Automating the Analysis Process: An Example

Stephen Fickas

CIS-TR-86-08
January 14, 1987

Department of Computer and Information Science
University of Oregon

Automating the Analysis Process: An Example

Stephen Fickas

Computer Science Department
University of Oregon

To be presented at International Workshop on Software Specification and Design, Monterey, 1987

Abstract

Our goal is the automation of the requirements analysis process using a problem solving approach. In this paper we discuss a) the components of our ideal system, b) the test of a smaller demonstration system on problem #1 from the workshop problem set, and c) our prescription for further work in the area.

1. Introduction

We are interested in automating the analysis phase of software development[1]. We are working on a system that attempts to acquire a problem description that includes requirements on the objects, actions, and constraints of the intended system, as well as descriptions of the time and labor resources available during development, and the constraints on the run-time environment. In this effort, we make three assumptions worth noting: 1) we assume that the user may have only a vague idea of what he or she wants, 2) we assume that domain knowledge will be a required component of our system, and 3) we assume that our system will need knowledge of the development process as a whole, including the design, coding, and maintenance processes, resource requirements, and

[1]Our use of the term analysis is fairly broad, and includes problem acquisition, validation, and production of a formal specification. It tends to encompass traditional notions of system and domain analysis, requirements analysis, and specification.

constraints on the runtime environment. These assumptions run counter to the view of analysis as a process of *translating* or *rephrasing* user intent to formal or semi-formal documents. In the translation view, the human analyst is taken to be expert in interviewing and model building skills, but ignorant of the domain. Lacking domain knowledge, the analyst focuses on syntactic error detection techniques similar to those one might find in a good compiler, e.g., missing inputs and outputs (parameters), interface mismatches (typing errors), unused data (dead variables), disconnected processes (dead code).

In our view, the production of a requirements document is not so much a translation process as an interactive *problem solving* process with both user and analyst involved in supplying parts to the final product. This requires our analyst to have a thorough understanding of the application domain, and the ability to both critique user descriptions, and suggest components of its own. Part of this process is indeed the type of syntactic analysis discussed above. However, we are attempting to extend analysis into validation of intent. That is, we expect to take a perfectly correct (syntactically valid, consistent, unambiguous) description of requirements, and attempt to poke holes in it. Our analysis will be based on a model of what is common and desirable in the domain.

Our proposed system, called KATE and described more fully in [5], is being built around the following components:

- A model of the domain of interest. This includes 1) the common objects, operations, and constraints of the domain, 2) the known hard design and implementation problems, 3) how the environment might affect the embedded system, and 4) a model of typicality in the domain that is rationalized by management policies and the way human/system interaction can be expected to unfold. We expect our set of domain models to grow over time, although as discussed in section 2, we have only one currently.

- A problem acquisition component that controls the interaction between client and system. Issues include use of abstraction in description, use of simulation in presentation, reuse of previously described and analyzed systems, recognition of user descriptions in model terms, and dialog[2] control in general.

- A critic that attempts to poke holes in the current problem description. Bugs that are of interest to the critic include syntactic errors such as missing inputs, disconnections, type mismatches, as well as "domain errors" that violate KATE's view of

[2]The system uses a non-NLP, graphics interface as a communication medium. We use the term dialog here to mean the interactive, bi-directional transfer of information.

typicality.

- A specification generator that can map a requirements model to an existing specification language. Our first attempts have used Gist [11] as a target, although we are interested in others. We would hope to have pluggable back-ends for various formal specification languages.

This paper will focus on the first and third components, the domain model and the problem description critic. In the spirit of the workshop, we will use one of the designated problems, that of a library management system, as an example.

2. What Makes a Good Analyst?

We'd like to be able to report that we have a satisfying answer to this section title. However, the best we can do is present clues we have picked up in studying analysts in action. Our work in this area is divided into two separate parts, both of which are discussed below.

2.1. Student project findings

Over the last four years, we have observed the software projects in our two quarter, Senior Project course. This has been a good testbed: we have been able to view not only the analysis process, but each of the subsequent post-analysis phases of development. In particular, we have been able to interview clients[3] well after the software has been in use, and find their level of satisfaction with its functionality, interface, and general performance. Using this information, we have attempted to trace client complaints back to analysis bugs. There has been at least three things that have emerged from this trace-back study: 1) the beginnings of a classification of analysis bugs, and a prediction of their manifestation in delivered code, 2) insight into what analysis techniques allow one to avoid certain classes of analysis bugs, and 3) conversely, what type of analysis deficiencies lead to certain analysis bugs. To give the reader a flavor on the direction this work is going, we will present an example of each below, and at the same time reiterate that our findings here are speculative, i.e., they have not been tested in any formal, experimental sense.

Analysis Bug: *the naive physical model.* In domains that require an information management system (IMS) to control a physical system (PS), the IMS must represent the

[3]To add realism to the course, students are encouraged to seek out clients from the local university and business communities with real software needs.

PS in an adequate level of detail. If the level of detail is to too idealized or naive, the delivered IMS will not model certain events and associated states in the PS, and will have a propensity to become wedged (see section 4.6 for a good example of this in the resource management domain).

Useful Analysis Technique: *usage scenarios.* To combat the naive physical model, we have found that the more scenarios that can be generated showing the system in use, the better the chance of finding the events and states that must be represented. The best analysts among our students were the ones able to find an adequate level of representation by setting up hypothetical situations for a client to work through. **Corollary**: the more knowledge a student has about the client's domain, the better he or she is at generating scenarios.

Non-Useful Analysis Technique: *the customer is always right.* Students who relied on the client knowing what was possible and what was required generally produced only marginally accepted systems. While students often held spirited discussions on what was deliverable in a given time frame, they rarely questioned the client's statements of need, acting more as Santa jotting down a wish list. As we will discuss in the next section, this seems contrary to the behavior of experienced analysts, and in our observation, is a critical factor in eventual client dissatisfaction with delivered code.

Of course there are inherent limitations in observing inexperienced student analysts in a largely academic setting. This lead us to devise a more realistic situation where we could observe experienced analysts working on larger projects. The next section discusses our work in this area.

2.2. Protocol analysis findings

In this section we discuss our initial analysis of laboratory protocols we have been collecting over the past six months. In all, four separate protocol experiments were conducted with three different analysts. Three of the experiments involved library automation systems, and the fourth an information management system for an academic department. All analysts were experienced in the problem domain, experience ranging from 5 to 15 years. All clients were considering a computer system and were eager to talk with an analyst about their requirements.

Our goal in these protocol experiments was one of exploration. We wished to use the results as a foundation of a control model for KATE. Below we itemize some of our protocol findings that will likely be realized in such a model.

Finding 1: experienced analysts use their domain knowledge to zero in on the key questions. They initially ask a set of context setting questions, and then begin to list

alternatives to consider. In particular, our analysts never asked a client for a free-form recital of his or her needs; on the contrary, our analysts often did most of the talking.

Finding 2: experienced analysts use hypothetical examples as both an explanation device and an acquisition device. We view this observed behavior in our protocols, along with similar observations of the behavior of our best student analysts, as strong evidence that an example-based acquisition and explanation component should lie at the heart of our control model in KATE. Section 3.3 discusses our current efforts along these lines.

Finding 3: experienced analysts are aware of the higher level policy issues in a domain, and are able to use this knowledge to show a client the benefits and drawbacks of including certain requirements in their document.

Finding 4: summarization is really verification. A repeatedly observed behavior was the analyst stating that he or she was finished, but just wanted to summarize his or her understanding. In practice, this would signal not an end to the session, but a whole new round of discussion, e.g., "Oh, that reminds me, I haven't considered ...". This would often be repeated two or three times during a session. In our view, the process of summarizing by the analyst was an attempt to verify that all items had been considered and that there was complete understanding between analyst and client. Rarely was either true, spawning off more discussion of components the analyst had forgotten to deal with, or patching misunderstandings between analyst and client.

While what we have listed is far from forming a complete control model, we have attempted to work some of these findings into a demonstration system. This system is introduced in the next section.

3. A Demonstration System

Our interest is in acquiring a problem description from a user, storing it in a form that allows us to reason about it, doing our best to find problems with it before design and coding starts, and finally mapping our representation to a language from which implementation may proceed. In section 1, we described the components of our ideal KATE system. We now describe a demonstration system, called Small-KATE (SKATE for short), that implements a subset of the components, i.e., a domain model and a domain critic.

One of our goals in constructing SKATE was to show that domain knowledge could be used to find errors in a problem description that would be difficult-to-impossible to detect using solely syntactic knowledge. We use SKATE to do this in the following way:

(1) We choose an application domain, and hand code a representation of it[4]. Call this representation the MODEL.

(2) A problem description in the application domain is obtained, and a hand-coded representation is produced. Call this the EXAMPLE.

(3) Hand coded correspondence-links are forged between the MODEL and the EXAMPLE. Because an EXAMPLE-to-MODEL link is often hypothetical, each such link is made in a separate context[5]. In this way alternative mappings from EXAMPLE to MODEL can be maintained, reflecting the frequent ambiguity in a client's description (for example, see section 4.4).

(4) A rule-based critic is constructed that looks for both syntactic errors, and for what we call intention or domain errors, i.e., components of EXAMPLE that appear untypical or incomplete when compared with MODEL. The critic is called JOG[6].

(5) The output is a component by component critique of EXAMPLE, using MODEL as a basis for explanation.

As can be seen, many important features of KATE are missing in SKATE: a control model for interactive acquisition and analysis; automatic recognition of EXAMPLE components in MODEL terms; generation of a formal specification. In [5] we present a plan for gradually incorporating each of these components into future demonstration systems. For the remainder of this section we will discuss, in more detail, what components we do provide in SKATE.

3.1. The Domain Model

We have chosen resource management systems as our first application domain, and have constructed a corresponding domain representation in SKATE. Concepts from the domain include

Resources, e.g., physical resources, borrowable resources, information resources, humans,

[4]As will be discussed in section 3.1, we use KEE [9] as the representation language for all but the rule-based component of SKATE.

[5]In practice, this is accomplished by creating a new KEE World for each link made.

[6]JOG -- as in both a) to shake; to stimulate, stir up, as the memory; to nudge, and b) to run slowly -- is implemented in the Oregon Rule Based System (ORBS) [4].

office space, furniture, hardware, courses, sessions, seats, books.

Resource depositories, e.g., physical plant, buildings, class rooms, banquet rooms, conference rooms, borrowing houses, libraries.

Resource managers, e.g., staff, editors, custodians.

Resource users, e.g., attendees, students, computer users, borrowers, patrons.

Resource operations, e.g., add resource, remove resource, gain access to resource, return resource, consume resource, lose resource.

Usage scenarios, e.g., resource browsing, resource acquisition, gaining group membership, waiting for an unavailable (full, checked out, on order) resource.

Security operations, e.g., give authorization, remove authorization, check authorization.

Resource constraints, e.g., maximum size, minimum size, borrowing limits, time limits.

Resource constraint management, e.g., waiting lists, dunning notices, fines.

Queries, e.g., resources by attribute, usage statistics.

Environmental aspects, e.g., available human resources (staff), available computing resources, performance constraints.

Policies, e.g., maintain user privacy, maximize available resources, allow effective resource usage.

This forms what we called the MODEL component of SKATE in the last section. The implementation of MODEL is in KEE [9]. Frame-based components, called units in KEE, are used to represent resource management components. The basic components of MODEL -- objects, actions, constraints -- introduce no novel representation ideas in our KEE implementation: each is based on earlier work by Greenspan in representing requirements models in RML [7]. A straightforward mapping from Greenspan's RML/TAXIS language to KEE was carried out, and hence we will not belabor the implementation further in this paper.

However, we do note that two "RML extensions" were needed to augment our model representation. These two extensions form the core of our notions of typicality in a domain. We will discuss each of them in the next two sections. At the same time we note that our interest is *not* in the construction of yet another requirements language.

Instead, we would like to use existing representations when possible to free us to explore the issues that we are interested in, e.g., acquisition, analysis, debugging of requirements.

3.2. The representation of policies

The first extension deals with the problem of representing policies. Here we are addressing such domain concepts as "keep a sufficient stock on shelves", or "maintain a user's privacy". These in turn can lead to sub-policies such as "avoid hoarding of borrowed resources", or "protect privileged information". In our view, these type of policies have much in common with Wilenski's general notion of goals and meta-goals [14], and in a more concrete sense, the implicit therapy goals uncovered by Mostow and Swartout when "decompiling" a medicine-dosage advisor [12]. In particular, both research groups recognized that such goals can often be in conflict[7]. While we aspire to the richness of representations such as Wilenski's, the current form of MODEL has only the most primitive representation for policies: a simple goal/sub-goal hierarchy with goals linked, both positively and negatively, to the objects, operations, and constraints of MODEL. In this way, goals are used to rationalize the inclusion of an individual component. If a MODEL component is not present in EXAMPLE, we can argue about the consequences in terms of our policy goals. A specific example may help here.

One object in MODEL is *patron* (library jargon for *user*). One of the attributes of patron is a *borrowing limit*, i.e., the maximum number of books a patron can have checked out at any one time. In MODEL, this limit is linked to the following three goals:

- *avoid_hoarding*. This goal is a sub-goal of *keep_adequate_shelf_stock*. The linkage is positive: the limit goes towards achieving the goal.

- *promote_timely_usage*. This goal is also a sub-goal of *keep_adequate_shelf_stock*, and a close cousin of *avoid_hoarding*. The linkage is positive: the limit goes towards achieving the goal.

- *allow_effective_use_of_resources*. The linkage is negative: a borrowing limit may stymie a user who needs k items to carry out a task, and k is greater than the limit.

Suppose that a client describes a system that has no borrowing limit. Using goals and linkages, we would like to be able to point out to the client that while unlimited

[7]It is clear from talking with professional librarians that not a small part of their expertise is related to the successful management of conflicting goals of users and staff.

borrowing goes along way towards achieving the effective use goal[8], serious problems arise with the hoarding and timely usage goals.

Does this mean that the client was in error in omitting a borrowing limit? Possibly - the limit might have been inadvertently left out. However, the client might just as likely have been making an implicit tradeoff among policy goals, e.g., in the client's particular environment, having a large enough working set outweighs problems with low shelf stock. In SKATE we are attempting to make these tradeoff decisions explicit. We are doing this by 1) analyzing the type of policies one encounters in a resource management system, 2) finding a representation for them in MODEL, and 3) linking policy to model components. What we have yet to address is the type of second order policy knowledge that would allow us to critique a particular tradeoff, the type of knowledge one might find, for instance, in Wilenski's notion of a meta-plan [14]. Currently we can only point out the policies that impinge on a component; we cannot offer advice on the optimal achievement of conflicting goals. Our protocols show us that this is a major deficiency in our model: expert library analysts can advise a client on optimal tradeoffs, given a particular environment. For instance, the following advice was given in regards to setting a borrowing limit:

> "Most university libraries have an unlimited borrowing policy. Of course this can lead to problems, such as a department setting up an unofficial branch library by borrowing all of the books in a particular field... In the long run you have to give researchers access to a working set, no matter how large. One compromise that seems to work is to have recall on demand. Now for non-research libraries, other policies might be more appropriate."

In summary, one cannot argue about completeness in a requirements model without knowledge of what purpose components play in some larger setting. One piece of this larger setting, as we discuss in section 4.2, is knowledge of the environment in which the final system will run. Another is the policies one might find given the environment. We are attempting to represent both in MODEL.

3.3. Representing domain behavior

The representation of domain policy discussed in the last section forms one major component of our model of typicality. The second major component is based on an *operational* representation of the typical ways humans (staff, users) interact with a system. It is from these models of behavior that we can further illustrate a) the importance of a particular object, action or constraint in a system, and b) how it achieves or violates policy goals. An example may be useful here.

[8]Fully achieving it would require 1) an unlimited borrowing time-limit, and 2) full borrowing access to all resources, neither of which is found in its purest form in most libraries.

Suppose that one of the actions in MODEL is a query by a particular user on the resources he or she has checked out. Suppose further that a client decides to omit such a query in his or her system, i.e., there is no such query in the requirements description he or she is building in EXAMPLE. What can we say about this? From the view of policy goals, this looks like a good move: most queries that give out information about a user are negatively linked to privacy goals. Although in this case the query is limited to a user's own borrowed resources, there is always the chance, for example through deceit, that others may gain access to a user's choice of reading material. On the other hand, the action can be positively justified by its link to the goal of allow_for_user_forgetfulness, which is a sub-goal of allow_for_human_foibles. As discussed in the last section, these links, both positive and negative, can be used to point out the justification for the query in a resource system. Indeed, in our observations (see section 2.2) we found that expert library analysts did point to goals such as these when analyzing the inclusion or omission of a particular component. However, they did something further - they generated both positive and negative examples. It appears that these examples were enormously persuasive and illuminating to a client. For instance, the avoid_hoarding goal comes to life when illustrated with an example of a user or group of users decimating the stock of books on a particular topic just before a midterm or final[9].

As discussed in section 2.2, we are lead to the following supposition: the generation of tight, convincing examples is a key component of an experienced domain analyst's expertise. Analysts appear to not only use examples to point out problems, but to acquire new information and disambiguate what they already have as well. The question then is can we hope to automate the process? An affirmative answer would seem to rest on three items: 1) a representation of human/system interaction, 2) large amounts of domain knowledge, and 3) an example generation system that is able to use 1 and 2 at relevant points during requirements definition. In SKATE, we have begun to address the first two items. Our representation of human/system interaction is as *usage scenarios*, which are roughly similar to Barron's interaction scripts [1]. A usage scenario consists of 1) a set of domain objects that take part in the interaction, 2) an optional list of policies that are positively or negatively reinforced by the scenario, and 3) a network of nodes, transitions and connecting arcs. Nodes represent system states, transitions represent system actions (and associated constraints), and arcs connect nodes and transitions. For the purposes of this paper, a usage scenario can be viewed as a specialized form of Petri-net, where transitions are equivalent to MODEL actions.

We employ usage scenarios in SKATE to capture the way users and staff typically use and misuse a system in a particular domain. Because usage scenarios are tied into the

[9]One library analyst used exactly this example, and further pointed out that the concept of *books on reserve* came about at least partially because of this type of hoarding behavior.

components of MODEL, i.e., they are instantiated with the objects, actions, constraints and policies represented in MODEL, they rest by definition on domain knowledge. In SKATE we thus have at least the rudiments of two of the three components that we have argued are needed to generate examples during requirements definition.

What we are lacking is the third component, an example-generator that can decide when an example might be useful (e.g., useful in showing the consequences of omitting a component, useful in disambiguating a description, useful in acquiring components by having the client fill in the slots of a scenario), and what form it should take (e.g., use of abstract components, use of real data, shown statically as a network "program", shown dynamically as a trace of a network simulation). Frankly, to build such a component will require a much more extensive model of requirements analysis than we currently possess. In the interim, we have focused on one component of example-generation, that of showing the rationale of domain actions in MODEL by illustrating their role in usage scenarios of the domain. SKATE can use such scenarios to show the client the drawbacks (and benefits!) of omiting (or including) a particular action in his or her system. We will continue with our query example to provide some detail.

To recap, we assume that a) in MODEL we represent a query that allows a user to find out what resources he or she has checked out, and b) a client omits the query in EXAMPLE. As we will discuss in section 4, another component of SKATE watches for such omissions and provides criticism when appropriate. As part of the critique, SKATE will attempt to find usage scenarios that contain the query as a transition. Two are found in this case, both of which are discussed below.

The first scenario involves a user Ui who wishes to know the reading material of a user Uj. The key to this scenario is the condition that Ui be in a state of knowing Uj's identification. Reaching such a state can be trivial if personal names are used as id; passwords would require a more complex impersonation scheme. The objects of the scenario are two users Ui and Uj, and a resource R. The scenario is linked negatively to the goal of maintain_user_privacy.

> **Usage Scenario 1:** a user Uj has checked out a resource R; Ui is in a state in which he or she can identify himself or herself as Uj; Ui queries the system for the resources checked out by Uj; Ui moves to a state in which he or she is aware of the identity of R.

The second scenario involves a user U who has checked out a resource, but has now lost track of its identity. The objects of the scenario are a user U and a resource R. The scenario is linked positively to the goal allow_for_user_forgetfulness.

> **Usage Scenario 2:** a user U has checked out a resource R; U is in a state in which he or she has forgotten the identity of R; U queries the system for the resources checked out by U; U moves to a state in which he or she is aware of the identity of R.

SKATE will present these two scenarios to the client with a warning that both are inoperable without the query action. Does an inoperable scenario signal an error? In this case, both usage scenarios are more or less operational representations of policy goals, so the answer must be the same as with such goals: yes if the omission was inadvertent; no if a tradeoff is being made.

In summary, we believe that usage scenarios can form the basis for both example-based acquisition and example-based explanation in KATE. In SKATE we have only addressed the latter, and then only in the context of domain actions.

3.4. The Example

We have used MODEL to analyze problems in conference and course registration, space planning, and now library management. In particular, for the workshop we hand coded problem description #1 into an EXAMPLE model in SKATE, and hand forged the links between EXAMPLE and MODEL. In the next section we discuss our use of this information in critiquing the description.

Before presenting SKATE's critique, we note that the question of hand tailoring of knowledge might be raised at this point. That is, have we simply taken the problem from the hand out, and built the MODEL around it? Specifically, how general is MODEL? With the risk of sounding flip, our answer is as general as possible. While we are clearly influenced by seeing specific examples of library problems, we have attempted to build a general model of resource management systems and incorporate libraries as a subclass of such systems. Our library representation in MODEL has evolved from both talking to library scientists and analysts, studying library science literature (see for instance [2]), and finally, looking at specific libraries. Does this mean that MODEL is general enough to handle any possible library problem that one can come up with? Unfortunately not. It seems clear that one can always add some new twist to a problem that will not be covered by MODEL. This argues strongly for a component of KATE that can "learn" new concepts as they arise. However, the acquisition of new concepts remains a difficult problem in learning research (see [3] for a summary), and one that we have yet to confront. Hence, any components of EXAMPLE that cannot be linked to

MODEL will be unanalyzable be SKATE[10].

4. A Critique of the problem description

The component of SKATE that we will be concerned with in the following sections is a rule-based critic called JOG. JOG has the responsibility of finding problems in EXAMPLE, given MODEL and correspondence links from which to reason. A JOG rule takes the following form:

```
(defrule <name>
        <MODEL component>
        <correspondence link>
        <EXAMPLE component>
   -->
        <warning action>)
```

The left hand side of the rule matches on a mapping from EXAMPLE to MODEL. The right hand side takes actions to alert the client of potential problems. Because JOG is meant to interact with the user through a dialog interface (a component of KATE that is unimplemented in SKATE), its rule form is less than useful for presentation purposes. Hence, we will dispense with listing individual rules, and instead discuss the type of knowledge JOG embodies.

In the remainder of this section we will look at parts of the JOG analysis of problem description #1 as given in the handout (and reproduced in the appendix).

One final word before launching into the example. It seems clear to us that the description of the library management problem was used originally by Kemmerer in [10] to focus on specific topics in his research, and no claims were made that it was a particularly useful system being described (although he did describe it as a *University* library system). By using more or less the same problem for the problem set, the workshop committee seems to be also focusing on "research" specifications, i.e., ones used to test out the power of various specification techniques. Given this, we can be accused of setting up an easy target: the problem description was not meant to describe a system one would actually want to use, but simply to be relatively complete and consistent, and in an information management domain familiar to all. However we would argue that the problems we describe in this section are typical of domain bugs found in real systems, although they are not often all present in a single system such as the case here. In essence, we argue that if a client, sophisticated or otherwise, is left to provide uncriticized, free-form descriptions to an analyst who lacks experience in the domain,

[10]In practice, they are linked to the frame unknown_component in MODEL.

something very similar to the workshop description will be generated (see section 2.1).

We will critique the problem description in sequential order for ease of understanding. Sprinkled throughout are comments we collected from a library analyst we asked to critique handout #1. Our goal in taking this protocol was to confirm that the critique produced by JOG was consistent with that of an experienced library analyst, and to point out areas of JOG's analysis that needed further work.

4.1. Setting the context

A major goal in our work on KATE is to avoid requiring a user to generate tedious listings of the common objects, actions, and constraints of the domain. By incorporating a model of library systems, KATE should be able to make available much detailed analysis that will not have to be regenerated by the user. Hence, a problem description should be able to make use of shared knowledge of the domain to reduce the amount of detail provided. Listed below is one of the uses of such shared knowledge that we found in the problem statement.

Text: "*Problem #1: LIBRARY*" and "*Consider a small library database ...*"

We argue that this text is used to implicitly define 1) the concept of book as a borrowable resource with attributes author, subject area, title, current borrower, and last borrower, 2) the concept of user as a borrower of books, and 3) the semantics of operations such as *check out, return, add, remove,* etc.. These concepts are referenced in subsequent portions of the text, but are never explicitly described themselves.

As discussed in section 3.1, library systems are part of MODEL's more general model of resource management systems. Hence, the word LIBRARY can be used to set the context in MODEL, just as one might suppose it sets the context for the human reader. This context in MODEL brings with it, among many other things, books and their attributes, users/borrowers of books, and typical actions performed in a library system.

We also note that the formal specification from which the text description sprung was required, not surprisingly, to explicitly define all objects of interest including book, book author, check out, remove, etc.. Hence, a direct translation of the specification into text would have been something like " A library consists of the following objects: ..., and the objects have the following attributes: ...; check out is performed by ..."[11]. To reiterate, one of our major goals is to avoid forcing the user to provide such tedious descriptions, and it is one of the main reasons we moved away from focusing directly on the formal

[11] In [13], Swartout discusses the use of such a translation system for the formal specification language Gist.

specification process (a much more detailed discussion of this point can be found in [5]). It seems from this example at least that we made the right choice: the author of the problem statement is indeed appealing to the reader's shared knowledge of the domain to avoid defining concepts common to the domain.

4.2. Performance information

It is clear that the environment in which a software system is to be embedded is important in determining satisfiable and useful requirements. For example, we would like to know how large data sets are expected to grow, the expected response time to queries, the hardware if pre-chosen, the type of users of the system, and the number and type of staff available. The next bit of text from the problem description provides some of this information.

Text: "*Consider a* **small** *library database ...*" (emphasis ours)

We can read at least four possible meanings for small here:

(1) the library collection is small, i.e., a small-library database,

(2) the database will remain small, i.e., a small library-database (which is at least partially subsumed by meaning 1),

(3) the supporting environment is small, i.e., a small-time operation, or

(4) small is synonymous with simple, i.e., a simple problem involving a library database.

Kemmerer's original description [10] gives no clue since it reads "The example system considered in this paper is a university library database", obviously not a small-library database, nor a small library-database, nor a small-time operation, nor a simple problem.

SKATE can make use of meaning 1 or 2 (with a more precise definition of small) in conjunction with information on the runtime environment, e.g., "query response should be no longer than 5 seconds", "the hardware will consist of a PC model 2300". Given this type of information, SKATE can make crude estimates on the satisfiability of performance constraints, as well as the desirability of certain actions. We must emphasize the word crude here since we have only begun to look at the use of environment and performance information during analysis. Using Kant's Libra system as an analogy [8], we currently only represent rules of thumbs, and have no ability to do a deeper analysis of

cost/benefit tradeoffs.

SKATE can make use of the third meaning to further refine its expectations of typical components. If this is a small-time operation, then there may not exist staff to attach call numbers, track late books, nor supervise check in and check out. Available equipment may be limited to a single CRT, and software to an off-the-shelf database package. Given this type of information, we can clearly get a better picture of intent and the type of policies that make sense.

In regards to the fourth (derived) meaning, our experience is that simple is a code word for simplified, i.e., "Please consider the following problem that has been simplified greatly to allow it [choose one: to fit on half a page; to avoid messy problems; to focus on certain points].". We believe the reader will see as we analyze the description in the following sections that this is certainly an appropriate reading of small here.

Finally we note that when our library analyst reached this point, a host of questions ensued regarding the type of library, the environment in which it was to run, the staffing available, and the characteristics of the users. As we discussed in section 2.2, we saw similar behavior in all of our protocol experiments.

4.3. Action descriptions

The problem description next presents eight desired actions. The first six are found to be typical actions in SKATE's model, i.e., correspondence links were found from IWSSD (we use IWSSD here and throughout the rest of the paper as an instantiation of EXAMPLE in section 3) actions to MODEL actions. Other than to mention that

> **Text:** "*(3) Get a list of books by a particular author or in a particular subject area.*"

is ambiguous -- it can be mapped to either two separate queries (our parsing), or to a single query that allows a disjunctive form -- we will not discuss them further here.

The queries

> **Text:** "*(4) Find out the list of books currently checked out by a particular borrower.*"

and

> **Text:** "*(5) Find out what borrower last checked out a particular copy of a book.*"

are more controversial. Our library analyst thought the former was a potentially dangerous invasion of privacy, and should only be granted to a select set of staff. The

same analyst saw little need for knowing who last had a book once it was on the shelves, although finding out what borrower currently has a book checked out is clearly useful (although it seems to us that both involve the same privacy issues as 4). In fact, we parsed 5 to mean the latter, and hence linked it to a query in MODEL for finding the current borrower of a resource; we have no representation in MODEL of a query on the history of check out, although one could easily make a case for a general history maintenance component of a borrowing system.

As discussed in section 3.3, SKATE's critique of domain actions is based on the presentation of usage scenarios. The privacy points raised in the previous paragraph are presented to the client in this context (see a similar example in section 3.3).

4.4. Object descriptions

The next bit of text from the problem statement introduces classes of users.

> **Text:** "*There are two types of users*[12]: *staff and ordinary borrowers.*"

We parsed this to mean that a user is either *library* staff or non-staff (ordinary borrower). An alternative parsing came out when our library analyst read the statement. The analyst parsed *users* as *patrons* (i.e., borrowers) and staff as *organization* staff (e.g., university staff as opposed to university professors or university students), thus believing that the statement was defining sub-classes of library users instead of sub-classes of system users. We did not actually catch this alternative parsing by the analyst until much later in the session -- the analyst did not explicitly note at the time of reading the statement that this was her parsing.

This brings up a critical point regarding the use of implicit information. While many implicit assumptions were seen to be made by both analyst and client in our protocols, such assumptions appear to be double-edged swords. On the positive side, they allow the dialog to concentrate on the important details of the problem, and avoid the many mundane details that would quickly swamp all other concerns. On the negative side, without a verification step, there is considerable danger that misunderstandings, in the form of mismatched assumptions, will develop between client and analyst. The analysts we observed achieved verification through a summarization process (see section 2.2). In SKATE we have no such verification process; we can only hope that an erroneous assumption will eventually lead to a contradiction or an untypical interpretation of later items.

---

[12] As discussed in section 4.1, *users* as a group have no explicit definition in the text.

4.5. Constraint descriptions

The problem description next presents constraints on what class of users can perform certain actions.

> **Text:** "*Transactions 1* [check out, return], *2* [add a copy, remove a copy], *4* [find what books a particular user has checked out], *and 5* [find what user last checked out a particular book] *are restricted to staff users, except ordinary borrowers can perform transaction 4 on themselves.*"

The idea of privileges among user groups is a common one. MODEL has the concept of action invocation requiring particular user group status. The one interesting note here is that the constraint on action 1 seems ambiguous: to paraphrase, "check out and return are restricted to staff users." Does this mean that a staff person has to check out a book FOR a user (i.e., there are three objects involved in the transaction: user, staff, book), or that only staff personnel can check out books (i.e., ordinary borrowers cannot borrow)? While we took the former meaning, it is interesting to look at the consequences of taking the latter meaning as an alternative. In fact, we created this parsing by restricting the participants of the check_out action in MODEL to be staff only. Running JOG on this alternative interpretation, no *syntactic* inconsistency was found. However, the following warnings were given:

Warning 1: the check_out action calls for staff control; a small-time operation may not be able to afford such staffing[13].

Warning 2: there is a usage scenario associated with borrowers forgetting what they have checked out (see Usage Scenario 2 in section 3.3). The scenario has two transitions (actions): check_out and query_resources_borrowed. While the query transition can be instantiated with an action invokable by an ordinary borrower, the check_out action cannot. In essence, ordinary borrowers can 1) find what books they have checked out, but 2) they cannot check out books.

What is compiled out of a usage scenario explanation such as the one above is a representation of why the *forgetful user* scenario is useful, or how its steps necessarily characterize it. Thus, relying strictly on usage scenarios, we cannot generate the perhaps more satisfying explanation

> "Useless queries are bad. Any query that returns known information is a useless query. Any query that returns a known constant is one that returns known

[13] Of course, the same warning was generated in our preferred interpretation as well.

information. Transaction 4 [query own books] always returns "nil", and it is known that it will return nil. Transaction 4 is a useless query."

The ability to reason from first principles in this manner is lacking from SKATE.

Warning 3: the set of participants of the check_out action in MODEL is {staff, patrons}. The members of this set are linked positively to the policy goal of allow_effective_use_of_resources. The removal of patrons (A.K.A. ordinary borrowers) from the set causes JOG to complain about the loss of support for the goal.

There are two interesting points to note about the analysis that produced the last two warnings. First, we would argue that these warnings are probably enough to alert a client to a potential interpretation problem. The second warning points out a kind of weird usage inconsistency, while the third notes the removal of support for a fairly important goal in a library system.

Second, the analysis was carried out in a representation-*dependent*, but domain-*independent* manner. In particular, the knowledge embodied in JOG to produce the second and third warnings pertains to usage scenarios and policy goals in general. The knowledge used to produce the first warning is domain-dependent. However, the domain is the rather broad class of physical-resource borrowing systems. By including library systems as a sub-class, we automatically inherit the relevant JOG rules.

4.6. More on Constraints

The next constraint described presents an interesting problem:

Text: "All copies in the library must be available for checkout or be checked out."

This clearly refers to the information management system, i.e., a book's status in the database can only take on two states. However, in the physical system it may be impossible to guarantee that this constraint holds; books can become lost or stolen. In general, many problems arise in information management systems that model a physical system in an ideal and unrealistic fashion.

Because the state-values *lost* and *stolen* are linked positively to the goal allow_for_human_foibles in MODEL, JOG warns the user of loss of support for the goal when they are eliminated in EXAMPLE.

Finally, we note that we observed a correlation between naive modeling of the physical world and wedged states in the IMS (see section 2.1). It appears that usage scenarios are at least one means of showing this correlation. In particular, we can foresee using a usage

scenario, which includes the eliminated state, as a basis for explanation. For instance, we have a scenario that models a user losing a borrowed resource, and hence producing a lost-resource state. From this state it is not possible to check_in the resource, nor remove it from the library, but only to mark it as lost. An IMS that does not model a lost-resource state must rely on either the lost-resource state never being reached (e.g., chaining resources to users), or on the cleverness of the staff in "getting around" the system (e.g., doing a check_in and then a remove). As discussed in section 2.1, this type of naive-modeling/getting-around correlation is pervasive in the projects we have studied.

4.7. Unneeded constraint

The next constraint described is found to be an uninteresting piece of common sense knowledge. In fact, we question its explicit presence given the use of shared knowledge earlier to avoid describing other typical concepts. Given the apparent "de-compilation" that generated the text, perhaps it is a remnant of the formal specification.

4.8. Borrowing Limits

The third and final constraint introduces borrowing limits, a more interesting concept. We have discussed the analysis of borrowing limits in section 3.2, and hence will omit further discussion here.

4.9. Wrap up

We have now come to the end of the text description. We have attempted to highlight JOG critiques, on a line by line basis. Other problems that were uncovered by JOG, using the same type of knowledge discussed in our line by line critiques, include missing resource classes (e.g., magazines, journals), lack of a borrowing time limit, no concept of reserve books nor books on hold, no discussion of security issues, missing size limits (e.g., shelf capacity, maximum number of library users), and missing queries (e.g., disjunctive and conjunctive forms).

While our library analyst did not produce many more different classes of problems than those uncovered by JOG, she was able to give a much more detailed analysis, and produce a seemingly unlimited number of examples on demand. In our view, she had 1) a deep understanding of the "first principles" of library science, 2) a knowledge of specific libraries and thus specific examples, and 3) the ability to generate hypothetical examples to illustrate some abstract principle or policy. Because this was a rather contrived problem, we did not see (nor expect to see) much in the way of interesting acquisition/control behavior. In summary, the comparison of JOG's analysis with that of our analyst

reassures us on at least two counts: 1) the representation of analysis knowledge in SKATE appears to be capable of producing the *type* of critique that we would like, if not in the amount of detail we would like; 2) our speculation that an example-based explanation component should be a key part of an automated requirements analysis system is further supported.

5. Summary

We have argued for a problem solving approach to the automation of requirements analysis. This approach is based on a cooperative effort between client and system, with both capable of contributing to the final requirements document. We have argued that such a cooperative effort must rely on a system with extensive knowledge of the application domain, as well as a model of interactive acquisition that will focus on the key details of a problem, while subsuming the mundane aspects.

To demonstrate the feasibility of the problem solving approach, we have begun to build a computer-based analyst that attempts to meet these demands. Our design of the system is based at least in part on our observations of human analysts doing requirements analysis. A demonstration system has been constructed to test two components of our larger system: a domain model representation, and a rule-based critic. The application of the system to problem #1 in the workshop problem set is discussed in the paper.

While we view the demonstration system as a successful first step in the automation of requirements analysis, it is clear that many difficult problems remain. In particular, we have yet to demonstrate an *interactive* acquisition model, nor a means for learning (or at least gracefully handling) new concepts introduced by a client. We would also like a more complete model of requirements analysis using our protocols as a basis. By the time of the next workshop, we hope to be able to report on further progress in some of these areas.

Finally we note that we have found ideas and inspiration from much good work in the area of knowledge-based software development (Green et. al., in particular, discuss the type of knowledge-based tools needed for complete life-cycle support, an automated requirements assistant being one such tool [6]). In [5] we provide a detailed discussion of our work in relation to the field as a whole.

6. Acknowledgements

Martin Feather, Jack Mostow, and Dave Wile have all provided useful and stimulating comment on KATE in particular, and knowledge-based software development in general.

We would also like to thank Sarah Douglas and Barbara Korando for their assistance in carrying out our protocol experiments.

This work is supported by the National Science Foundation under grant DCR-8603893.

7. References

(1) Barron, J., Dialogue and Process Design for Interactive Information Systems Using TAXIS, Tech Report CSRG-128, Computer Systems Research Group, University of Toronto, 1981

(2) Corbin, J., Developing Computer-Based Library Systems, ORYX Press, 1981

(3) Dietterich, T., Michalski, R., A Comparative Review of Selected Methods for Learning from Examples, In *Machine Learning: An Artificial Intelligence Approach*, Tioga Publishing, 1983

(4) Fickas, S., Design Issues in a Rule Based System, In *ACM Symposium on Programming Languages and Programming Environments*, Seattle, 1985

(5) Fickas, S., *A Knowledge-Based Approach to Specification Acquisition and Construction*, CIS-TR 85-13, Computer Science Department, University of Oregon, Eugene, Or., 97403

(6) Green, C., Luckham, D., Balzer, R., Cheatham, T., Rich, C., Report on a Knowledge-Based Software Assistant, RADC-TR-83-195, Rome Air Development Center, 1983

(7) Greenspan, S., Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, PhD Thesis, Computer Science Dept, Toronto, 1984

(8) Kant, E., Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach, Ph.D. Thesis, Computer Science Dept., Stanford, 1979

(9) KEE Reference Manual, Intellicorp, Palo ALto, Ca.

(10) Kemmerer, R. (1985), Testing formal specifications to detect design errors, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1

(11) London, P., Feather, M., Implementing specification freedoms, *Science of Computer Programming*, Number 2, 1982

(12) Mostow, J., Swartout, W., Towards Explicit Integration of Knowledge in Expert Systems: An Analysis of MYCIN's Therapy Selection Algorithm, Tech. Report LCSR-TR-81, Department of Computer Science, Rutgers University, 1986

(13) Swartout, W. The Gist behavior explainer, In *Proceedings of the National Conference on AI*, 1983

(14) Wilenski, R., Planning and Understanding: A Computational Approach to Human Reasoning, Addison-Wesley Publishing, 1983

Appendix A

begin problem statement

Problem #1: LIBRARY (R.A. Kemmerer, "Testing formal specifications to detect design errors")

Consider a small library database with the following transactions:

(1) Check out a copy of a book / Return a copy of a book;
(2) Add a copy of a book to / Remove a copy of a book from the library;
(3) Get a list of books by a particular author or in a particular subject area;
(4) Find out the list of books currently checked out by a particular borrower;
(5) Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff and ordinary borrowers. Transactions 1, 2, 4 and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The data base must also satisfy the following constraints:

(1) All copies in the library must be available for checkout or be checked out.
(2) No copy of the book may be both available and checked out at the same time.
(3) A borrower may not have more than a predefined number of books checked out at one time.

end problem statement