed that the analyst will be familiar with the commonly used dozen or so RAAP commands and their function, and that he will also be familiar with the names and roles of cliches. Analyst input will take the form of S-expressions. RAAP output will be preceded by a "RAAP>" prompt. Cliche and role names will begin with capital letters. User provided names and values will begin in lower case. RAAP commands and keywords will also begin in upper case.

Informal specification distributed in tools session

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 checkout.
- G2 - No copy of a book may be both checked out and available for checkout.
- 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.

Figure 2-2: The Library Example

For the most part the scenario will follow the order and content of Figure 2-2. Some additions have been made to demonstrate capabilities not exercised by the published example.

To realize how underdescribed the example really is, and thus the added power of reasoning with cliches, try the following experiment. Forget the semantics you have associated with the words add, added and related forms. Wherever you see one of these words in the description use the semantics you have associated with any of the other three operations check out, remove, or return. Do you find any glaring problems? In fact, the example is close to silent about the structure of anything except the notion of check out. The only cues are words like "to" and "from" (if you replace add with remove you may be bothered by the resulting phrasing of "remove to"), and "currently" and the the use of the present tense (the time notions associated with check out and return our different, a return action is basically an instant whereas a check out period is an interval). Of course even these notions require the analyst to make his everyday attachments to them. With respect to pure structure, i.e. encode all the words in the example, draw out the logical structure of the resulting description (retaining the knowledge of what parts of speech the words are since this is knowledge unintentionally garbled in encoding), and try to match the description to anything particular in the real world, the text is virtually vacuous.

2.3 Scenario

(Define ULDB Instance-of Information-system)

ULDB stands for university library database. This command attaches the name ULDB to an instantiation of the Information-system cliche which includes constraints on the various, as yet unfilled, roles.

(Fill-Role Relation.Object With Book)

```
RAAP> Relation.Schema = (Book (Title Author ISBN))
```

A Schema was defined as the column headers (Fields) and the name of a relation. Part of the relation cliche includes knowledge that relations can model objects. When the Relation.Object field is filled in, the Schema field is derived from the name of the object and a list of its intrinsic properties.

The "pathname" Relation.Object is relative to the current context. It expands to Information-system.Relation.Object. The dot notation denotes property or role. In general, RAAP first attempts to resolve references in the current context and then looks back in the stack for the most recently entered context where the reference can be resolved. Since this is the only statement to be made in the Relation context, the analyst has not bothered to explicitly switch contexts.

With respect to completion analysis, the definition of the Book Relation gives reason to assume that definitions of appropriate operations for removing and adding books will follow.

```
(Fill-Slot Book.Unique-key With ISBN)

(Fill-Slot Book.Unique-key With (Title Cross Author))
```

Unique-key is a multi-valued slot and thus both statements store values there. The notion of Book stored in RAAP's knowledge base upon entry into the scenario is that of the intrinsic properties of a book. Books have Titles, Authors, and ISBN. If you look at the book you can find these. Books do not yet have due dates or borrowers. These are library specific concepts. The notion of a Unique-key is not intrinsic to a book rather it is intrinsic to objects stored in an Information-system. The analyst understands the Information-system cliche and fills in this important derived slot.

It may seem contrived that RAAP does not understand books in the context of libraries. It seems more plausible that RAAP does not understand the particular object book in the context of information systems. However, the interesting characteristic of this situation is the use of a cluster of knowledge in a new situation and the transfer of knowledge that takes place.

RAAP might automatically deduce the value for the Unique-key slot if the names of the intrinsic slots in Books were connected to generic description cliches. If ISBN were more than four letters of text, i.e. attached to the notion of "unique 10 digit integer that encodes information about the book", then in a search for Unique-key RAAP might find the ISBN.

```
(Within Overlapping-partition
          (Fill-Roles Universe With Users
                      Set-a With normal-borrowers
                      Set-b With library-staff))

(Define Synonyms normal-borrowers borrowers
                 library-staff staff)
```

The current focus is the Information-system cliche. The "Within Overlapping-partition" temporarily changes the focus. The statement defines two subclasses of Users that are overlapping and comprise the entire User population. The names normal-borrowers and library-staff are uninterpreted.

The cliche Overlapping-partition was described previously. It is a shorthand for two statements A union B = Universe and A intersect B is Non-empty. The fact that A and B are subsets of the Universe follows from these two statements; as does the fact that neither A nor B are empty. Cliches provide a shorthand for relatively small bodies of information (such as the half dozen statements above) or for larger bodies of information (such as the Information-system and its defined structure).

```
(Push-Context Information-system.Operations)
```

This command pushes a new context onto the stack. In this context, the analyst will describe the operations that provide access to the Book relation. The cliche guides the expectations for the definition of a set of operations. A certain minimal set of operations is expected to be defined, i.e. a Remove, Add, and Query operation. Inputs to these operations can be resolved relative to the current context.

```
(Define Check-out Name-of Remove
                        (Fill-Role Relation-name With Book))
```

Check-out is the name of a Remove function with the roles filled in as follows:

- Relation-name = Book (explicitly given)
- IS-name = ULDB (deduced by being the only known IS)
- Selection-predicate = Choice from Book.Unique-keys (deduced from the default attachment to the Relation's Unique-keys)

User-name and Predicate-data remain inputs that must be provided to Check-out. This is the transaction specific information. The deduced values instantiate the generic "remove object from a collection" operation. The definition stands as follows:

```
check-out(User,Book) =
PRE:        member(Book,ULDB)
POST:       Not member(Book,ULDB)
EXCEPT:     (Not-In, Nop) (Unauthorized, Bad-attempt(User))
```

Note that the cliche knowledge about the Remove Operation allows RAAP to make many deductions. It attempts to fill in the various roles and either succeeds in deducing which objects fill those roles based on the current context, or constrains the roles based on constraints in the cliche. The cliche provides the semantics for check-out. These semantics are reliable in the sense they come from a well debugged cliche definition. Furthermore this definition can guide search for relevant information. In the above example, it would cause consideration be given to authorization routines and the Bad-attempt routine.

```
RAAP> check-out = Remove(User,Book)
```

RAAP acknowledges with a version of its understanding of the statement. This acknowledgment might be augmented with a description of the roles in the Remove Operation that RAAP filled in. In any case this information is available if the analyst wishes to ask for it.

```
(Define return = check-out.Inverse)

RAAP> return = Add(User,Book)
```

From the knowledge in the information-system that the Inverse of Remove is Add (or perhaps derived from the attachment to Inverse, that gives conditions on pre and post-conditions for Inverse functions, and a search through IS.Operations) RAAP derives the definition of return.

```
(Define remove Name-of Remove
               (Fill-Role Relation-name With Book))

RAAP> Redundant definition of remove and check-out.
      Both are realized as equivalent invocations of Remove.
```

The error discovered above was somewhat deliberately inserted[1] to demonstrate redundancy checking. Check-out is a property altering operation which effects a book's status in the library but not its inclusion.

The original definition of check-out in terms of Remove was in error. The analyst's perception of the nature of the system is changing. Originally, he (I) thought that the library database was keeping track of a set of physical, uniquely titled books relative to whether they were in or out of the library, something akin to the notion of your personal library. The change to the use of a checked-out property corresponds to the notion that the library database tracks books relative to whether they are part of the library and whether they are borrowed or available.

The above models are psychological. They do not exist in RAAP. They are mirrored by various requirements constructs. The point to notice is that RAAP objects and commands ought to have a natural correspondence to the sorts of psychological models that the user will progress through as he develops a clearer understanding of the problem. Though no claims will be made for literal correspondence, RAAP will provide support for both the use of objects and cliches, and for their structural modification to support model evolution [6].

RAAP has pushed into a redundancy resolution mode. It will try to use the next input to resolve the redundancy.

---

[1] In my initial analysis of the library problem I made a number of errors, including the definition of check-out in terms of Remove, due to lack of a good global view of the problem. In retrospect the errors are obvious but their cause is a legitimate problem to be addressed.

```
(Define check-out Name-of Alter-property
        (Fill-Roles Property With checked-out
                Value With True))
```

```
RAAP> check-out = Alter-Property(User,Book,checked-out,True)
      Therefore remove realized as IS.Remove
```

Check-out is given a new definition, note the inference of the Schema property being Book from the old definition, this resolves the redundancy and RAAP indicates by use of the word "Therefore" that remove retains its original definition based on IS.Remove. There is no need to repeat this definition since it has not changed. RAAP points to the previously defined object of concern in an abbreviated fashion.

```
RAAP> return is no longer the Inverse of check-out
      Unable to deduce a new definition for return
```

RAAP's consistency checking reveals that the stated inverse relation no longer holds. It attempts to deduce a suitable binding of return to maintain the relation but cannot find one. RAAP prefers not to change check-out back to its old consistent definition since it was just changed and presumably the intent is to actually change it. The reason RAAP cannot find the new definition depends on the notion of Inverse it has but can be boiled down to the fact that in this case RAAP does not know that the inverse of True is False. Granted this may seem unlikely but imagine if the analyst had defined check-out as altering the "status" to "gone". What is the inverse of "gone"? In this case our analyst uses a heuristic of preferring boolean properties. The next statement will allow RAAP to recognize that the checked-out property is boolean. Introducing that cliche seems more plausible than introducing the highly domain dependent "status" cliche.

```
(Set Type checked-out Boolean-toggle)
```

Boolean-toggle is a cliche that says two things. The value is a boolean and it can only be set to the inverse of its current value. Intuitively, checking out a checked-out book is not sensible. The information is now present for RAAP to compute an inverse.

```
RAAP>Return = Alter-property(User,Book,checked-out,False)
```

```
(Agenda Choose book-id)
```

The agenda is a list of expected inputs that RAAP maintains as another aid for the analyst. For his own reasons the analyst chooses to address a particular entry (book-id) now. In this way he does not have to pop and push through contexts or specify a long pathname. This information is in the agenda item. Also by specifically choosing a pending agenda item the user receives a more in depth description of what is required.

```
RAAP> Context = Book
            There exists a choice as to the default method for uniquely
            identifying a book in procedures that require a book as input.
            They are:
             1. Unique-Key ISBN
             2. Unique-Key Title Cross Author
```

```
(Choose 1)

(Define remove-all Name-of Remove
              (Fill-Role Relation-name With Book)
```

```
RAAP> Redundant definition remove-all and remove.
      Both are realized as equivalent invocations of Remove.
```

Another clarification in the analyst's view of the ULDB is made. The assumption that one copy of a book is the same as any other (perhaps because a personal library has only one copy), mirrored in the definition of the Unique-keys, is corrected below. The new definition accounts for a library database that must keep track of many instances of the same title.

This points to another large research area involving debugging assistance. RAAP simply points out the errors. It attempts no explanation of the cause of such errors, e.g. there are no "bug cliches". However, RAAP should provide assistance for traversing the logic path that led to its conclusions.

As before, RAAP has pushed into a redundancy resolution mode.

```
(Fill-Slot Book.Unique-key With
           (Book.Unique-key Cross Book.Instance))
```

```
RAAP> Remove = Remove(User,Book)
      Because Book.Id was modified,
      Conjecturing remove-all assumes old definition of remove.
```

Modification is a key focus of the scenario. It is what a requirements analyst spends a lot of time doing. Thus it is important for RAAP to support the propagation of changes. It was assumed that the statement would help resolve the redundancy. The change to book did not resolve the redundancy rather it could have been propagated to effect both remove and remove-all. A heuristic assumed this was meant to create two definitions based on IS.Remove, one to be bound to remove and the other to be bound to remove-all. Remove is bound to the changed version since if the analyst wished to change the binding of remove-all he could simply have redefined it.

```
(Define missing Name-of Alter-property
           (Fill-roles Property With checked-out
                       Value With missing))
```

```
RAAP> Value missing incompatible with type Boolean-toggle
```

(Alter Type checked-out Augment-range With missing)

The addition of missing is in anticipation of further functionality. The solution adopted by the analyst of simply changing the range is sloppy. Later in the scenario, focus will be brought back to the checked-out property and RAAP will detect the problems with the change just made.

(Done Context Operations)

Done is a context pop with the additional property that it causes RAAP to think about what it has seen and perform a completeness and structural analysis. In matching the ULDB against the Information-system cliche RAAP will find some omissions.

```
RAAP> Context Operations.Add Empty
      Implication: No way to enter data into Information-system

RAAP> Context Operations.Query Empty
      Implication: Limited ability to observe contents of Information-system
```

These deductions are driven by the Information-system cliche. The ULDB, an instance of this cliche, still has unfilled roles.

The next deduction is an example of a structural analysis that might occur if the property slot of Alter Operations were hooked to a sophisticated cliche about finite state machines (FSM). When the type of the checked-out property was changed, the new set of possible values became (True, False, missing). The knowledge about FSM would include that of state transitions, accepting states, and dead states. A dead state is a non-accepting state that cannot be exited from. Since the operations check-out and return toggle checked-out between True and False:

```
RAAP> checked-out.Type.missing Viewed-as Fsm.Dead-state
      Implication: No transition out of State.missing
```

(Push-Context Reports)

(Define books-by-borrower Name-of Report :guided)

The :guided keyword directs RAAP to lead the analyst through the instantiation of the cliche. The analyst has given the dialogue initiative over to RAAP. The >> prompt indicates RAAP prompting for input.

```
>> Description ?
     Find out what books are currently checked out by a particular
     borrower.

>> Input ?
      <Borrower>

>> Using-relations ?
     Book

>> Using-fields ?
     Title, Due-date, Borrower

>> Compute-data ?
      Set of [ <Due-date>, <Title> ]

>> Data-computation-definition ?
      <Due-date> <Title> | (Book.Borrower = <Borrower>) &
                           (<Title> = Book.Title) &
                           (<Due-date> = Book.Due-date))

>> Print-header ?
      Books checked out by: <Borrower>

>> Output-sort ?
      Numerical by <Due-date>

>> Print-data ?
      <Title>

>> Boundary-Case ?
     No books checked out to: <Borrower>

>> End
```

RAAP steps the user through the roles in the cliche. Description is textual documentation. The Input description, <Borrower>, refers to an item of type Borrower. Since this is unambiguous there is no need to provide a name for the item. If the report took two Borrower's names as input the description might include a name, e.g. primary-borrower:Borrower, where the ":" separates the name from the type. The Using-relations is a redundancy mechanism that says which tables of information will be accessed. This could be deduced from the Data-computation-definition. The Using-fields is also a redundancy mechanism. The Field name Title refers to Book.Title but since this is unambiguous based on the Using-relations the abbreviation Title is fine. The Compute-data line redundantly describes data to be computed in the report. Note that not all Computed-data has to be printed. In this report note the use of two fields in Compute-data. One field is printed out in the report. The other is used to order the report. The Data-computation-definition says: "Compute the set of pairs of

<Due-date> and <Title> such that the Book's Borrower equals the Input <Borrower>, <Title> equals the Book's Title, and <Due-date> equals the Book's Due-date." The Output-sort describes how to order the set of Computed-data. The Print-header describes what to put on the top of the page. The Print-data describes the body of the report, i.e. the format of a line. The Boundary-case role is a textual description of the observation that the Report is empty.

```
(Push-context Authorization)

(Fill-roles Operation With books-by-borrower
                  Input With User
                  Function With
   (If (Not (Is-a <User> Staff)) Then (Must-equal <Borrower> <User>)))
```

A Report has an Authorization function. Normally data is provided by indexing off a particular Report or Operation and filling in its properties. In this case the analyst has centered on a particular property, the authorization function, and is indexing off it and filling in its value for various Reports and Operations. In this context the Operation role says which routine the description applies to. The Function definition uses the cliche Must-equal which describes a behavior that will return successfully if its arguments are equal but will signal the appropriate error if not.

```
(Fill-roles Operation With (Remove Remove-all)
                  Input With User
                  Function With (Is-a <User> staff))
```

The authorization functions for remove and remove-all are filled in.

```
(Push-context Data-constraints)
```

The Data-constraints on an Information-system are logical assertions that further define the behavior of the system. These constraints must be accounted for in the design process. RAAP checks them for logical consistency.

```
(Create Constraint
   (For-all Books (Book.Checked-Out Or Book.Available)))

RAAP> New Field Book.Available
```

This is constraint G1 from Figure 2-2. RAAP notifies the analyst that Available is a new notion.

```
(Define Book.Available Equal-to (Not Book.Checked-out))
```

The analyst provides the obvious definition for the new field.

```
RAAP> "Not" Incompatible with Type-of Book.Checked-out
```

The problem is with the possible value "missing" in Book.Checked-out. Here is where the lazy resolution of the problem with missing begins to cause trouble. Missing is really a separate property of a Book. Regardless of whether the checked-out bit is set, a Book is missing if it becomes physically inaccessible. The analyst decides to go back to his previous heuristic of making properties boolean valued.

```
(Invoke IS.Heuristics.Change-property-value-to-property
        checked-out missing)
```

```
RAAP> New Field Book.missing Type Boolean
```

The Information-system cliche also includes some algorithms for assisting in the requirements definition. This algorithm deletes the value (missing) from the property (checked-out), checks if the new range of the property is recognizable (it reverts back to boolean-toggle by dependency directed reasoning that detects the removal of the assertion that caused the change of type), and makes the value a Boolean property.

Now the analyst poses constraint G2.

```
(Create Constraint
  (For-all Books Not (Book.Checked-Out And Book.Available)))
```

```
RAAP> Previous constraint is a tautology
```

```
(Independent-constraint)
```

The function Independent-constraint states that even though a statement is a tautology it should be retained as a valid constraint if its current support is removed. Tautological statements may be useful for both human factors reasons and to assist in guiding the inference process.

The constraints posed above are of the form, in order, A or B, A = Not B, Not (A and B). Having all three of these may seem strange since asserting the second one implies the other two. The statement A = Not B, available = Not checked-out, is not quite true. Availability has to do with physical ability to produce the book whereas checked-out is indicative of a library procedure having occurred. The rule "missing implies ((Not available) and (Not checked-out))" would be included to model the possibility of a book getting misfiled. This conflicts with the definition "available = (Not checked-out)". In this case the analyst would have to decide whether or not to keep the definition. If he wishes to keep it he is in effect stating that the ULDB will not model the possible real world occurrence of a book being

mislaid. In order to model the real world more accurately he would have to revoke the definition in favor of a more accurate set of constraints on available, checked-out, and missing.

```
(Push-context Library.Fines (Fill-role Object With Book))
```

Fines is a library domain specific cliche applicable to different sorts of library objects. Up to now all interaction has been involved with instantiating the Information-system cliche. The analysis on domain cliches is the same as the analysis on general requirements cliches. This switch is made to demonstrate the use of cliches in detecting data dependencies and also as a reminder that domain analysis is part of the process.

```
(Fill-role Billing-function With Mail)
```

Mail is a cliche that is referenced inside the Fines cliche.

```
RAAP> New Field User.Address
```

RAAP informs the analyst that this is the first mention of a need for the user's address. The analyst is not sure he wants to provide the user's address and asks for the options available.

```
(Choices? Role Billing-function)
```

In this case a choice is explicitly encoded in the cliche. In general the choices? function might try to describe the options for a certain piece of the requirement.

```
RAAP> Fines.Billing-function can be filled by:

1. Daemon-invocation or 2. Mail

(Choose 1)
```

The analyst was not prepared to meet the data dependency regarding the user's address.

3. Discussion

User Interface

The interaction with RAAP is a dialogue. A major requirement on this process is that RAAP acknowledge input with replies that allow the analyst to determine RAAP's degree of understanding of what has been said. In the same way that the analyst can infer understanding from RAAP's reply, RAAP will attempt to infer the intent of the analyst's statements based on the context set by RAAP's previous reply. For example, if RAAP has noticed an inconsistency and revealed it to the analyst, RAAP may be disposed to using the next input to resolve the inconsistency. A secondary requirement is that RAAP also announce any unexpected interesting conclusions that it derives as a result of its input processing. Though the dialogue may be mixed initiative, RAAP, since it is an assistant, ought intrude infrequently. In concert with the assistant notion, the analyst will of course have the option of ignoring RAAP's inquiries and redirecting the dialogue.

The scenario could easily give the impression that RAAP will operate in a "line editor" oriented fashion. In addition to a command line, a more revealing picture of the RKB will be present. One major requirement RAAP must satisfy is the ability to produce a readable document from the RKB. One might envision that RAAP's display will consist of the evolving requirements document focused in on the area of most recent concern and that interaction with RAAP would be thought of as requirements editing. In the KBEmacs system [37] for knowledge based program construction, the interaction is of this nature. This view is natural with respect to program construction where the program text is clearly the end product. It is not completely appropriate to requirements analysis since the requirements document is not clearly the only desired end product.

Though requirements editing is a possibility, the following stance on the requirements analysis process points to a different mode of interaction. The creation of a requirements document is not necessarily the optimal conclusion of requirements analysis. However, technologically it is the only current feasible conclusion. Furthermore the creation of a requirements document, whether it is via the intermediary of an RKB or not, is an analysis process that requires contextual information to proceed. In general a requirements document is supposed to hide the possible options that could have been taken and concentrate on the implications of the path chosen. The analysis process, however, requires a view of the choices available. In addition to a view of the requirements document RAAP must provide a view of the current focus with respect to: the decisions that have been made, the decisions still to be made and the options available, and the options not chosen in the decisions that were made.

The display of a portion of the requirements document is one component of RAAP's output presentation. This will pose certain difficulties since information in the RKB is distributed in the document. Two consecutive inputs to RAAP may add information to two widely separated portions of the document. Even though input to RAAP is localized on a particular context this property may not be echoed in the evolving document. Another problem is that redundancy is present in the document and therefore a single RAAP input could also cause changes in two widely separated portions of the document.

A second aspect of the output will be a display of the current cliche highlighting roles that remain to be filled and constraints on those roles. Given the aggregate nature of cliches, this display should permit zooming in on cliches used to instantiate a particular role and zooming out to the parent cliche. A third aspect of the output, which will be partially redundant with the cliche display, is an agenda of expected pending input. As the requirements definition proceeds, RAAP will perform a completeness analysis that will determine information RAAP expects to be provided with. The agenda will organize this information for the user in a nonintrusive manner.

What is a Requirement?

Traditionally a requirement has been thought of as a contract between a system builder and customer. To be useful in this capacity, good faith and an understanding of intent must be applied by the parties involved. Attempts to write a perfect self-contained requirement, e.g. even for a simple text processing problem [26], have met with failure. In RAAP good faith and an understanding of intent are provided by the modeling skills of the requirements analyst as he describes the content of the requirement.

The best requirements consist of "why" descriptions, augmented where necessary with "what" descriptions, and exclude "how" descriptions. *Why* descriptions capture notions of intent. Perhaps the ideal requirement would consist only of a *why* description but this is infeasible for a number of reasons. First, it is unclear whether people can actually explain their needs without resorting to *what* descriptions, if only for the purpose of analogy. Second, the notion of what counts as a *why* description is not independent of the perspective and capabilities of the requirement reader. For example consider the following requirement: "A system is required for moving people from some given location A to a given location B". Is this a *why* description? Maybe, but one can always ask the question: "Why?" again and get an answer like: "A System is required that allows people to access resources at a given location B from a given location A." From a philosophical point of view the why questioning could go on until the answer is couched in terms of satisfying ones life function. Finally, *why* descriptions admit an enormous degree of freedom in realizing the system. They may lack

constraints to focus the design process on a manageable chunk of the solution space. Consider again the requirement for a system to move people from place to place. This may truly express the extent of the perceived problem but it admits too many solutions, an elevator or escalator, a highway, or the Star Trek transporter. *What* descriptions are used to restrict the requirement to a context in which system analysis can be reasonably done. A useful requirement will strike a balance between *what* and *why* descriptions. It should have enough *what* to focus the problem and enough *why* to guide the analysis of the *what* decisions. It should avoid *how* descriptions as these introduce decisions at a lower level of abstraction[2].

A requirement is a closed system description. A closed system has its boundaries explicitly modeled. This permits the use of a closed word assumption to deal with omissions. As an example, consider a procedure for computing square roots in a real number arithmetic package. Assuming the package does not guarantee that only non-negative numbers will be passed to the square root procedure, the closed world assumption is that any real number might be passed to it. However, square root can only return a valid answer, i.e. a real number, for non-negative input. A closed version of the square root function would incorporate a check on the input and signal an error if necessary. An open version would rely on the felicity of the environment to not call it with a negative number and might fail. The closed system version, based on a particular environment description, will not fail as long as that description was accurate. Any failures that result from unanticipated interactions with the environment are not implementation errors rather they are errors in defining the closed system's boundaries. Adequately modeling the environment is clearly a hard problem and relies on previously discussed issues such as intent, good faith, and an accurate perception of the world. RAAP provides support for analysis of environment models and provides reuse of well debugged models via cliches. The implementation problem residual from the circumscription of function provided by environment modeling is more tractable assuming a knowledge rich automatic programming approach. The purpose of making this distinction is to support "blame assignment" and more importantly principled, albeit still unverifiable, techniques for problem analysis. This separation also divides the automatic programming problem into a portion that inherently requires human mediation and a portion that is conceivably a compilation process.

A requirement is a model of a piece of the world and a problem to be solved. In concert with the

---

[2]The assessment of what counts as why, what, and how descriptions depends on your perspective. Consider the following set of intentions: a person wants to be happy, to achieve this the person wants a good job, to achieve this the person wants a good education, to achieve this the person goes to State U. Once the why level is chosen, i.e. the intent, the what and how levels are determined. For example a good education is how a person might achieve happiness; it is also what a person requires to get a good job. RAAP assumes that the user provides the appropriate models of intent, can choose an appropriate why level for analysis, and assists in the design of a solution to the problem so described.

above characterization, requirements will be defined as consisting of four bodies of information: *a domain description, a problem boundary definition, a function definition, and a user interface design.* This division will permit delineation of the environment the system is to work in, the perceived problem in that environment, and the solution being worked on. Explicit knowledge of these boundaries provides a guide as to when good faith and intent must be applied. The environment description is particularly necessary since in the contract model it is defined by the implicit augmentation of a contract with a body of precedent knowledge and attachment of great meaning to key phrases.

The **domain description** is expertise about the problem environment. It is not the sort of information that normally might appear in a requirements document rather it might be an assumed body of common knowledge and common sense knowledge akin to the precedence knowledge in the contract model. This description is a partial model of the world. It allows all parties to agree on a model of the real world, provides an additional source of constraints, i.e. the real world constraints, and it can be worked on separately from any particular problem and reused [8].

The **problem boundary definition** is used in conjunction with the domain description to achieve a closed system description. It defines the limits of the systems ability to affect and be constructively affected by the world. Consider another small requirement: "The system needs to move people from a given location A to a given location B in less than a given time T." The "features" of this problem boundary definition are location and time. This feature definition is important in constraining the behavior of the system and in seeing that the system meets the needs of its users. For example, in this case a system might satisfy the above requirement but might also subject passengers to extremes of temperature. In addition to such errors of omission in feature definition (which will be addressed below) there is also the issue of recognizing features that are implicitly referenced. For example, if the requirement speaks of location and time, the derived feature of acceleration might be important. Part of the issue of effective use of knowledge is the ability to derive appropriate conclusions from implicit references. Thus in this case the system should be able to relate statements about acceleration to those about location and time. More interestingly, if the user makes another implicit reference to acceleration, disjoint from the reference via location and time, for example talking about passenger comfort, RAAP should attempt to make the reference to acceleration explicit.

Maintenance of an accurate closed system model is important. Incomplete domain modeling can yield a system that does not fulfill the analysts intentions, e.g. when a full moon caused a massive warning on our American Ballistic Missile Early Warning System [34]. The domain description provides a model of the world the system is to operate in. Incompleteness in this domain knowledge restricts RAAP's ability to analyze the requirement and may also yield system behavior that violates

the customer's intent. Feature selection corresponds to paying attention to certain pieces of domain knowledge and ignoring others. One purpose of making feature selection and the domain description explicit is to permit "assignment of blame" in system failure. The purpose of this is to distinguish between errors caused by knowledge absent in the system which is considered readily available and of clear relevance and errors caused by an incomplete perception of the real world.

As perceptions change, customers may wish to change their focus and pay attention to different aspects, e.g. features, in the domain. This raises the question as to what guides the selection of features. For example, in an elevator system you do not want to have to worry about features such as the inclusion of a mechanical arm that puts funny noses on the passengers; you probably do not have to worry too much about the temperature of the elevator, but you need to worry about accelerating the car to inappropriate g-forces. One would like to rely on some sort of closed world assumption and default scheme to guide the choice of features. Happily the design space provides one. If a feature is critical it is likely to have been modeled before. Design experience helps define a space of relevant features and thus RAAP does not have to explicitly worry about feature selection. Reuse of requirements analysis will provide appropriate relevance criteria.

The **function definition** describes the behavior that the system is to demonstrate. In the example above, the function is the actual movement of the people according to the model that was defined. This information may simply reference knowledge in the domain description or may provide new knowledge to be integrated into the domain description, at least for the current analysis. Consider a banking system with a domain description that included an understanding of deposits and withdrawals. The user may wish to describe a system that included the notion of a transfer. The definition of transfer would be part of the function description.

Finally, a system must have a **user interface design** that describes the needed flow of information, in terms of form and content, in and out of the computer[3]. This design will define the I/O specifications of the system and it will be presumed that the presentation methods defined by it meet the customer's needs. It may augment the function description, or it may simply describe how to access already defined functions. Automatic tellers at some banks were modified about a year ago to include transactions that permit expedient execution of what must have been considered common transactions. To withdraw fifty dollars and get a print out of account balances, for example, one has to press less buttons than before. No new ability has been introduced to the system but the augmentation does impact speed and simplicity of operation.

---

[3] If the user is another machine instead of a person, then this concept still applies and the requirements are formal interface specifications. These describe a form of information flow more precise than that of an interface designed to satisfy an abstract notion of cognitive usefulness.

A requirement may be well formed yet unrealizable and this can happen in at least two ways. One could imagine stating a requirement that would require computing the halting problem or traveling faster than light. Though a challenge to our basic science, these are still valid descriptions. The situation becomes less absurd if we consider a requirement that requires a polynomial solution to propositional calculus satisfiability. This is an open problem and we certainly do not want to limit creative design in the requirements description [8]. A requirement should not, however, be internally inconsistent. If it can be detected that a requirement implies that A has property P and A does not have property P then we will consider it inconsistent[4]. There are a number of different measures of internal consistency. In Section 3.2 four domain independent conditions amenable to knowledge based analysis are discussed.

Creating Requirements

The process of creating an exemplary requirement may be characterized as follows. First the customer starts with a vague notion of a problem and a need for a solution. This vague notion is refined into a mental model of the problem that may include notions of relevant domain information, intentions, and solution spaces. Processes like WISDM [38] and JAD [23] have support for this internal explication of the requirement. This mental model must then be externalized and made concrete. Structured analysis techniques, such as SADT [33], provide a methodology for externalizing the mental model. For the purpose of this discussion the mental model will be called an informal model and its externalized versions will be called high level formal models.

Eventually a low level formal model, an implemented system, will be derived from a high level formal model. The purpose of RAAP is to assist in the process of creating the initial high level formal model (and refinements of it) from the informal model. Careful attention must be payed to, in effect, begging the question of this acquisition by providing RAAP a formal model to start from. In essence once anything is down on paper with semantic attachment to terms and a definition of problem structure it is a formal model. Because of this, certain approaches to automatic programming are inappropriate since they start from a formal model.

This is not to say that preliminary work should not be done before the analyst uses RAAP. It may be more useful for RAAP input to be based on a preliminary analysis that provides a lexicon of terms and a framework of relevant information on which the requirements analysis could be based. RAAP

---

[4]There is a slippery slope argument here regarding holding something like logic inviolate. Certainly one might imagine a world where propositional calculus satisfiability was solvable in polynomial time, so why not imagine a world where logic did not hold. My answer to this would be an empirical one. What you consider inviolate is determined by how much you would have to give up and change if it were changed.

cannot really substitute for certain processes, internal or external, that help firm up vague problem notions. Some amount of "thinking" and defining problem and solution boundaries is necessary before people can engage in constructive dialogue about system definition. RAAP provides a starting point for computer-based support of the system analysis process. It assists in the initial formalization of the informal requirement providing a repository for that information. As will be described below, it serves as an analysis tool for the formal requirement while automatically monitoring some of its critical properties.

The Assistant Approach

In the area of software specification research two approaches are quite visible. The first is to base the specification on a mathematical formalism, such as Petri Nets or finite state machines, and import the mathematical techniques associated with the formalism. The second is based on the notion of textual transformations. Neither approach seems fully appropriate to handling the capture of informal ideas into a requirement. To deal with this concept engineering problem, an apprentice approach is adopted.

One implication of an apprentice approach that is perhaps too strong for RAAP is the implied ability to learn. RAAP will learn in the sense that it can be taught reusable information but this does not cover stronger aspects of learning such as transfer of knowledge between problems, "ah ha" discoveries, or original design. With such an ability, an apprentice could become an expert. It is not intended that the initial version of RAAP will ever become an expert. In view of this, the word assistant will be adopted to describe RAAP's abilities. As an assistant RAAP will have the knowledge to enable it to automatically make the relevant, relatively shallow conclusions from statements and to perform deep inference when guided.

3.1 Knowledge Rich Computation

Cliches

When the customer speaks with an analyst, each word is tied to a network of information. This means that the actual content of any sentence goes far beyond its limited surface meaning. One goal of RAAP is to perform analysis supported by access to the rich attachment associated with "cliches" in the domain of interest [37, 8]. This will give RAAP greater access to the deductive power in the domain. In particular, RAAP needs to represent the relevant logical connections associated with various concepts in the domain.

Two properties of cliches make this notion appropriate for requirements creation. The most obvious

property, the one to be stylistically avoided in writing, is that a cliche is used over and over. In knowledge representation this capability for reuse is a positive, computationally powerful quality. The second less obvious property is that a cliche references a body of knowledge that is more or less identical in each person. This is an important property if RAAP is to have the knowledge the customer and the analyst take for granted. The negative connotation that cliches are unoriginal or boring is unfortunate. RAAP will strive to be boring enough to reason with the cliches used by the analyst.

Cliches will be represented as an aggregation of roles structured by constraints between the roles. The roles in a cliche can be filled by other cliches or by reference to other structures called *basic RAAP objects* (described in Section 4). Role filling is one of RAAP's basic processing abilities. Some roles will be instantiated by the user. RAAP will deduce or constrain values for other roles automatically based on specific information provided by the user, constraints in the cliches, and other knowledge in the RL. The ideal application for RAAP is one where the analyst can fill in certain key roles and RAAP can proceed to make a relatively large number of deductions in effect redoing a previous similar analysis based on information stored in the RL.

Reusable Libraries

A desiderata for cliches are that they represent a great deal of information about the objects or phenomena they refer to. Getting this information into the machine will presumably be costly, so the analyst should be able to reuse these definitions to amortize the cost. This reuse is not likely to happen as a chance occurrence from one project to another. It will have to be planned by constructing libraries that model domains and problems in which there will be continued programming activity.

Requirements analysis involves tapping of an analyst's previous experience. Therefore one of RAAP's tasks will be to support effective reuse of requirements knowledge. Ideally, creating the RKB for a new system would make use of predefined cliche knowledge 90% of the time and would require detailed and explicit description only 10% of the time. This fits the assistant model of computation in that an assistant has the ability to structure and store information and has his own store of knowledge from previous experience. The difference between the assistant and the expert is largely the ability of the expert to engage in creative thought, e.g. to notice analogies which recast problems in terms of already solved problems, and thus provide perhaps half of the remaining 10% of the needed information. The last 5% is provided by the "raw intelligence" of the expert.

To get a handle on the amount of information in a cliche and the investment required in constructing a system to assist in reuse of information consider this example. In the limit, i.e. once

most of the needed cliches are available, a session with RAAP proceeds with the analyst filling in roles in a cliche. Thus by the 90/10 notion, the analyst types in 2 pages of requirements analysis and RAAP produces a 20 page requirements document. On the other hand, if we consider a session with RAAP where many of the appropriate cliches are not defined, then the analyst must first type say 30 pages to define robust cliches, then type in his 2 page requirements analysis and finally RAAP will produce the 20 page output document. More information is input than is output. The obvious point is that a system designed to provide power by reuse of information is only useful if you do in fact reuse cliche definitions. On the second use of the cliche the total amount of data input is 34 pages and the total output is 40 pages. These numbers are fanciful. Perhaps our studies with RAAP will allow us to estimate what degree of reuse is required in order for the technique to be efficient.

3.2 Requirements Knowledge Base Maintenance

The following sections discuss four domain independent processes that RAAP will support for maintaining properties that are generally agreed to be characteristic of good requirements.

Consistency

Consistency might be narrowly defined as not holding two beliefs or propositions that are explicit converses of each other. The definition can be further extended to not holding a set of beliefs that allows derivation of the converse of another belief under the particular laws of deduction governing the system. The extended definition is usually not practical to compute. When a proposition is added to the RKB, RAAP will try to prove its negation in order to check for consistency. This procedure will make use of a notion of relevance of information in the RKB and RL in order to limit the amount of time spent on this proof procedure. The following scenario is envisioned. While in an interactive mode with the analyst the notion of relevance will be limited so that the checks for consistency are relatively shallow. At the analysts request, perhaps overnight, the notion of relevance will be loosened to permit a more complete check for consistency.

Completeness

Completeness of the RKB will be analyzed with respect to a closed universe defined by cliches. Cliches have roles and *basic RAAP objects* have properties to be filled. If RAAP can not deduce the value of some role or property then it is noted as an incompleteness to be resolved later. Completeness checking can be invoked explicitly or implicitly by a shift in focus in the the analysis process.

The interconnected nature of the knowledge base can be exploited to go beyond the explicit definition of completeness. To borrow from the scenario, consider a requirement where an operation

that adds books to a library is described (required). There are three axes from which this can be generalized (or specialized): books, libraries, and add. This might lead to consideration of operations that: add paper based literature to libraries, add physical objects to collections, alter the book content of the library, and remove a book from the library. Generalization in a hierarchy is only one way that the structure might be searched. Searching related structure might lead to the following sort of interaction. A particular function "x" has been defined and in searching up towards its containing class it is found to be a unary function. Searching out from the unary function class the notion of idempotence is touched. Consideration may now be given to whether this particular function is idempotent.

Redundancy

Redundancy is handled in a similar manner to consistency but instead one tries to infer the statement itself rather than its negation. One may still wish to keep redundant information since it may prove important with respect to change of the RKB or it may be indicative of a potential bug as will be seen in the scenario. If the analyst makes a redundant assertion he may not be aware that it was a logical conclusion of all that has been said. When this happens there are two distinct courses of action. The first is to note the assertion as an explicit statement of a fact deducible from the RKB. The second is to add new support for the statement which will permit it to remain true even when the original set of supporting assertions become false.

Ambiguity

In RAAP, underspecification and incompleteness will qualify as ambiguity. In this sense RAAP's basic goal of supporting the evolutionary description of a requirement, which includes the processing of fragmentary description and informal terms, will provide tolerance for ambiguity during the requirement creation process and assistance for resolving this ambiguity during analysis.

Ambiguity could also be dealt with in a more linguistic sense, but this will not be supported. Imagine this simple cliche hierarchy: a bicycle cliche with two specializations: touring-bicycle and racing-bicycle. Under the assumption that the analyst knows the proper names for cliches he could only access this information through the names touring-bicycle or racing-bicycle. Accessing through the name bicycle means none of the unique connections to the two specializations will be made available to the analyst. The cliche bicycle is a less detailed version than the specializations. Ambiguity processing could be added by assuming that when the analyst says bicycle he really means touring- or racing-bicycle and attempting to resolve the ambiguity based on further assertions.

3.3 Knowledge Integration

In RAAP, the notion of knowledge integration has to do with the acquisition and presentation of data so that it can be interpreted as a coherent whole. Coherence includes consistency between different parts of the data. It also includes a notion of completeness given the degree of understanding of the presentation mechanism. Recently there has been attention to providing such knowledge mediums [14, 24] with a shift away from the task oriented nature of AI. Successful creation of such mediums will support both intelligent analysis and intelligent knowledge sharing and communication.

In RAAP emphasis is placed on integration of knowledge in the RKB. This is reflected in the consistency, irredundance, and completeness processing and in deductive role filling. This goal can also be viewed in terms of the creation of a useful, coherent requirements document. It also explains the lack of emphasis on task oriented requirements analyses.

The process of validating that the system described by a requirement actually solves the real world problem under consideration is currently left almost wholly to the skill and experience of the project leaders. It is unclear whether any techniques can formally guarantee this correspondence. In any case, considering that enhancement of systems is inevitable, the need for tools that present a unified picture of the requirement to be used as a basis for communication, debate, and analysis is great. RAAP's goal of knowledge integration addresses these issues.

Reuse of cliches is a powerful mechanism to support the knowledge integration task. The original definition of these cliches is therefore crucial. Some cliches may be derived from preexisting ones via structural changes, supported by RAAP, and semantic enhancement, supported by the intelligence of the user. Manual definition of enhancement to cliches or new cliches requires the analyst to enter a comprehensive amount of information to describe the new structure. RAAP will not support any particular data acquisition methodology.

Another knowledge integration problem is an aspect of the problem of dealing with inconsistency in the RKB. As described in Section 3.2, RAAP will detect some of these occurrences by attempting to prove the negation of a statement before asserting the statement. RAAP will not, however, necessarily be able to automatically resolve the discrepancy. This is the issue of detection versus correction. Current techniques provide little support even for detection. While RAAP may not support automatic correction, it will support detection and provide support for the correction process such as the presentation of relevant data. Presently the unknowing introduction of errors into a system description is a major problem. The cost of correction of these errors, if caught early enough, is

considered relatively inexpensive. Requirements analysis is the earliest and cheapest place to detect and correct errors.

The degree of knowledge integration in RAAP can also be viewed in terms of the degree to which RAAP can learn. To state the point before explaining it fully, RAAP can be thought of as an electronic notepad that provides access to and analysis of a highly interconnected knowledge base. Given a knowledge structure, RAAP can make full use of it in the context of a particular problem. As far as RAAP's ability to transfer its experience from one problem to the next (one account of learning) that is left to the adeptness of the analyst. Briefly, to explain why this sort of discovery is left out of RAAP, one view of learning would have it that two prerequisites for transfer of knowledge between problems are a well understood and expressive representation formalism and a concept of relevance to guide the search for useful information. These are currently unavailable in a suitable form, thus two primary exploratory goals of RAAP are: to suggest a viable formal requirements modeling language and to assist in analysis that will make clear what is relevant to the requirements analysis process. Put another way, I am taking the view that in this domain, effective learning cannot be done without reference to some representation, i.e. cannot be done purely on the basis of uninterpreted structure, and furthermore that the choice of representation is critical since it will predispose the learner to recognizing certain classes of behavior. The analyst may learn and reuse a previous analysis with RAAP's assistance but it will not be considered incumbent on RAAP to discover this possible transfer.

3.4 Requirements Analysis

The previous section on RKB maintenance deals primarily with incremental maintenance and augmentation of the RKB. Analysis functions operate on the entire structure of the RKB as opposed to localized pieces. Actually two classes of operations are associated with working on the entire RKB structure. The first is viewing operations. These present the data actually in the RKB without adding new links. If the RKB is viewed as a huge connected graph then it is clear that structured presentation of the graph is necessary. The second class is constructive analysis operations which are characterized by their ability to add new information, e.g. concerning security or performance analyses, to the RKB. Initially only viewing operations will be discussed. The knowledge required to conduct useful analyses is the sort which might best be built by knowledge engineers working with RAAP as a communications and reasoning medium.

Two RKB Views

One of the primary operations in RAAP is role filling, explicitly or by inference. The deductive ability is supported by constraints. Constraints can exist in a number of forms, between roles, on roles, on a *basic RAAP object*. A declarative representation of these constraints will exist in RAAP, and they may have complex interactions. In order to view the interaction of constraints, particularly when there is a conflict and the analyst must decide how to weaken one, a procedural view of the constraints on a particular entity is a useful view.

Consider the following example based on two constraints which effect the check out operation.

"Constraint 1: All system operations invoked must be accountable to a staff member."

"Constraint 2: All operations that check out books must be accountable to a borrower."

Suppose borrowers and staff are overlapping non-identical classes neither of which is contained in the other. Then consider the check out operation with respect to these constraints. One deduction from the two constraints is that check out operations must be performed only by borrowers who are also staff. In this case there are probably two different notions of accountability but the informality of the requirements description has merged them into one.

Propagating the full impact of each constraint will not be practical due to the highly connected nature of the RKB. By focusing on a particular entity and the constraints that affect it, the analyst is in effect making a relevance statement that allows RAAP to proceed with a deeper analysis. This analysis seems particularly useful for dealing with inconsistency and for interfacing with the design process. The designer must be able to see the constraints that effect an entity in order to allocate responsibility for their maintenance.

Another view of the RKB has to do with the presentation of objects defined in it. The links between objects that conceptually exist in the requirement are relations which must be accounted for in the implementation. Consider the following relations: R(Book,Title), R(Book,Borrower), R(Book,Checked-Out), R(Book,Available), R(Book,Id). The syntax reflects that, e.g. a relation exists between books and titles. The relations listed above are of many kinds. A book's title is an intrinsic property of the book like its author or weight. A book's borrower is an operationally derived property. Also there is really no ordering to any of these relations, so thinking from an implementation viewpoint for a minute, a book may have a borrower and/or a borrower may have a book. Checked-out is another operationally derived property, whereas available is a conceptually derived property

equivalent to being not checked-out. Id is a composite property perhaps derived from the book's ISBN. This list of relations can be thought of as useful input to a conceptual database design process [11].

The relations define the abstract view of the system and it is a design problem to decide which relations will be stored, which will be derived, and which will be recreated. Relation analysis can include a usage analysis that will describe how the relations are accessed in the requirement, i.e. which fields are generally inputs and which are outputs. For example, there may be references in the requirement to Book.Borrower or to Borrower.Book (the dot syntax has a property connotation). Book.Borrower refers to a book's borrower indexing the relation off of the book as input and outputting the borrower. Or for example, if available really means not checked-out then it should be noted as a synonym but not as a separate property to think about. From the requirements analyst viewpoint the use of this analysis is for both the concise presentation of relations used in the analysis and for a "conceptual cleanliness" check allowing redundant or mutually dependent relations to be recognized and presumably expressed in a canonical form or defined in terms of their constituents.

4. Implementation

Basic RAAP Objects

Examination of a few software requirements revealed certain basic units of communication used in describing systems. The major types of statements were: deadlines, purpose, operations, daemons, concepts (terminology), objects, constraints, external behavior, questions, and notes. Five of these classes of statements are included in the *basic RAAP objects*. These five appear to be the basic units of communication in the software function description. They were chosen to facilitate description of the functional requirements of a system.

A *basic RAAP object* is a structure with an associated special purpose semantics. The RKB contains information derived from the instantiation of *basic RAAP objects* either by the analyst or by the analyst through the cliche mechanism. When a cliche is instantiated the base of this recursive process involves instantiation of *basic RAAP objects*. Along the way, constraints between roles are recorded and these are also entered into the RKB.

A **Thing** object represents entities in the world, viz. the system being described, with properties. The special purpose semantics of *things* are constituency and inheritance.

An **Operation** object represents a function that is used to assist in the description of the required behavior of the system. With respect to the system being described, this function is explicitly invoked in one or more places. An operation has an I/O definition, pre and post-conditions that are logical assertions, and exception conditions that are pairs of exceptional conditions and logical assertions or function invocations.

An **External-behavior** object is like an *operation* with the single additional property that the user of the system can invoke this function and further that he can only invoke functions so designated.

A **Daemon** is like an *operation* except that its invocation condition is not explicit, rather it has a trigger condition which specifies by description when invocation should occur.

A **Constraint** is a property of the system being described that is to be maintained. There is no notion of invocation associated with a constraint, it is a requirement invariant. A constraint is a logical assertion that can refer to any of the objects above.

Preliminary work has been done on modeling a RL with KEE[5]. KEE provides a frame based reasoning system which has supported the definition of cliches and basic RAAP objects. The modeling activities in creating a RL are facilitated by the frame description and browsing features of KEE.

Work has also begun on using CAKE [29, 30] to support the reasoning needed in the creation of the RKB. Instantiations of cliches and basic RAAP objects can be translated into assertions in the CAKE system. Cake provides dependency directed reasoning and a truth maintenance system necessary for consistency and redundancy analysis. It also provides strong support for reasoning with and detecting equalities which is useful in redundancy analysis.

4.1 General Issues

Abstract Models and Representation

In the library example, when the analyst says "Book" he accesses a cliche that models the real world notion of book. What is in this model? It can be constructed from any of the *basic RAAP objects*. In particular, however, a model will have a feature structure (roles and slots), a list of associated constraints, a list of associated operations, and perhaps lists of other *basic RAAP objects*. Presumably any of these could be empty except the feature structure. It is central to the model. It is the data that the other objects refer to. The notion of modeling adopted is that all models have features which define their structure and everything else is function.

Certain problems exist in the scenario because the library database is an abstraction of behavior in the real world. An explicit understanding of the real world is necessary, and partly missing, to allow the propagation of certain physical constraints from the notion of a real world library containing physical copies of books interacting with real people to the abstraction of a database system tracking information about books interacting only with a librarian.

In the scenario, the Information-system cliche is used to model the library database system but there exists no cliche which models the physical activity that happens in the library. Notions like checked-out and available are not attached to their meaning in the physical world. The fact that a book can be not checked-out and not available, i.e. missing or stolen, is not modeled. Without this other model, it is harder to capture the intent of the customer. Both models should be present and an abstraction function should exist which defines the correspondence between elements of the computer model and elements of the real world model.

[5]Knowledge Engineering Environment (KEE) is a trademark of Intellicorp

Modeling and Propagating Invariants

When two models are brought into correspondence their features must be appropriately aligned. Consider a case where a model of "My-Book" is based on a model of "Book".

The alignment of features can happen in a number of ways (the lower case letters represent the names of features in My-Book semantically, perhaps not lexically, related to the upper case features of Book):

Book(Physical Object)	My-Book(Derived Object)	
Book Features	My-Book Features	Type of alignment
A	goes to a	equality (1-to-1)
B,C	goes to b	compression (many-to-1)
D	goes to d1,d2	expansion (1-to-many)
E	goes to nil	deletion (1-to-none)
nil	goes to f	addition (none-to-1)

Once the alignment of the feature structure is obtained, constraints can be propagated from Book to My-Book. In the sense that My-Book is a representation of Book there are two sorts of representation invariants on it. The first kind makes sure My-Book is a legal Book. These can be propagated from constraints on the features of Book. The second kind provides data invariants (canonicalizations) to assist in domain analysis. These are decisions to be made by the analyst.

In the library example the global constraints are simpler in light of the appropriate real world model.

Let W = the whole collection of books in the library.

Let I = the subset of W which is physically in the library at a particular time.

Let B = the subset of W which has been borrowed and traceable to users of the library.

I union B does not equal W since some books may be stolen, misfiled, missing.

Let M = the subset of W that is missing.

I union B union M equal W and the three sets are pairwise disjoint.

checked-out(book) = book in B

Logically, Not checked-out(book) = book in I or book in M

available(book) = book in I

Logically, Not available(book) = book in B or book in M

Notice that checked-out(book) does not equal Not available(book) and that available(book) does not equal Not checked-out(book), despite the assertion in the scenario that a book must be either checked-out or available.

The nature of the global requirements which refer to books being checked-out or available remain the same but now they are expressed at a more natural level, i.e. they refer to inclusion in various natural sets and properties of these sets are things that computer systems can automatically reason about quite well. The effect each operation has on these sets can be determined. If these constraints are violated by an operation then the user can be notified of the violation in terms of why it violates a real world constraint. More importantly the constraint is easier to understand and more accurate since it is expressed in the proper model, i.e. the model which generates it.

Analogy

It would be ideal if there existed cliches for everything the analyst wanted to represent and that in fact he chose the correct cliches the first time around. Suppose, however, the analyst picks the wrong cliche. For example, in the library example what if it were discovered that users were not always people but that **a user could also be a corporation?** Understanding how people can be users, i.e having a modeling scheme where a person can be viewed as a user, can assist in creating the analogy that allows a corporation to be viewed as a user. In other words if A occupies a role in B compute how C could replace A in that role. This seems to be a subgraph isomorphism problem. The current A that occupies a role in B is defined by a graph and the piece of that graph that defines its interaction in its role in B is what is of concern. To determine how C could replace A match the relevant structure of A into C. This is an NP-complete problem but the graph has typed nodes and arcs, i.e. features and higher order relations that have types and are expressed with typed links, which might make computation feasible.

5. Related Work

In looking at related systems there are a number of factors to consider. The first is whether they support requirements analysis, in the sense of creating a formal description from informal ideas, or what up to now has been considered the closely related process of refinement of formal system specifications. Having determined the coverage of the system the following three issues arise: representation technique, processing and analysis strategies, and lifecycle integration.

SAFE

The SAFE system [2] addresses issues considered critical to requirements analysis in this proposal. SAFE is a computer-based tool which deals with some of the issues of formulating formal models from informal requirements. The main issue addressed is integration of partial descriptions by using the context defined by the entire interaction. The input to safe is a stylized Lisp-English. The type of informalities SAFE resolves include missing operands, incomplete references, implicit type conversions, terminology changes, recognition of refinements and elaborations, and recognition of implicit sequencing decisions. SAFE also makes use of a domain description which describes the structure of parts of the domain and provides further context to resolve informalities in. Even though the input to SAFE was not natural language, some of the types of informality detected in the description are natural-language-like in that they are imprecise, perhaps incomplete and/or inconsistent, "textual" references. In RAAP, the cliche defines the context and resolution of partial description is done with reference to this explicit context. Certain natural language problems such as terminology changes are simply not permitted with RAAP. In a sense SAFE attempts to compile a description into a specification and in this sense it assumes more about the well-defined nature of the problem and domain than RAAP does. It also does not provide for analysis to assist in debugging the assumed correct but informal description. The SAFE system is closer in spirit to RAAP than any other system in that it attempts to provide a computer-based tool for resolving informality in system description.

Design Methodologies

Design methodologies (Structured Analysis and Design Technique (SADT), Jackson Design Method (JDM), STRUCTUREs - by Warnier-Orr, DeMarco's Bubble Charts, Yourdon's structured design, Structured Systems Analysis (SSA) by Gane and Sarson) address the extraction of informal ideas and their description in the formal syntax of the methodology. For the most part the only computer support for these methodologies is for the syntax and graphics of the methodology's expressive notation. Little computer support is provided to assist in analyzing the semantics not made explicit in the structure of the methodology's notation.

For example, SADT [33, 31, 32] can be viewed as a thought structuring tool. It provides a blueprint-like notation to capture informal thoughts and provides a methodology for guiding the exploration of unformalized thoughts. No knowledge based techniques are brought to bear on SADT to do semantic analysis. It is basically a pen and paper methodology. The modeling constructs of SADT are well defined enough so that every SADT model has a meaning (it is what it is) but there is no way to measure how well the model created corresponds with the customers needs except by methodological review. SADT also does not provide mechanisms to assist in reuse of models or incremental enhancement of models.

These methodologies diverge in spirit from RAAP in that they are not really computer-based. They do not formally capture the full semantics of the system blueprint rather they rely on the large body of cliche knowledge in the community. Any analysis of the requirement must therefore be done by hand. Certain aspects of data acquisition in RAAP might be greatly facilitated by the application of the thought structuring techniques of one of these methodologies. The RKB is intended to augment the graphical notations by supporting automatic analysis.

GIST

GIST [16, 18] is a specification language that has certain nice features including provision for modeling the environment as well as the system, historical reference to data, constraint specification support, and maintenance of relations between objects. Still, it is assumed that the initial GIST specification can be acquired from informal ideas by some unstated process. RAAP is concerned with obtaining this initial requirement from which to begin the implementation process associated with GIST (a transformational scheme). Nevertheless GIST does jump out of the realm of being just another programming language since its constructs constrain rather than explicitly define the program. "Compilation" of GIST involves making decisions in a very large space of possible implementations.

PAISLey

PAISLey [41] stands for Process-oriented, Applicative, and Interpretable Specification Language. It supports the creation of a process model of both the system to be built and the environment it is to operate in. PAISLey takes an operational approach to system requirements. This means a PAISLey model is a formal executable model. It can be used as a prototype of the system to be delivered. A PAISLey specification is a formal object and creating it is a difficult process (in the same way a GIST specification is a formal model and creating it is difficult). Once this formal model is obtained a transformational implementation approach can be adopted.

In [42] Zave contrasts the operational approach to the conventional approach, i.e. top down decomposition of black boxes. One major advantage that the operational approach has is that it avoids verification and testing phases on the implementation. Once the initial formal model is created and manipulated into a form where it is agreed that it **solves the problem**, it can be refined by correctness preserving transformations into an efficient implementation. This process does not introduce any bugs into the *agreed upon* correct initial specification. Of course there are other pros and cons to both approaches. As with a GIST specification, creation of a PAISLey specification is still hard and unsupported. It is also unlikely that an operational specification could fill the contract role of a requirement since the removal of imprecision also removes the flexibility of good faith and intent. This flexibility is needed because verifying that a formal description matches the intent of the customer is impossible.

In general, transformational implementation systems [4] diverge from RAAP's goals in that they require a formal model as input to the transformation process. Also for example, in [1] they work from an informal strategy which already assumes knowledge RAAP wishes to capture. The process of developing this informal strategy that already has a number of commitments, i.e. the strategy works and captures all the relevant features, is not described and presumably unsupported. Furthermore, they freely use and assume an understanding of domain terms that need to be defined, and should be formally represented, but are not. The acquisition of this high level formal model that anchors computer-based analysis is itself a complex task and it is the task RAAP addresses.

Formal Models

Many mathematical formalisms exist for specifying systems. Some examples are Petri Nets, Nassi-Shneiderman Diagrams, and finite state machines (FSMs). A number of systems are based on such formalisms. The Requirements Language Processor (RLP) [15] is based on a stimulus-response format for describing real time problems which is basically an FSM model. The Software Requirements Engineering Methodology (SREM) [9] is based on "R-Net" flowgraphs (requirements networks) which permit specification of data flows and process control in something of a structured programming manner. The requirement is stored in the Abstract System Semantic Model (ASSM). Certain analyses can be performed on the ASSM including report generation, static consistency checks, and flowgraph simulation.

Mathematical formalisms have the advantage of a theory to guide some sophisticated analyses. The disadvantage is that the representation is limiting. FSMs are best for describing real time systems that will perform in a stimulus response state machine manner. They cannot fully describe conceptual modeling problems.

An alternative "formalism" is the entity-relationship database approach. The Problem Statement Language/Problem Statement Analyzer (PSL/PSA) [35] is a requirements documentation and analysis system. The PSL is based on an object-property-relationship philosophy with eight predefined aspects required to describe a system. Data entered in PSL format is stored in a fixed schema database by the PSA. The PSA also supports generation of a number of standard cross-referencing reports. The database approach has the advantage of a powerful and flexible representation but the analysis that can be performed on this relatively unconstrained structure tends to be limited to static semantic checks and report generation.

DRACO

Draco [28] is a system for searching a space of possible implementations based on a library of reusable software components indexed by a domain analysis. Draco does not help in performing this domain analysis which is the hard problem RAAP is trying to provide assistance for. Draco is interesting because it takes seriously the idea of reusing abstract software components that are expensive to construct form scratch. Other work [13] casts this in the light of a software construction process analogous to that of hardware construction using integrated circuits. Certainly hardware designers still need computer aided design tools, but the situation would be unbearable if not for the advent of the increasing complexity of integrated circuits. It would seem, however, that software engineering has not produced anything more than SSI or MSI components and not nearly as much standardization.

RML

The Requirements Modeling Language (RML) [21, 20] is among the first, if not the first, applications of AI knowledge representation techniques to the software requirements problem. RML is a formal frame-based requirements representation language. It has three basic types of objects: entities, activities, and assertions. In addition, it includes a mechanism to model time concepts which are considered quite import in modeling many application domains. RML obtains its formal definition by virtue of a translation between RML and first order logic. On the other side of the coin, RML interfaces to the world of informal thoughts and modeling through a link to SADT. SADT bridges the gap between mental models and RML's semantic model. The SADT diagrams can be translated into RML by a relatively concrete process and augmented with other semantic information.

The sort of work that is a precursor to RML is that of conceptual modeling in database systems and the application of AI techniques to database systems. RML is in some sense a natural extension of such work when one views a requirement as a special purpose database.

Still, RML is a language. It lacks analysis tools, save consistency maintenance at the logic level. In a section on future work in [21], Greenspan talks about an open bridge to other work in AI including richer modeling techniques, multiple viewpoints, and presumably other types of analysis ability. Greenspan has taken the tack of designing a language for expressing a requirements model. RAAP concentrates on using the information in a requirements model to perform useful requirements analysis. No commitment has been made to a particular modeling formalism. Rather, RAAP's knowledge representation has been designed to be expandable to accommodate the breadth of knowledge that will be required to support the desired behavior.

RAAP

In terms of the factors that were listed at the beginning of this section to be considered in evaluating related systems, it is clear the RAAP is addressing the requirements modeling problem and not the processing of formal specifications. RAAP is concerned with obtaining a formal model from informal input, something of a concept engineering system. AI techniques are brought to bear on both the representation and the analysis capabilities and thus the advantages of a knowledge based inference system are to be looked for. Analysis ability is key in that RAAP is not only meant to provide modeling ability, but also to provide support for the useful requirements analyses as outlined in the scenario. RAAP does not directly address issues of lifecycle integration. Providing an RKB that can be used as a communication medium seems to be a good start, but the issues involved with such integration are so far ahead of current technology that they are best left to people with more resources. A proposal which addresses these issues in a long term development plan is the "Report on a Knowledge-Based Software Assistant" [19].

In some sense there has been only one really major breakthrough with respect to managing the complexity of programs that can be written. The introduction of high level languages, such as Fortran, provided insulation from low level details including machine specific dependencies. In some respects, progress beyond that has not been made. No matter how high level the language, we still have been programming, as opposed to creating systems, for the past twenty-five years [7]. Programming requires paying attention to the details of the specific language, i.e. how that language can be used for modeling, whereas what really needs attention is the actual modeling decisions for specific objects and activities. Automating requirements analysis is one step in the path to opening up a new level of complexity in our computer systems. It is a move away from managing programs and towards managing concepts. A goal of the research proposed here is to assist in the production of more reliable systems by partially automating the software requirements analysis process.

References

[1] Russell Abbott.
Program Design by Informal English Descriptions.
Communications Of The ACM 26(11):882-894, November 1983.

[2] R. Balzer, N. Goldman, D. Wile.
Informality in Program Specifications.
IEEE TOSE SE-4(2):94-103, March 1978.

[3] R. Balzer, N. Goldman.
Principles of Good Software Specification and their Implications for Specification Language.
In *Proceedings of the Specification of Reliable Software Conference*, pages 58-67. IEEE Computer Society, 1979.

[4] Robert Balzer.
Transformational Implementation: An Example.
IEEE TOSE SE-7(1), January, 1981.

[5] R. Balzer, T. Cheatham, C. Green.
Software Technology in the 1990's: Using a New Paradigm.
Computer 16(11):30-45, November 1983.

[6] Robert Balzer.
Automated Enhancement of Knowledge Representations.
In *IJCAI 85*, pages 203-207. 1985.

[7] Stephen Smoliar, David Barstow.
Who Needs Languages, And Why Do They Need Them? or No Matter How High the Level, It's Still Programming.
SIGPLAN Notices 18(6):149-157, June 1983.

[8] D. Barstow, P. Barth, P. Dietz, R.Dinitz, S. Greenspan.
Observations on Specifications and Automatic Programming.
Third International Workshop on Software Specification and Design :89-90, August 1985.

[9] T. Bell, D. Bixler, M. Dyer.
An Extendable Approach to Computer-Aided Software Requirements Engineering.
IEEE TOSE SE-3(1):49-59, January 1977.

[10] V. Berzins, M. Gray.
Analysis and Design in MSG.84: Formalizing Functional Specifications.
IEEE TOSE SE-11(8):657-670, August 1985.

[11] Janis Bubenko, Jr.
Information Modeling in the Context of System Development.
In P. Freeman, A. Wasserman (editors), *Tutorial on Software Design Techniques*, pages 156-172. IEEE Computer Society Press, 1980.

[12] R. Babb, R. Kieburtz, K. Orr, A. Mili, S. Gearhart, N. Martin.
Workshop on Models and Languages for Software Specification and Design.
Computer 18(3):103-108, March 1985.

[13] Brad Cox.
Software-ICs and Objective-C.
December 1984.
Productivity Products, International.

[14] D. Lenat, M. Prakash, M. Sheperd.
CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks.
AI Magazine 6(4):65-85, Winter 1986.

[15] A. Davis, T. Miller, E. Rhode, B. Taylor.
RLP: An Automated Tool for the Processing of Requirements.
In *COMPSAC 79*, pages 289-299. November 1979.

[16] M. Feather, P. London.
Implementing Specification Freedoms.
Technical Report ISI/RR-83-100, USC Information Sciences Institute, April 1983.

[17] Martin Feather.
Language Support for the Specification and Development of Composite Systems.
May 1985.
Submitted to ACM Transactions On Programming Languages And Systems.

[18] Stephen Fickas.
Automating Software Development: A Small Example.
In *Proceedings of the Symposium on Application and Assessment of Automated Tools for Software Development*, pages 3-9. November 1983.

[19] C. Green, D. Luckham, R. Balzer, T. Cheatham, C. Rich.
Report on a Knowledge-Based Software Assistant.
Technical Report RADC-TR-83-195, Kestrel Institute, August 1983.

[20] S. Greenspan, J. Mylopoulos, A. Borgida.
Capturing More World Knowledge in the Requirements Specification.
In *Proceedings 6th International Conference on Software Engineering*. September 1982.

[21] Sol Greenspan.
Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition.
Technical Report TR CSRG-155, University of Toronto, March 1984.

[22] J. Guttag, J. Horning, J. Wing.
The Larch Family of Specification Languages.
IEEE Software :24-36, September 1985.

[23] Anthony Crawford.
Joint Application Design A New Way to Design Systems.
In *Guide International Proceedings*. Guide International Corporation, 1982.

[24] Mark Stefik.
The Next Knowledge Medium.
AI Magazine 7(1):34-46, Spring 1986.

[25] Z. Manna, R. Waldinger.
Synthesis: Dreams --> Programs.
IEEE TOSE SE-5(4):294-328, July 1979.

[26] Bertrand Meyer.
On Formalism in Specifications.
IEEE Software :6-26, January 1985.

[27] James Neighbors.
Software Construction Using Components.
Technical Report TR 160, University of California Irvine, 1981.

[28] James Neighbors.
The DRACO Approach to Constructing Software from Reusable Components.
IEEE TOSE SE-10(5):564-574, September 1984.

[29] Charles Rich.
Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too.
In *AAAI 1982*, pages 193-196. AAAI, 1982.

[30] Charles Rich.
The Layered Architecture of a System for Reasoning about Programs.
In *IJCAI 1985*, pages 540-546. IJCAI, 1985.

[31] D. Ross, K. Schoman, Jr.
Structured Analysis for Requirements Definition.
IEEE TOSE SE-3(1):6-15, January 1977.

[32] Douglas Ross.
Structured Analysis (SA): A Language for Communicating Ideas.
IEEE TOSE SE-3(1):16-34, January 1977.

[33] Computer Magazine.
Douglas Ross Talks About Structured Analysis.
Computer Magazine :80-88, July 1985.

[34] Brian Smith.
The Limits of Correctness.
Computers and Society (SIGCAS) 15:18-26, 1985.

[35] D. Teichroew, E. Hershey, III.
PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE TOSE SE-3(1):41-48, January 1977.

[36] Anthony Wasserman.
The User Software Engineering Environment Methodology: An Overview.
In P. Freeman, A. Wasserman (editors), *Tutorial on Software Design Techniques*, pages 669-713. IEEE Computer Society Press, 1982.

[37] Richard Waters.
KBEmacs: A Step Toward the Programmer's Apprentice.
Technical Report TR 753, Massachusetts Institute Technology, May 1985.

[38] Western Institute of Software Engineering.
Using the WISDM Team Method to Define System Requirements.

[39] IEEE Computer Society, et. al.
Workshop Notes: International Workshop on Models and Languages for Software Specification and Design.
IEEE Computer Society Press, Orlando, Florida, March 1984.

[40] IEEE Computer Society, et. al.
Workshop Notes: Third International Workshop on Software Specification and Design.
IEEE Computer Society Press, London, August 1985.

[41] Pamela Zave.
An Operational Approach to Requirements Specification for Embedded Systems.
IEEE TOSE SE-8(3):250-269, May 1982.

[42] Pamela Zave.
The Operational Versus the Conventional Approach to Software Development.
Communications Of The ACM 27(2):104-118, February 1984.

[43] P. Freeman, A. Wasserman.
Tutorial on Software Design Techniques, 4th Edition.
IEEE Computer Society Press, 1983.