CIS-TR 85-13

A Knowledge-Based Approach to Specification Acquisition and Construction

Stephen Fickas

Department of Computer and Information Science
University of Oregon

Summary

This work is concerned with the automation of the software specification process. Specification automation research to date has concentrated on non-interactive translation of *complete, correct*, but informal problem descriptions into formal specifications. The work proposed here is unique in that it addresses the construction of complete, correct informal problem descriptions from *incomplete*, often *incorrect* user descriptions. The proposed system will rely on interactive problem solving and large amounts of domain specific knowledge to carry out this process.

Our work centers on a system that interacts with a user to elicit the details of a problem in a specific domain. It attempts to incorporate certain skills found in expert, human, domain analysts. These include 1) the ability to refine a sketchy, incomplete problem description into a complete form, 2) the ability to recognize known examples during problem acquisition, and 3) the ability to critique a user's description in terms of missing detail and lack of coverage. To realize these skills, we propose using the problem solving system Glitter to represent refinement knowledge, a combination of frames and state-transition diagrams to model the domain itself, and the ORBS rule-based system to implement knowledge-based symbolic evaluation.

Finally, the system will automatically translate (i.e., compile) the informal problem description into a formal specification. Our current target is the Gist specification language, although other languages are under consideration.

Table of Contents

Proposed Renewal of NSF Research Grant:

A Knowledge-Based Approach to Specification Acquisition and Construction

Stephen Fickas

Computer Science Department
University of Oregon
Eugene, OR. 97403

1. Introduction

We are now in the second year of a two year research grant (MCS-8312578) funded by the National Science Foundation. In this paper, we will review our original research goals, discuss our progress on achieving them, and finally present a research plan for continued support.

2. The Specification Problem

We are concerned with automating the construction of software specifications. Our model is one of a human client requesting help in specifying a problem for computer solution. We are attempting to build a computer-based system that will *interact* with the client to acquire the details of the problem, and finally produce a formal specification in an existing specification language[1]. We expect such a specification will be translatable to compilable code by technology on the horizon [4, 40]. However, the translation process itself lies outside of the concerns of this proposal.

In our original proposal, we made detailed arguments for why it was important to mechanize software specification. Since that time, others have made similar arguments (c.f. [2, 4, 7, 9, 26]) so we feel we can omit further motivational material here; we do review the major points of our original proposal in the next section. It is important to state explicitly, as we did in the original proposal, that our current research and proposed continuing research rests on much solid groundwork, some of it ours and some of it others. We now have been studying formal specifications for 6 years. Further, for the

[1] Lehman characterizes this as the *abstraction process*, which transforms an application concept into formal specification [32]. It also sometimes falls under the heading of Requirements Analysis (see the discussion of KBSA in section 9.1).

past two years we have been informally studying the software specifications coming out of the two quarter, Senior project course at the University of Oregon (approximately 20 projects total, each of 1.5 man/years effort). Out of this work, and related research projects discussed in section 9 and appendix D, has come a realization that automation should not start with the formal specification process, but instead with the problem acquisition process. In particular, we need the machine to help us figure out what we want it to do. A formal specification is the result of pondering, circling, going too far, covering too little, refining, and finally zeroing in on the problem to be solved. It is these processes that we focus on in this proposal. We will argue that to build a system that assists problem acquisition will require a thorough understanding of the problem domain.

The remainder of the proposal will further define the problem, and present a plan for studying portions of it.

3. Our Original Proposal (and what we still believe)

We can summarize our original proposal in terms of four themes:

(1) Reuse is important. Complex domains lead to a myriad of details in a formal specification. We must have a way to avoid regenerating them from scratch on each new effort.

(2) Building a specification can be viewed as a problem solving process. Hence, an automated system must deal with construction goals and methods for achieving them.

(3) To tackle interesting specifications, an interactive system is needed. Here we rely on the user to supply insightful reasoning, and the system to take care of the many mundane details.

(4) We should rationalize all of the products of the software development lifecycle. This includes the specification. With new automation techniques, the specification becomes the keystone of maintenance. We must know how the specification got to be in the state that it is in, and hence what ramifications we can expect by changing it.

In our original proposal, we proposed using a combination of Gist schemas and the Glitter problem solving framework to implement these ideas[2] [18, 21]. After working on

[2]Gist is a formal, operational specification language based on a relational database with spontaneous computation, constrained non-determinism, and historical reference [33]. Glitter is a problem solving system that allows the definition of goals, methods for achieving those goals, and selection rules for choosing among competing methods [20].

the project for one and a half years, we still believe each of the four arguments above. However, we also believe that we underestimated the type and amount of *domain* knowledge necessary to address these issues. We believe now that we need knowledge much more akin to that of an expert domain analyst or system analyst. When taking this view, several differences became apparent:

- We should be working above the level of a formal specification language. The formal specification should be the output of our system, and the language a pluggable backend. We are interested in problem acquisition first, and only then statement in formal terms. In particular, we expect the system's choice of specification language to formally state the problem will vary with the problem itself. Thus, it may choose Gist for embedded systems, Draco [35] for business applications, or even a rule-based language for AI problems (see section 4.1 for some of our preliminary work in this direction; see also [14]).

- While we still view the specification process as a problem solving task, we now believe that the control is much more complex than the simple goal/subgoal approach of Glitter. This is particularly acute due to our reliance on interactive problem solving.

- If our system is to model a domain analyst, then it must have deep knowledge of the domain. Our earlier work relied upon a syntactic knowledge of Gist schemas. We found that this was not enough knowledge to reason about completeness, consistency, or ambiguity bugs. We are now in the midst of defining and constructing a domain knowledge representation that will allow us to support example generation, specification criticism, and problem simulation.

  We also note that this extends our "assistant" approach. We now expect our resulting system to not only take care of mundane problem details, but also to be expert in constructing specifications for particular domains.

- What we are reusing is domain analysis as opposed to formal-specification writing. We have not lost the notion of a catalog of previous efforts. We have just made them concrete instances in our model. Once integrated into the domain model, they are 1) rationalized by links to more general domain concepts, and 2) available for various expository tasks, e.g., showing as examples.

- An expert analyst knows more than the domain. He or she knows what is hard and easy in implementing programs in various languages and on various hardware. In other words, the analyst has a foot in the software development world as well.

We are advocating a move to a true knowledge-based approach. That is, our proposed system will have large amounts of knowledge on specific domains.

In one sense then, this proposal is in the spirit of continued work on specification automation: we are still concerned with the construction of formal specifications. In another sense, it shifts attention away from formal specification writing and towards problem acquisition. In particular, the problem becomes less one of *translation* from informal to formal, and more one of *acquisition* of the problem in the first place. Given this shift, a new set of attendant problems appear, many concerned with the inability of human users to know exactly what they want and exactly how to describe it when they first sit down in front of the machine.

Our new work will build on 1) results of our past work, including lessons learned, and 2) the expertise of the project personnel -- four faculty members and three graduate students -- listed in appendix A. Taken together, we argue that this forms the critical mass necessary to do research in this area. The next section discusses our work over the last 18 months. Subsequent sections describe our proposed new work.

4. Summary of Results

We are now in the second year of a two year research grant, MCS-8312578. Our original proposal was titled "The Mechanization and Documentation of Software Specification". In this section, we will discuss what we have accomplished on the project from the start of the granting period, April '84, until November '85, approximately a year and a half.

We believe there are three principal points to make in this section. First, we have done a substantial amount of work in building tools that allow us to represent and use specification knowledge. These tools were useful in our initial work, and remain useful in our continued work. Having them in hand allows us to build and experiment with prototypes much more readily. Second, we have learned a great deal from our past efforts. These lessons may be the most critical part of our results. We will refer to them frequently throughout the proposal. Third and finally, we have begun to work on parts of our new system, which we call KATE[3]. This is work that is not referenced in our original proposal; it comes from our switch in focus to a knowledge-based approach. It is interesting in the sense that it gives us a small jump on the work proposed in subsequent sections, and allows us to ground portions of this proposal using partial results, e.g., we have built a simple boundary-condition generator, we have begun to represent portions of the conference domain, we have been able to map small portions of a problem description to code.

Below, we will discuss these three aspects of our results.

[3]Knowledge-Based Acquisition of Specifications.

4.1. Tool building

Our original goal was to use the Hearsay III system [16] to implement and test our ideas. Fickas' thesis program, Glitter [20], is implemented in Hearsay III. We planned to use Glitter as the repository of problem solving aspects of the software specification process. After several unsuccessful attempts, it became clear that Hearsay III would not easily transfer to the Symbolics 3600. Hence, we began a major effort to reimplement Hearsay III on our Symbolics. In the tradition of the Hearsay lineage (versions I, II, III), we decided to use the good ideas and throw out the bad ideas in building the new system. We also paid attention to the user interface to our new system. Out of this work came the Oregon Rule Based System [19, 22, 23], or ORBS for short[4].

ORBS has proven useful in several ways. First, and least relevant to this proposal, it has proven to be a good language for writing AI programs. It provides a flexible model of control ala Hearsay-III with tools to support the construction of control strategies (e.g., a catalog of control cliches for composing a control strategy, a control strategy animator for debugging). ORBS is objectized: multiple instantiations of the system can be extant at any time, each with its own rules, database, and control (see [19] for a further discussion of the features of the system). ORBS is currently being used by research projects within our department to implement an ICAI system, a learning system, a VLSI CAD system, and a distributed problem solving system. We have also honored requests for copies of the system from several research projects outside of the university.

Second, we have begun to use ORBS to implement portions of our new system. This includes a re-implementation of Glitter in ORBS that supports abstract refinement (see section 7.1), a "compiler" written in ORBS that produces ORBS code from problem descriptions in the system (see next paragraph), and a boundary-condition generator for testing portions of a problem description (see section 5).

Finally, we have begun to use ORBS as an implementation target for our work in specification. So far, mapping from specification to ORBS code has been a manual task, although a graduate student, Bill Robinson, is working on a transformation-based compiler (written in ORBS) that translates process diagrams in KATE (see section 4.3) to ORBS code[5]. We agree with Swartout&Balzer [46] and Barstow [6] that it is important for the specification process to be tied into an implementation process; certain types of specification bugs can only be found by actually trying to implement a system. In this sense, ORBS performs the same function as TAXIS does for Greenspan's RML [27], and in some sense what WILL does for Gist [4].

[4]We decided to stop the line at III, and resist the temptation to name the system Hearsay IV.

[5]This is not to imply that this is a solved problem. Robinson's work tackles only a small part of the implementation problem. See Balzer's perspective [4] on the general automatic compilation problem for a good view of how broad and deep the problem is.

Our other major tool building effort has been in the area of graphical interfaces to our knowledge bases on the 3600. We now have a collection of graphics packages that can be (and have been) combined to form KATE tools. Two of these packages form the interface to the tools discussed in section 4.3.

4.2. A knowledge-based approach to specification

Our early efforts were focused on carrying out our original goals: build a catalog of Gist schemas; retool Glitter to handle specification refinement problems. For the latter, we constructed a set of goals, methods and selection rules for automating Goldman's narrative development of a baseball specification [25]. This Glitter system at least partially validated Goldman's original speculation:

> "Although we have not yet done so, it seems plausible that the development steps stated in English in this paper could be formalized directly in terms of functions mapping processes to processes rather than as mappings from specifications to specifications. In that case, the initial specification and sequence of structured modifications would present a complete definition of the final process and would arguably be both easier to produce and easier (for a person) to comprehend."

In our system, the user could post refinement goals (e.g., handle special cases, add attributes, break out actions), and the system would attempt to find methods for achieving the refinement. The construction of this system lead us to an impasse. To build the types of sophisticated methods needed to automate the process[6], we found that we needed to represent and reason about domain knowledge. We also found problems with Glitter's rather rigid control strategy of following goal/subgoal chains until completion. As Goldman notes,

> "It appears that the best way to achieve a change along one dimension [of refinement] may involve making a change along one of the other dimensions."

That is, a user will often want to suspend the current task to work on something else, e.g., suspend working on structural detail to work on coverage of special cases.

During this same time, we were attempting to construct a catalog of carefully crafted schemas for several domains that we had been studying, i.e., transportation, resource management within a Computer Science department. These schemas were actually Gist skeletons with holes to be filled in by the user. A problem with this approach was the lack of composability. What specification writers wanted was the ability to build-by-

[6]If the system is unable to find a method, it asks the user to step in. This is a very manual process with the system acting mostly as recorder of the user's steps.

pieces. A further problem was the representation and management of these skeletons. We began to see the need for an entire other representation sitting ontop of these schemas that would organize and rationalize the catalog.

The culmination of problems associated with both of the above efforts caused us to reevaluate the type and amount of knowledge needed to build specifications in a particular domain. We concluded that we would need more domain knowledge, and a better representation of it [24]. While we still believed that examples from the domain are important starting points, we abandoned the use of Gist schemas to represent them. We also saw the need for more flexible control in Glitter. As discussed in the next section, we have begun to work on portions of these problems.

4.3. Current work: representing domain concepts

We are building representations for objects, actions, and constraints for the domain of conference organization. We briefly discuss our work on each below.

Our original proposal called for representing domain objects using Gist's typing mechanism. Attributes would be represented in relational form. We have now moved to a frame-based representation. We have constructed an object editor MOSS [24] that allows a Strobe-like [39] representation of class-attribute hierarchies to be defined and modified graphically. Multiple graphs may be in the editing buffer at any one time. The editor allows graph-to-graph linking, sub-graph manipulation, and primitive node and arc modification commands. As with Strobe, each frame slot is further represented by a *facet*. Facets can be used to place datatype restrictions, and attach access and assignment procedures. We have integrated ORBS into MOSS by allowing an ORBS object[7] to be attached to a slot. Figure 1 shows MOSS with the facet for the slot *interests* expanded.

Using MOSS, we have built up object hierarchies for resource management within our department (see figure 1), and for a general transportation domain. We are in the process of building up objects in the conference organization domain, which, as discussed in Appendix B, nicely shares our previous efforts.

We note finally that our mixture of frame-based and rule-based representation has much in common with the PHI-NIX project at Schlumberger-Doll Research Lab [6], discussed in more detail in appendix D. We find this comforting since PHI-NIX also attempts to represent large amounts of domain knowledge, in their case applied to Automatic Programming.

[7]The user can instantiate ORBS multiple times.

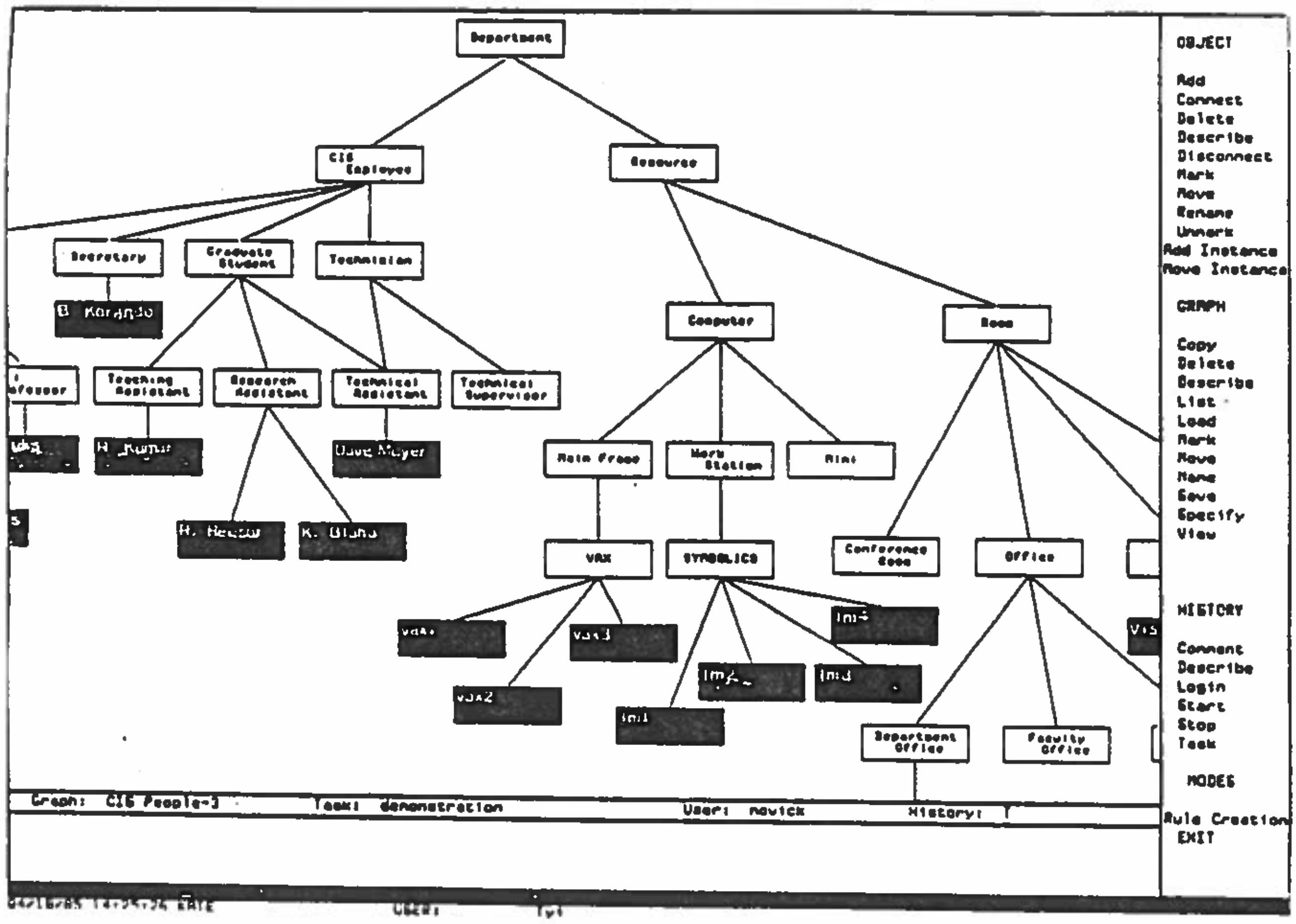
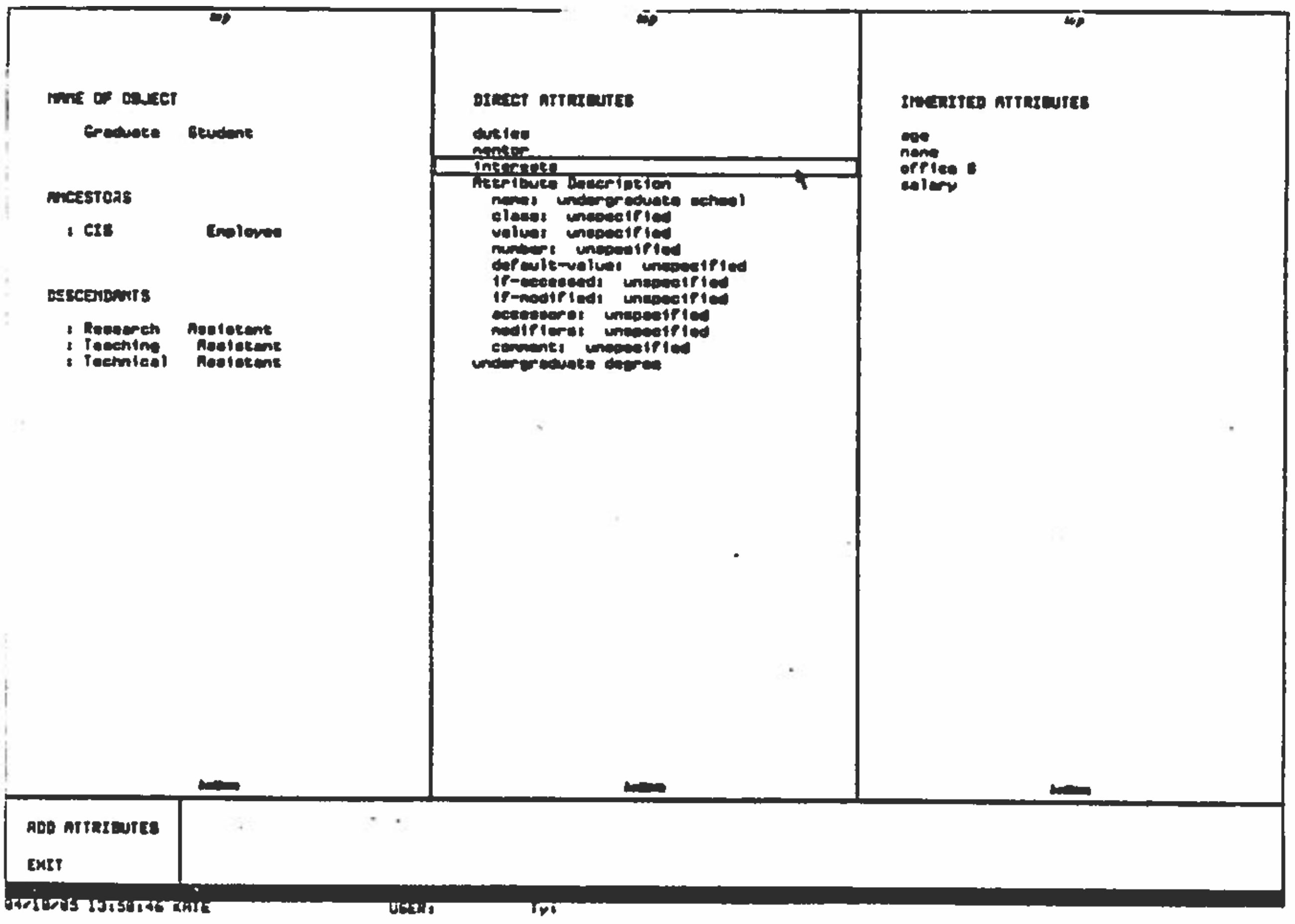

Department
CIS Employee
Resource
Secretary
Graduate Student
Technician
B. Korando
Professor
Teaching Assistant
Research Assistant
Technical Assistant
Technical Supervisor
H. Kumar
Dave Meyer
R. Rector
K. Bland
Computer
Room
Main Frame
Work Station
Mini
VAX
SYMBOLICS
vax4
vax3
vax2
lm1
lm2
lm3
lm4
Conference Room
Office
Department Office
Faculty Office
OBJECT
Add
Connect
Delete
Describe
Disconnect
Mark
Move
Rename
Unmark
Add Instance
Move Instance
GRAPH
Copy
Delete
Describe
List
Load
Mark
Move
Name
Save
Specify
View
HISTORY
Comment
Describe
Login
Start
Stop
Task
MODES
Rule Creation
EXIT
Graph: CIS People-3
Task: demonstration
User: novick
History: T

Figure 1



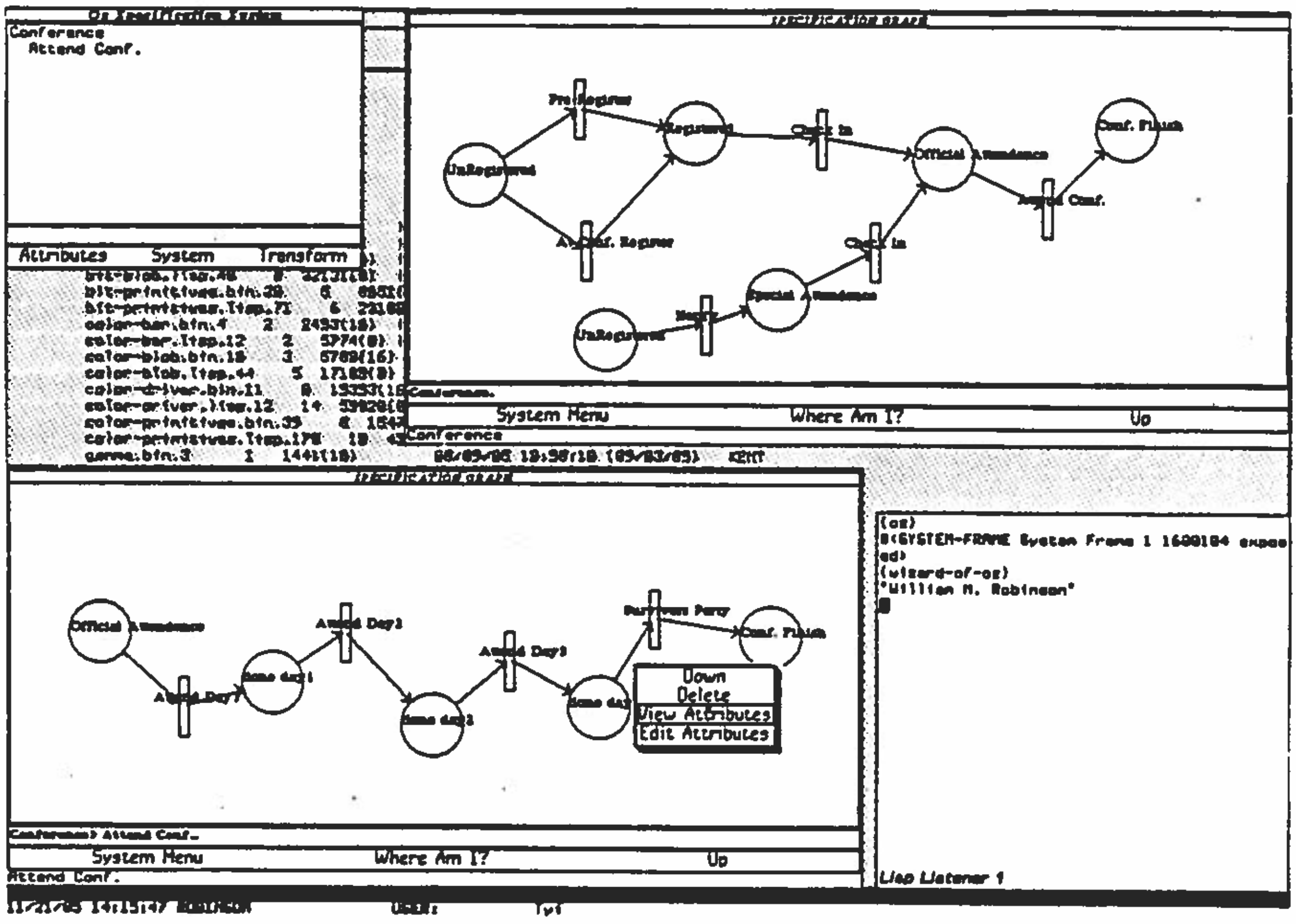
```
NAME OF OBJECT

    Graduate   Student

ANCESTORS

  : CIS          Employee

DESCENDANTS

  : Research   Assistant
  : Teaching    Assistant
  : Technical   Assistant
```

```
DIRECT ATTRIBUTES

duties
mentor
interests
Attribute Description
   name:  undergraduate school
   class:  unspecified
   value:  unspecified
   number:  unspecified
   default-value:  unspecified
   if-accessed:  unspecified
   if-modified:  unspecified
   accessors:  unspecified
   modifiers:  unspecified
   comment:  unspecified
undergraduate degree
```

```
INHERITED ATTRIBUTES

age
name
office #
salary
```

```
04/18/85 13:58:46 KATE          USER:          Ty:
```

Figure 2



```
bit-blob.ltap.48        8   2213(18)
bit-primitives.bin.28     6   6851(
bit-primitives.ltap.71     6  2318
color-bar.bin.4       2   2493(18)
color-bar.ltap.12       2   5774(8)
color-blob.bin.18      3   6789(16)
color-blob.ltap.44       5  17189(8)
color-driver.bin.11      8  15353(18
color-driver.ltap.12    14  53828(8
color-primitives.bin.35    6  1547
color-primitives.ltap.178   18  4
game.bin.3        1   1441(18)
```

```
08/09/85 19:38:18 (09/03/85)    KENT
```

```
(oz)
#<SYSTEM-FRAME System Frame 1 1600104 expos
ed>
(wizard-of-oz)
"William M. Robinson"
```

```
11/21/85 14:15:47 ROBINSON        USER:        Tyi
```

Figure 3

In our original proposal, we planned to use Gist demons and actions to represent states and events. We have switched to a a more explicit, graphical, state-transition representation. We have built a process-diagram editor called OZ. With OZ, a user can define and edit a state-transition diagram (which shares attributes of a petri-net; c.f. [10, 43]). Actions are currently represented in rule-based form, a prototyping measure that we believe is not powerful enough to handle the type of conference events we wish to model. We expect that a form closer to Conceptual Dependency representation [37] will better model goals, intentions, and other social aspects of conferences. We are also attracted to the use of the Script-like structures that TAXIS uses to represent transitions in similar process diagrams [28]. Figure 3 shows an example from OZ taken from the conference organization domain. As shown on this figure, a transition can be further refined into a more detailed state-transition diagram (TAXIS [28] allows a similar abstraction hierarchy). This is one of the functions required to implement temporal refinement discussed in section 7.1.

We have moved from a Gist-based representation of constraints to a rule-based representation modeled on that of AP5 [11]. This leads to a two part constraint: a pattern that states the constraint; a method for reinstating the constraint when it becomes violated. There are both plusses and minuses to such a representation. On the plus side, it models well the types of if-then descriptions that users often employ, e.g., "if the list of city-tour attendees is below k, cancel the event", or "if the banquet list becomes full, place people on a waiting list". On the minus side, these are actually *implementations* of pure constraints. The effect of this is that users must always state their constraints in this if-then form. The bottom line is that we do not expect to do further work on constraint representation (i.e., we plan to use what we have) until the research problems presented in section 7, the heart of this proposal, are solved.

4.4. Protocol analysis

One thing lacking in our original proposal was the running of protocol experiments on expert, human, specification writers. We have been informally studying students in this role over the last two years as part of our two-quarter, Senior projects course; in particular, the course has attempted to delineate good analysis techniques and pass them onto students. As part of this teaching process, we have also brought in local experts involved in specifying software for industrial applications. We now feel ready to formally study the problem acquisition process. We are currently designing a set of protocol experiments using subjects from academia and industry. Professor Douglas is assisting with experiment design, and will also collaborate on running the experiments and analyzing the results. Appendices A and C discuss the facilities and personnel involved in these experiments. As we discuss in section 6, we expect the results of this work, among other things, will lead to a better view of interactive control in KATE.

4.5. Published papers

From the start of the grant (April '84) until present (November '85), members of our group have attended 4 Conferences and a Knowledge-based Design workshop at Rutgers. During the same period, we have published the following papers related to the project:[8]

Fickas, S.
Mechanizing software specification,
In *Workshop on Models and Languages for Software Specification*, Orlando, 1984

Fickas, S., Laursen, D., Laursen, J.,
Knowledge-based software specification,
In *Workshop on Knowledge Based Design*, Rutgers Univ., 1984

Fickas, S., Novick, D., Reesor, R.
An environment for building rule based systems,
In *3rd Annual Conference on Intelligent Systems and Machines*, 1985

Fickas, S., Novick, D.
Control in rule based systems: relaxing restrictive assumptions,
In *5th International Conference on Expert Systems and Their Applications*, 1985

Fickas, S.
Design issues in a rule based system,
In *ACM Symposium on Programming Languages and Programming Environments*, Seattle, 1985

Fickas, S., Downing, K., Novick, D., Robinson, W.
The specification, design, and implementation of large knowledge-based systems,
Invited paper, *IEEE Annual Northwest Conference on Computer Science*, Portland, 1985

Fickas, S.
A problem solving approach to software development,
In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11 Nov. 1985

Simoudis, E., Fickas, S.
The Application of Knowledge-Based Design Techniques to Circuit Design,
In *IEEE International Conference on Computer-Aided Design*, 1985

We have also produced the following technical reports:

[8]As requested, we've included all papers that have acknowledged the support given by NSF.

Fickas, S.
The Oregon Rule Based System (ORBS)
Tech report CS84-5, CS Dept, U of Oregon, 1984

Fickas, S., Novick, D., Reesor, R.
Cataloging Control Strategies,
Tech report CS85-2, CS Dept, U of Oregon, 1985

4.6. Theses related to the project

There have been four Masters theses completed on our project. Two PhD students are working on KATE as part of their dissertation work. Details are given in appendix A.

4.7. Projected results

By the end of the current granting period (April '86), we expect to have completed our protocol experiments. We also expect to have a reasonable portion of the conference domain model completed.

5. A System for Acquiring and Constructing Specifications

Before presenting our new proposal, we will summarize the results that lead to it. Our original goal was to use the GIST language from ISI [4] as a basis for our work in specification construction. We spent part of the first year attempting to build abstract Gist schemas for resource management and transportation domains. During this effort it became clear that Gist was not the right representation for storing knowledge about the specification *process*. What we needed was a representation that captured the objects and operations concerned with building a specification. In particular, our original notion of reusing specification knowledge by storing away abstract Gist specifications in a schema catalog proved infeasible. For one, the Gist schemas did not capture enough information, e.g., why a construct was present, what would happen if it was removed or changed. While we could answer these questions at a syntactic level, we were not able to generate answers in terms of the domain itself. In essence, our reasoning was limited to a rather superficial level. In summary, the catalog of Gist schemas was at once too weak (it did not easily represent development concerns) and too powerful (we could not reason about it effectively).

At the same time that we were becoming dissatisfied with our original Gist schema proposal, we were coming to realize that there was more to the specification process than simply giving a user a set of refinement-based editing commands, even if they were part of a problem solving system such as Glitter. Specifically, it appeared that expert human domain analysts combined skills of listening, setting up analogies, selling (i.e., guiding the user to a particular view), summarizing, paraphrasing, and testing. They brought this all off by having a good understanding of the application domain, and extensive experience in specifying problems in the domain. We found that such experts used conversational techniques such as the following[9]:

- *"What you have described sounds a lot like a foo. How is it different?"*. Here foo is a well known concept in the domain.
- *"Let me make sure I understand this: your problem is ..."*.
- *"I do not understand what happens in this case: ..."*. The analyst sets up a hypothetical situation for the client.
- *"Case fum is usually a sticky problem in these type of programs. We should discuss what to do about it."*
- *"You have specified that you want to have action z available. This type of action is often costly to implement in this domain. Do you still want it?"*.
- *"Let's talk in more detail about y"*.

These observations, along with the problems discussed in sections 3 and 4, lead us to reconsider what knowledge was required in building specifications. We now see the following type of knowledge as necessary:

- *Knowledge of the application domain.* Terms and concepts: can map from client's descriptions to known cases; can use known cases in presenting examples. Special cases, error and boundary conditions (major causes of specification completeness bugs). Beyond simply representing the objects, actions, and constraints of a domain, difficult questions include what does it mean to be a prototypical example, how can new domain concepts be added by the user, how can the system modify existing domain concepts to fit the new problem.
- *Knowledge of the specification process.* Our model is one of an initial, incomplete, buggy problem description being transformed into a complete, correct, formal specification. Difficult problems abound, e.g., what abstraction techniques are useful, how can the myriad details of a problem be elicited from the user, how does the domain analyst fill in missing detail, how is a specification verified with the

[9]We reiterate, our empirical evidence to date is by informal observation. We are in the process now of verifying our results by formal protocol experiments, as discussed in section 4.4.

user.

- *Knowledge of the program development process.* In general, what is hard and easy to do. For example, in many applications, an undo capability adds at least an order of magnitude of effort to design and implementation. Also, novel systems lead to higher design and development costs (c.f., [1]).

- *Knowledge of the production environment.* This includes the implementation language, host machine, user skill levels, and real-time constraints. Each of these may have a strong influence on what is feasible to implement and use. For instance, if an on-site PC is to be used at the conference (as opposed to a terminal to a remote mainframe), then certain interactive operations may be infeasible.

- *How does the expert learn from experience?* Each new specification construction effort should lead to some new insight into the specification. How can this be assimilated so to be of use to the next client? Simply integrating a new example into the database is not too hard. Learning why it was a good or bad session -- the user went directly to the problem, the user spent a lot of effort in backtracking -- is much more difficult.

Our ultimate goal is to build a system that uses each of the above types of knowledge to acquire a problem and produce a formal specification. It should be interactive, and based on a control mechanism that carries on a natural dialog with the user. As we shall emphasize several more times in this proposal, we are not proposing to bring all of this off in a three year time span (the length of our proposed funding period). However, it is important to note the long range goals of the project. The next section shows how a user will interact with KATE. Section 7 discusses the underlying mechanisms we propose.

6. System Behavior

In the last section, we itemized the types of knowledge our system will need. We have yet to describe how it will use that knowledge to build specifications. To incorporate the skills of an expert domain analyst, we would expect a system to carry on a mixed-initiative dialog with the user in natural language. We might expect the user to drive the dialog when he or she was describing new portions of the problem or refining existing portions. We might expect the system to drive the dialog when pointing out similarities with other problems it has seen, when trying to fill in missing detail, or when pointing out bugs in the current description. While these are our long term goals, this proposal tackles only a subset of them. In particular, we expect that the dialog will be user-driven as opposed to mixed-initiative. The system will present the user with three basic tools with which to work:

(1) An object/class editor called MOSS. A snapshot of MOSS is shown in figures 1 and 2.

(2) A state-transition diagram editor called OZ. A snapshot of OZ is shown in figure 3.

(3) A specification-refinement editor implemented in Glitter [20].

None of these tools use a natural language interface; a user's actions are mouse-and-pointer operations with MOSS and OZ, and command-driven interaction with Glitter. The system will take control when it is filling in pieces or reporting interesting cases and/or bugs to the user. Other than this, it will be up to the user to direct the system to the next specification task.

We have chosen this approach for a practical reason: we do not believe we can solve the research problems we have set out (see section 5) *and* the interface problem *and* the control problem in the same three year time frame. We do expect to investigate certain aspects of control (see Professor Dehn's research description in Appendix A). Using our experience (see Professor Douglas's research description in appendix A) and the results of our protocol experiments, we also expect to make improvements in the interface. However, we do not expect to attain system-driven control nor a natural language interface within the given time frame.

In summary, we propose a system with the following capabilities, each of which is discussed more fully in subsequent sections of the proposal: 1) *Acquisition* of new objects, processes and constraints, 2) *Recognition* of existing objects, processes and constraints, 3) *Refinement* of sketchy, incomplete descriptions, and 4) *Critiquing* of the current specification. Both 2 and 4 are system processes that are constantly monitoring the user-driven processes of 1 and 3.

6.1. User/System interaction

We will give several short examples here of how the user interacts with the system. It is important to note that our use of natural language is as a meta-description of what's really going on, i.e., the user selecting various options through menus, adding objects using MOSS, editing diagrams using OZ.

Interaction with the system will take on, alternatively, one of two styles during problem acquisition. In the first, the user gives commands to the various editors to carry out descriptive or refinement actions. Hence, he or she might add objects/classes to the given conference class hierarchy, add or delete conference actions, or state general development goals.

As these actions are carried out, KATE checks several things, each of which may cause the system to take over the dialog. First, it attempts to match the current problem description against known problem instantiations in the domain. Thus, if the user describes a conference that has a small number of attendees, long question answering periods, and a minimum of conference staff, KATE will let him or her know that it has the concept of *workshop*, which at least partially matches those specifications. The user can accept *workshop* outright, accept it with modifications, or reject it entirely. Acceptance here allows a host of details to be implicitly specified; they come with the workshop concept.

Second, if the user adds a new object, event or action, the expert attempts to represent it in known domain terms. For instance, if the user added a new event, *survivor's party*[10], the system would attempt to understand it in terms of known conference events:

> **System:** You have added a social event "survivor's party". I'd like to try to characterize this event. Where is it held?
>
> **User:** conference chairman's house.
>
> **System:** who goes to such an event?
>
> **User:** conference attendees.
>
> ...

In this way, the user is not hamstrung to just the concepts initially provided by the system. On the other hand, if the user attempts to add a new concept that cannot be viewed in terms of conference primitives, then while it will be recorded (under *unknown-events*), it will not become part of later reasoning (see below).

Third and finally, the system attempts to follow the ramifications of the user's actions. Here we are interested in two particular actions: modification of existing concepts; addition of new concepts.

Modification of an existing concept: Suppose that the user deletes an object, event or action. This may have consequences in the rest of the specification. For instance, assume that the user accepts KATE'S suggestion of viewing his or her conference as a workshop. Along with this comes a particular process view:

> **System:** Let me show you what the "workshop process" is in my view:

[10]As discussed previously, new objects and events are added using MOSS. We assume that the user classifies survivor's party as a social event by placing it under the appropriate class.

[Here the system displays the workshop process using OZ (see figure 3).]

User: *[deletes on-site registration action]*

System: let me make sure I understand what you want. You have deleted the action of registering at the conference. This implies several things:

1. No person will be able to attend unless they have pre-registered.
2. There will be no need to handle money or registration forms at the conference.

[The deleted action, on-site registration, was one of two routes to attending the conference. The system lets the user know that only one remains (in reality by highlighting portions of the state-transition diagram). Further, the deleted node had a more detailed description that broke on-site registration into smaller actions of handing out registration forms and collecting money. The system confirms that these actions should go away with their deleted parent (again by appropriate highlighting).

In general, we find that expert analysts do frequent misunderstanding checks. In this case, we want to make sure that the user is aware of the implications of his or her actions. We believe that this type of post-analysis of specification changes will be one of the most powerful functions we can provide regarding specification correctness.]

System: Is this correct?

User: Almost. There is a special case: local VIP can attend without registering.

[The user is reminded of a special case. Note that reminding goes both ways: the system is reminded of useful examples as it is "listening" to the user's problem; the user in turn is reminded of special cases by being informed of implications.]

Addition of a new concept: On addition of a new concept, the domain expert will attempt to integrate it into the existing description. For a new event, like survivor's party, KATE will attempt to simulate conference attendees getting to the event, entering the event, attending the event, exiting the event, and getting back from the event. This is part of the expert's built in knowledge of what happens at conference events. In attempting this simulation, further questions will arise regarding properties of the event, e.g., maximum and minimum limits on attendees, its duration. If a "bug" is found in the current problem description, e.g., there is no transportation to the event, then the user is made aware of it.

If instead of an event, a new object is added, the expert will attempt to move that object through the conference process. For instance, we have seen the addition of the concept of *local VIP*. KATE will first classify (through user interaction) local VIPs as conference attendees, and then verify that they can perform the same functions as normal attendees, e.g., get past the guard at the door, attend sessions, attend social events. Again, if a bug is found here, e.g., there is no way for the guard to identify them, the user is made aware of it.

In the next section, we will discuss the mechanisms that will support the type of interaction presented above.

7. The Underlying Methods

There are three skills of an expert domain analyst that we believe are key in bringing about the system behavior proposed: 1) taking a problem description from an abstract, sketchy, and often buggy version to a refined, detailed, complete specification[11], 2) using past experience with problems in the domain for both acquisition and explanation of new problems, and 3) using simulation or symbolic execution to disambiguate, criticize, and understand problem descriptions. Note that these skills are based on system/user interaction. In particular, we are not interested in, nor do we think it is reasonable to expect, batch processing of problem descriptions, English or otherwise.

We propose building a computer system that embodies these three skills. After interacting with a user to obtain a problem description, the system will produce a formal specification in a language such as Gist[12].

We are now building a domain model of conference organization (Appendix B discusses our choice of this particular domain). This will act as the testbed for studying the KATE system. The next three sections discuss the three general methods we will need.

7.1. Problem refinement

Goldman [25], Feather [17], and Adelson&Soloway [1], among others, have proposed a model of gradual refinement when designing complex artifacts. Taking the view that a formal specification is such an artifact, Goldman lays out three headings under which refinement can be viewed:

[11]Words like *complete, correct, buggy* are relative when discussing specifications. For instance, a specification bug could involve inconsistency, in some sense a real bug, or lack of coverage of some special case, more of a conceptual bug.

[12]We do not want to commit to any particular specification language at this point. We would hope to be able to build pluggable language-based generators for various languages.

- *Coverage:* the cases covered by the specification.
- *Structural:* the amount of detail in a state.
- *Temporal:* the amount of change between states revealed by the specification.

For Goldman's example, that of Baseball, we were able to "Glitterize" the development. That is, we came up with a set of specification development goals, a set of refinement methods for achieving the goals, and a small set of selection rules for selecting among competing refinement strategies. As discussed in section 4.2, we learned three lessons from this effort:

(1) Glitter must use domain knowledge to carry out refinement. This will require a better integration of Glitter with the existing knowledge representation tools of the system, e.g., MOSS, OZ.

(2) We cannot assume that the user has a complete grasp of the many nuances of a problem, and is just looking for a way to get it all into the machine. Instead, we expect 1) the system to have knowledge about the special cases and error conditions in a domain, and 2) the system to drive the refinement, when appropriate, using this knowledge. This is not a change to the Glitter model per se, but a change in the way it used, now by both user and system.

(3) Perhaps the most difficult, we see the need for a more flexible model of control during refinement. Some of Wile's work on PADDLE [48] seems to fit here, and we hope to benefit by it. However, this problem of control is one we plan to put off until substantial progress is made on other aspects of our system. Section 8 discusses this in more detail.

In summary, we still plan to use Glitter as the basis for our problem solving view of refinement (discussed and motivated extensively in our original proposal). However, we will need to better integrate it into our proposed work on problem acquisition.

7.2. Domain cliches

We have argued that it is ludicrous to start from scratch when building a problem specification. Our original proposal called for a catalog of carefully crafted schemas or templates to be available to a user when starting a new specification. The user would choose one as a starting point, and then tailor it to his or her needs. In section 4.2, we discussed our problems with using such an approach. In its place, we propose representing a predefined, known-to-be-useful (as gathered from domain experts) set of conference concepts. We will refer to these as *domain cliches*. Some examples seen in this section include *workshop, hail-and-farewell, unsigned check, first-come-first-serve.*

As a user is describing his or her problem, the system is doing a continual pattern-match against the domain cliches. On a match, the system will ask to use the cliche as a piece of the specification. The user is free to accept the cliche (with modifications if necessary) or reject it. The final specification may be any mixture of domain cliches and user generated descriptions. As an example, the user and system were beginning to characterize "survivor's party" in the previous section. After entering information about the location and time, we might expect the system to respond

> **System:** this is beginning to sound like what I call "hail and farewell". Would you agree?
>
> [*System shows its hail-and-farewell event using OZ and MOSS.*]
>
> **User:** yes.

The user may accept this event (as is done above), and hence implicitly accept a myriad of details about the event that would normally have to be specified bit by bit. The user may instead use MOSS to modify the event to fit some special case for his or her survivor's party, accepting the remaining portions as is. Or the user may simply reject the event, and continue filling in details about her or her survivor's party.

There are two research issues lurking here. First, how will we represent cliches? In our original proposal, we proposed representing this type of information as a catalog of skeleton Gist specifications. In KATE, we retain the catalog notion, but stock it with groupings of MOSS and OZ objects. In essence, our goals are quite similar to that of the Programmer's Apprentice project, which uses programming cliches to build up a program [47]. Using a *plan* mechanism, the Programmer's Apprentice allows cliches to be composed. We also must allow composable objects. Thus a *workshop* is just a collection of existing sub-pieces (possibly including other cliches) from the conference domain. In essence, some grouping of conference objects are known to be useful and common; these become the cliches of our system. Other groupings can be constructed by the user. Together, they allow familiar problems to be described with minimal effort, and novel problems to be described, but with correspondingly more effort.

The second research issue centers not on the representation, but the use of domain cliches. There are at least three questions relating to this:

(1) Are exact matches necessary? Or, should some elements give a higher "matching weight" than others? It seems clear that the user's description of the survivor's party should match the hail-and-farewell cliche because of their relative occurrence, i.e., last event of conference. However, it may be that the hail-and-farewell cliche does not include chairman's house in the set of places the party can take place. Should this cause the match to fail? Hayes-Roth, among others, discusses the problem of partial matches in frame-like systems [29].

(2) Related to the above, when should cliches be presented? If we have a loose matching policy, the system will bombard the user with potential matches.

(3) We find that users often will accept a "right sounding" cliche without checking all of the details. Thus, an expert analyst may say "what you have described so far sounds like a foo", and the user may accept the foo cliche without further misunderstanding checks, e.g., "everyone knows what a foo is".

Both the first and second problems are knotty. We view them together as one of our major research efforts. We note that one of the senior investigators on the grant, Natalie Dehn, has studied (and continues to study) aspects of the reminding problem from a cognitive science viewpoint [13]. We expect that our protocols will shed further light here.

For the third problem, we propose a primitive safeguard initially: we plan to mark all problem details accepted implicitly through acceptance of system proposed cliches. As we will discuss in the next section, these markings will be used to generate what-if scenarios to insure user and system concurrence.

7.3. A knowledge-based approach to symbolic evaluation

It is commonly recognized that some form of "operationalization" is necessary to find specification bugs (see [46] for strong support of this). The rapid prototyping approach provides this by building a source code program from the specification, and running it against intent. We choose to use a different approach to the problem by using KATE itself to "execute" portions of a problem specification. This is commonly known as symbolic evaluation (see [5], [42], [49] for some representative approaches). Cohen [12] and Swartout [45] have built a symbolic evaluator and behavior explainer for Gist that serves as a good reference point for our approach. In their system, the user designates a particular Gist action to test. The system then runs the action on symbolic data, and outputs an execution trace. Finally, the system sifts through the trace to summarize the highlights. While this is a useful approach to the problem of handling the often massive amounts of data produced by a run, Swartout notes the following [45]:

> "The current [Gist] symbolic evaluator is not goal driven. Rather than having a model of what might be interesting to look for in a specification, the evaluator

basically does forward-chaining reasoning until it reaches some heuristic cutoffs."

Swartout goes on to argue that if the Gist symbolic evaluator [12] had some notion of what was interesting, it could avoid lengthy and unproductive paths.

We agree wholeheartedly with Swartout's conjecture, but feel it must be elaborated along two axes. First, interestingness for a domain-independent language such as Gist must center on features of the language rather than features of the domain. What we propose is to explicitly represent what is interesting about the domain, again part of the knowledge we see an expert domain analyst having. Second, the goal of the Gist symbolic evaluator is to allow *the user* to test the specification; all the machinery is set up to explain the results of the test. We foresee a need for an active critic of the specification. Such a critic would generate its own test cases to try to poke holes in the current problem description. This distinction between the "attitude" of the two approaches is important. The work on symbolic evaluation to date is based on a view that the user knows what he or she wants; the problem is making sure the machine has represented it properly. In our view, a user has a sketchy idea of what her or she wants, and has rarely thought out all of the consequences. One of the roles of an expert domain analyst is to recognize and show to the user the ramifications of his or her actions, e.g., adding a new session to a conference, extending it another day, allowing non-registrants to attend, each of which is likely to have difficult to foresee interactions with the existing description. We believe all of this must happen in an interactive environment *tied to the development process*. Thus, symbolic evaluation is not something you invoke after the fact as you would a compiler, but instead should be part of the construction process itself, e.g., integrated with the editor. Given this, we propose the following roles for symbolic evaluation in the KATE system:

(1) The role of *validator*, as used by Swartout [45] and Cohen [12]. Here, *the user* selects test data and an action or actions to be confirmed; the system executes the action, and presents the results.

(2) The role of *validator*, as used in section 6.1. Here, *the system* selects test data and an action or actions to be confirmed. Selection is based on details -- slot values, prerequisites -- implicitly accepted by the user.

(3) The role of *refinement driver* that checks for missing details, e.g., "Let's suppose that x has occurred; how do you want to handle it?". This type of testing is initiated by the system to fill in holes in the current problem description (the DESIGNER system plays a similar role in algorithm design [42]).

(4) The role of *critic* that punches holes in the user's description, e.g., "You have specified a maximum limit of 100 on any one session. I can generate a scenario where 110 show up as follows ... What happens then?" This is an ever watchful system process that uses knowledge of the domain to generate boundary and error

conditions.

The distinction between the latter two is one of degrees. In both cases, the system expects the user to modify the problem description to more tightly specify details, special cases, or error handlers. Note that the above three uses of symbolic evaluation address both the *product* and *process* of problem description. That is, the user can do misunderstanding checks on the description itself. Further, the system can check the development steps taken to build the description, e.g., check the ramifications of adding an event or modifying an object. Finally, the system can drive the development by use of cleverly constructed examples that point to missing detail. Below are some hypothetical examples illustrating these points in the conference domain.

[Assume the user has added the concept of "local VIP" to the system, and it has been classified as a type of non-registered attendee.]

System: Assume the following:

a. One of the local VIPs shows up at the conference.

System: How will he or she be identified?

[When a new object is added, the system attempts to follow the interaction of the new concept with existing events and actions. Here the system is attempting to simulate how local VIPs move through the conference process. One action in the conference process is identify *(by the conference staff). Normal attendees can be identified by being on a registrant list. So far, local VIPs have no way of being identified.]*

User: Add a list of non-registered attendees, and place local VIPs on it.

System: Some conferences have a list of special-case people who are allowed to attend without registering. Such a list contains the person's name, affiliation, and sponsor. Would this work as the list of non-registered attendees?

[Instead of asking for details about the new "list of non-registered attendees", the system attempts to use a case it knows about.]

User: Yes, but forget the sponsor field.

System: Alright. Assume the following:

a. There exists a list of unregistered attendees with local VIPs on it.

b. A conference staffer wants to verify a particular local VIP on the list.

System: How can verification take place?

[Again, the system tests the implications of the user's modification. Part of the system's domain knowledge is that attendee lists must often be verified by conference staffers, and that the sponsor field is one of the inputs to the verification action. If the sponsor field is removed, the verification action is incomplete.]

User: Delete the verification action, I don't need it.

[This in turn will lead the system to look for ramifications of a missing verification action.]

...

In the above scenario, the system was attempting to add the details necessary to incorporate local VIPs into the description. In the next scenario, the system turns to the role of critic.

System: Assume the following:

a. the workshop is limited to 100 registrants (as you specified)

b. the session room will hold 110 people (as you specified)

c. 11 local VIPs are on the list of unregistered attendees

d. 111 registrants and local VIPs attempt to go to the same session

System: What happens?

[The system generates this scenario by using knowledge about a) the limited resources of a conference (e.g., room capacity), and b) that special case lists often neglect/bypass the maximum and minimum constraints on resources. In essence, the system is looking for overflow conditions caused by new objects (e.g., local VIPs) being added to the problem.]

User: first-come-first-serve [*A domain cliche.*]

[This is the easy way out for now. A more thoughtful approach would likely lead to modifications to the maximum registered attendees, the maximum number of unregistered attendees, or both. In any case, the system will dutifully show the user the implications of his or her modifications.]

...

Finally, the user may check that there is no misunderstanding on what to do in certain situations. Assume the user has specified that incomplete registration forms will be accepted and marked as such. He or she now wants to make sure that this will lead to the right behavior.

User: Assume the following:

a. the size of the registrant list is 99

b. you receive the following from potential attendee P100

1. completed registration form
2. completed questionnaire
3. unsigned check [*A domain cliche.*]

User: how is P100 handled by the registration action?

System: P100 is placed on the registrant list. Entry is marked as incomplete.

[At some point the user or system will get around to dealing with the changing of an incomplete to a complete. In particular, the system knows that many things accepted in incomplete form must have some action associated with them to make them complete.]

The system generation of examples such as those above is a major component of our research plan. We see the need for the following components to carry it off:

- A rule-based evaluator that will interpret OZ state-transition diagrams. This follows the same approach as that taken by the DESIGNER [42] system (see appendix D). As with their system, we expect to be able to execute incomplete designs (nee problem descriptions).

- A structured domain model that explicitly ties objects to and classes to superclasses (as does MOSS), and objects/classes to OZ processes and actions. Thus, a change to an object or class can be transmitted to all relevant actions. A change to an action can be transmitted to all relevant objects or classes. Balzer categorizes what type of ripple effects one can expect between objects and classes [3] (see also section 14.2). We expect to extend this into processes and actions.

- A rule-base that can generate 1) interesting *domain* conditions to verify, and 2) test data for producing scenarios from those conditions. We have built such a rule-base for a toy world; our work will be to extend the ideas into the conference domain.

In summary, we see the work intersecting with research in the areas of operational specifications, knowledge-based editing [47], and automated theory testing. We believe the latter work in particular holds great promise for our work. We discuss its relation to KATE in section 9.2. See also Professor Dietterich's description of research in appendix A.

8. What We Propose

We are embracing an interactive, knowledge-based approach to specification construction. Below we layout the dimensions of the knowledge we will need, and discuss which we will investigate initially. Notation: start+k refers to start-of-grant plus k months, where k ranges from 0 to 36. Section 4.7 discusses expected functionality at start+0.

Knowledge representation of the conference domain.

Includes primitive terms and concepts, special cases, error conditions, concrete examples. Must have the ability to reason about each in terms of the acquisition and explanation processes.

What we propose: Of course such a model can easily be an unbounded sink of effort. For instance, we might expect a good model to include goals and intentions, authority, rights, and attitudes (cooperative, competitive, indifferent), along with a myriad of other social issues. We expect we must represent aspects of all of these, *at some level.* Our experience from working with the resource management and transportation domains, both of which eventually forced us to deal with similar social interactions, tells us that prototype systems can be built by shallowly modeling some components while concentrating more deeply on others.

Research plan: in pursuit of bringing up a prototype system quickly, we plan to concentrate first on breadth and then on depth. We expect to be able to put on limited demonstrations by start+12. More elaborate demonstrations are planned for start+24 and start+36.

Automation of the specification process.

Models of refinement, task agendas, task suspension and resumption, cliche presentation, misunderstanding checks, critiques.

What we propose: This will be one of the major efforts of our research. There are four issues to contend with: 1) the representation of the refinement model in an interactive system, 2) the use of domain cliches to both acquire and explain problems, 3) the use of symbolic evaluation to understand intended behavior, and 4) controlling interactions among the preceding three. Each of the these are major research efforts, e.g., we expect a PhD thesis to come out of the third during the granting period (see appendix A). We plan to address the first three in detail, and finesse the fourth, as discussed in section 6.

Research plan: Our work on Glitter will focus first on representation of refinement knowledge in our domain (demo by start+18), and second on elaboration of control (demo at start+36). Our main concern with domain cliches is one of functionality: how should they be represented, matched, presented. We plan to build a representative set by start+12 to investigate these issues. We expect to conduct matching and presentation experiments throughout the grant. Symbolic evaluation will be undertaken in three stages: generation of boundary and error conditions (full demo by start+24, partial demos earlier); generation of refinement examples (functional by start+30); generation of validation examples (demo at start+36).

Knowledge representation of the software development process.

Includes known hard implementation problems, machine and language limitations.

What we propose: This will not be a major focus of our initial work. We expect this initially to be of the form of rules-of-thumb, similar to Kant's resource rules [31].

Research plan: 3 month task that can be undertaken any time after the first year (demo at start+36).

Interface issues.

Natural language, graphics, menus.

What we propose: We are in essence freezing this problem (see section 6). We expect our system to have a state-of-the-art interface based on the tools we have constructed to date (discussed in section 4.1 of this proposal). The use of natural language is currently of secondary interest to us.

Research plan: We expect to elaborate the interface as experience is gained, e.g., analysis of our protocol experiments is finished (start+6).

Learning from doing: incorporating new domain knowledge automatically.

Our goal here is to automatically incorporate new experiences of the system into the domain model, and then apply it when appropriate during subsequent specification construction efforts.

What we propose: As we discuss in section 5, there are moderate to very hard parts of this problem. We do expect the system to have an experience base from which to work, and to be able to pick up new examples as new conferences are described. We do not expect our final system to be able analyze why it did a good or bad job on assisting a specific user, and learn what to do in the future to enforce or correct its behavior.

Research plan: Initially, we plan to code examples by hand. These will be part of the demonstration system at start+12. We expect to be able to automatically incorporate new problem descriptions as examples by start+24.

Generation of formal specifications from KATE specifications.

We expect the output of our system to be specifications written in languages such as Gist [33], RML [27], Draco [35], and ORBS [19].

What we propose: We have argued that even with a system like KATE, building a correct specification often requires feedback from the implementation process. Up until now, we have been manually translating KATE problem descriptions into compilable code to get this feedback. While this is painful and sometimes inaccurate, we feel it is less important to automate this backend function than the specification acquisition functions discussed above. Our plan is to wait until the third year before addressing this problem in detail. As a benchmark, we will likely choose Gist as the target language. However, we see a collaborative effort of building a conference organization domain model for Draco as an approach worth pursuing as well. If successful, this could lead to a very small gap to bridge for our specification generator.

Research plan: We will begin to build a Gist backend at start+24. We expect to be able to generate Gist code by start+36.

We believe demonstration systems, even if they do not fully implement all the goals of a project, are useful (c.f. KBEmacs [47]). Hence, our goal is to have a working demonstration system early (start+12) with more elaborate demonstration systems at one year intervals.

9. Related Work

In previous sections, we have attempted to tie in related research projects within the context of particular aspects of our proposal. We will not discuss those projects further here, but instead point the interested reader to appendix D. In this section we will discuss two topics that we believe need further emphasis. The first places our work in perspective and makes arguments for its significance to the problem of software development in general. The second discusses work in the area of learning that we think has great potential for our research.

9.1. A proposal for a KBSA

We find interesting the close similarity between the functionality Green, et. al. [26] see in the *Requirements* component of a Knowledge-Based Software Assistant (KBSA) and our proposed system:

> "Requirements will be acquired by KBSA via dialog with end-users. These end-users will define and modify the requirements and behavior of their desired system by a combination of high-level, domain-specific requirements languages, examples, traces, state-transition diagrams, graphics, and so on, in whatever mix they find comfortable. The process will be a mixed initiative dialog, where the sequence of statements need not correspond to the organization of the final program. KBSA's role is to have enough knowledge about requirement analysis and specific domain applications to be able to accept and process these descriptions. The KBSA will organize the stated requirements and incorporate them into existing descriptions. It will notice inconsistencies and missing parts of the requirements, and suggest remedies, fil in pieces, and point out tradeoffs whenever it can. The KBSA will also, on request, describe the current state of the requirements specification in natural language, graphically, or by simulating behavior of the system as much as possible. The KBSA will help integrate new requirements into an existing requirements specification and will use knowledge-based program refinement techniques to help transform these requirements into executable specification languages.
>
> Knowledge-based tools for requirements analysis will have a high payoff... with most of the human effort in software development eventually going into this process"

9.2. Theory formation

Tom Dietterich (see appendix A) is working on a system that learns UNIX commands by building theories, testing them, revising them, repeating the cycle. He proposes three types of experiments that seem close to our notion of building up a problem description and then testing it:

(1) *Exploration experiments* involve trying a new situation that the theory makes no predictions about. This is similar to our use of example generation to fill in holes in the current description.

(2) *Confirmation experiments* test a situation that the theory makes predictions about, but predictions are built on plausible inference. This is similar to our notion of verifying that cliches, presented on partial matches and accepted by the user, actually fit the problem.

(3) *Discrimination experiments* attempt to lead credence to one out of a set of competing theories. Although we showed no examples of this type of problem, it is clear that more than one cliche may be a candidate given partial matches. We expect results of Dietterich's work here to be of major value to this selection problem. It would also seem to contribute to focus of attention type of control problems in the system.

All in all, we expect this work, along with Downing's (see appendix A), will have a major impact on our work of example generation discussed in section 7.3.

Also related to our work is DeJong et al.'s (1985) use of experimental design to refine partial theories of real-world phenomena. But while their system designs experiments after noticing discrepancies between the world model and the real world, ours will do so to resolve potential, implicit difficulties in the current conference specification (the system's partial theory). Predefined models of general conferences will aid recognition of these user-unforeseen problems. DeJong experiments to help the computer model conform to the natural world, while we will experiment to help the user refine his conference specifications in accordance with the computer's common sense and general conference knowledge. They experiment to explain problems. We will experiment to point them out.

10. References

(1) Adelson, B, Soloway, E., The role of domain experience in software design, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

(2) Arango, G,, Freeman, P., Modeling knowledge for software development, In *Proceedings of the 3rd International Workshop on Software Specification and Design*, London, 1985

(3) Balzer, R., Automated Enhancement of Knowledge Representations, In *Proceedings of 9th International Joint Conference on AI*, 1985

(4) Balzer, R., A 15 year perspective on automatic programming, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985 IEEE Transactions on Software Engineering, Vol. 11, No. 11, Nov. 1985

(5) Balzer, R., Goldman, N., Wile, D., Operational specifications as the basis for rapid prototyping, *SIGSOFT Softw. Eng. Notes*, 7(5) (1982)

(6) Barstow, D., Domain-specific automatic programming, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985 IEEE Transactions on Software Engineering, Vol. 11, No. 11, Nov. 1985

(7) Barstow, D., Barth, P., Dietz, R., Greenspan, S., Observations on Specifications and Automatic Programming, In *Proceedings of the 3rd International Workshop on Software Specification and Design*, London, 1985

(8) Bennet, J., ROGET: A Knowledge-Based System for Acquiring the Conceptual Structure of a Diagnostic Expert System, *Journal of Automated Reasoning*, **1(1), 1985**

(9) Blum, B., On how we get invalid systems, In *Proceedings of the 3rd International Workshop on Software Specification and Design*, London, 1985

(10) Bruno, G., Marchetto, G., A methodology based on high level petri nets for the specification and design of control systems, In *Proceedings of the 3rd International Workshop on Software Specification and Design*, London, 1985

(11) Cohen, D. *AP5 Manual*, Information Sciences Institute, Marina Del Rey, Ca., 1985

(12) Cohen, D. Symbolic execution of the Gist specification language, In *Proceedings of the 8th International Conference on AI*, 1983

(13) Dehn, N., A reconstructive model of long term memory, PhD Thesis, Computer Science Department, Yale University, New Haven, Ct.

(14) Doyle, J., Expert Systems and the "Myth" of Symbolic Reasoning, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

(15) Eastman, C., Explorations of the Cognitive Processes in Design, CMU, Feb. 26, 1968 Tech. Report NTIS AD 671 158.

(16) Erman, L., London, P., Fickas, S., The design and example use of Hearsay III, In *7th International Joint Conference on AI*, Vancouver, 1981

(17) Feather, M., Language support for the specification and development of composite systems, Information Sciences Institute, Marina Del Rey, Ca., 1985

(18) Fickas, S., Mechanizing software specification, In *Workshop on Models and Languages for Software Specification*, Orlando, 1984

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

(20) Fickas, S., Automating the Transformational Development of Software, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

(21) Fickas, S., Laursen, D., Laursen, J., Knowledge-based software specification, In *Workshop on Knowledge Based Design*, Rutgers Univ., 1984

(22) Fickas, S., Novick, D., Reesor, R., An environment for building rule based systems, In *3rd Annual Conference on Intelligent Systems and Machines*, 1985

(23) Fickas, S., Novick, D., Control in rule based systems: relaxing restrictive assumptions, In *5th International Conference on Expert Systems and Their Applications*, 1985

(24) Fickas, S., Downing, K., Novick, D., Robinson, W., The specification, design, and implementation of large knowledge-based systems, Invited paper, *IEEE Annual Northwest Conference on Computer Science*, Portland, 1985

(25) Goldman, N., Three Dimensions of Design, In *Proceedings of the Third Annual National Conference on Artificial Intelligence*, AAAI, Washington, D.C., 1983

(26) 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

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

(28) Greenspan, S., Mylopoulos, J., A Knowledge Representation Approach to Software Engineering: The Taxis Project, In *Proceedings of the Conference of the Canadian Information Processing Society*, Ottawa, 1983

(29) Hayes-Roth, F., The role of partial and best matches in knowledge systems, *Pattern-Directed Inference Systems*, Academic Press, 1978

(30) Kahn, G., Nowlan, S., McDermott, J., MORE: An Intelligent Knowledge Acquisition Tool, In *Proceedings of 9th International Joint Conference on AI*, 1985

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

(32) Lehman, M., A Further Model of Coherent Programming Processes, In *Proceedings of the Software Process Workshop*, 1984

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

(34) Mostow, J. Towards better models of the design process, *AI Magazine*, Spring 1985

(35) Neighbors, J., The DRACO approach to constructing software form reusable components, In *IEEE Transactions on Software Engineering*, Vol. 10, No. 9, Sept. 1984

(36) Ramanathan, J., Shubra, C., Modeling of Problem Domains for Driving Program Development Systems, Computer and Information Science Dept, Ohio State University, 1981

(37) Schank, R., Abelson, H., Scripts, Plans, and Goals, Lawrence Erlbaum Associates

(38) Simoudis, E., Fickas, S. The Application of Knowledge-Based Design Techniques to Circuit Design, In *IEEE International Conference on Computer-Aided Design*, 1985

(39) Smith, R., Structured Object Programming in STROBE, Shlumberger-Doll Research Lab, Ridgefield, Ct., 1984

(40) Smith, D., Kotik, G., Westfold, S., Research on knowledge-based software environments at Kestrel Institute, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

(41) Smith, R., Mitchell, T., Winston, H., Buchanan, B., Representation and Use of Explicit Justifications for Knowledge Base Refinement, In *Proceedings of 9th International Joint Conference on AI*, 1985

(42) Steier, D., Kant, E., The Roles of Execution and Analysis in Algorithm Design, In *IEEE Transactions on Software Engineering,* Vol. 11, No. 11, Nov. 1985

(43) Sunshine, C., Thompson, R., Erickson, S., Gerhart, S., Schwabe, D., Specification and Verification of Communication Protocols in AFFIRM
Using State Transition Models, *Software Specification Techniques*, Eds. Gehani & McGettrick, Addison-Wesley, 1985

(44) Swartout, W. Gist English generator, In *Proceedings of the National Conference on AI*, 1982

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

(46) Swartout, W., Balzer, R., On the inevitable intertwining of specification and implementation, *Commun. of ACM* **25**(7) (1982)

(47) Waters, R., The Programmer's Apprentice: A session with KBEmacs, In *IEEE Transactions on Software Engineering,* Vol. 11, No. 11, Nov. 1985

(48) Wile, D., Program development: formal explanations of implementations, In *Commun. of ACM*, 1983

(49) Zave, P., The Operational versus the Conventional Approach to Software Development, In *Commun. of ACM*, Vol. 27, No. 2, February 1984

12. Appendix B - The Conference Organization Domain

There are three principal reasons why we have chosen the conference organization domain. First, it ties in with our work on analyzing problems of resource management and transportation, two domains we have analyzed as part of our effort to catalog specifications. In particular, we have built up representations of academic departments that includes people, rooms, equipment, and constraints on assigning resources. We have already begun to map this work onto the conference domain. Our work on abstracting transportation systems ties in with our need to worry about the transportation of people among sessions, special events, airport, etc.

The second reason for choosing this domain is that it is one of a woefully small set of common problems studied in specification research. Up to now, it has been used to test the representational power of proposed specification languages. We are proposing to use it to test the ability of an automated system to help build a specification in these languages. Because others have worried about specification issues in this domain (c.f. [27]), we hope to gain from their experience. It also allows us to preview what type of target we must shoot for as output of the KATE system.

The third reason is that the conference domain is of the right complexity. It moves up from the primarily mathematical domains studied traditionally. It is somewhere between programming-in-the-large and programming-in-the-small -- right where we should be given the state-of-the-art of knowledge-based specification tools.

As a final note, we have an expert conference organizer in our department in the form of our chairperson, whose recent efforts include the Foundation of Computer Science (FOCS) conference in Portland, and the summer workshop on Graph Theory held in Eugene. He will be one of our protocol subjects.

12.1. Useful software

Below we list a sampling of software components that we might expect to be produced for the conference domain[13].

- *Maintain attendee lists for main conference events.* Will include registration lists and meal lists.

- *Maintain resource lists for special conference events.* Will include special attendees list, special event lists.

[13]We reiterate that our research is concerned with the production of the *formal specification* of this software, not its design and implementation.

- *Check-in registrants.*
- *Check-in special-case attendees.*
- *Maintain session schedule.*
- *Maintain meal schedule.*
- *Answer questions about local restaurants and entertainment.*
- *Provide message handling.* May be simple (give room number, print message on board), or more involved (she is giving a talk at 1100, and is attending the banquet Tuesday night, and is in room 23). Store message for record.
- *Track submitted papers.*
- *Referee management.* List of referees, what they're specialities are, what papers they have been assigned and when, number of dun notices sent.
- *Conference committee management.* List of committee members, what conference events they are responsible for.
- *Maintain checklist of organizational milestones.* Conference announcement/call-for-papers, submission deadline, acceptance deadline, accommodations, meals, etc.

14. Appendix D - Extended Discussion of Related Work

This appendix supplements section 9. It discusses in more detail the relation of work mentioned in brief in the main portions of this proposal.

14.1. The SAFE project

The SAFE project was an attempt to translate natural language descriptions of a problem into a formal specification. A system was built that could take a parenthesized English description of a message routing system, and convert it into a more formal process representation. As Balzer notes in [4], the work on SAFE spawned the work on Gist. Balzer further notes that the Gist spawn was inevitable: the SAFE project needed a formal specification language to target its output. Our work on KATE shares the goal of SAFE in accepting an informal problem description and building a formal specification[14]. However, our approaches are quite different. SAFE was a batch compiler of "complete" specifications. In essence, its concern was less with the specification construction process and more with the specification translation process. Further, SAFE was meant to be a domain-independent tool. Its knowledge was in the general area of process control. It's main power was *automatically* disambiguating its English input. It did this by using a powerful form of symbolic evaluation. With KATE, our focus is on interactive problem solving. This brings with it a central concern for the specification process (c.f., [25]). We also expect to rely less on full blown symbolic reasoning, and more on intelligent generation of verifiable examples, as discussed in detail in section 7,3. In summary, we would hope to be able to use results from SAFE in automating portions of KATE'S inference machinery. However, we believe that automatic (and behind the scenes) reasoning must be understandable by and verifiable with the user.

14.2. Specification Refinement

More recent work out of ISI plays a much more central role in our research. In particular, Goldman's and Feather's work on specification refinement [25, 17], forms a key component of KATE. Goldman lays out three axes along which refinements take place: structural, temporal, and amount of coverage. These are the main drivers of our specification acquisition process.

Balzer has defined an editor [3] for maintaining knowledge bases that provides commands for adding, deleteing, and modifying frame-like objects. The system relies on an explicit model of class and attribute structure to analyze what the ramifications of a

[14] In some sense, our work completes the SAFE circle. After struggling for the last six years in building Gist specifications, we see the need for a specification assistant.

change will be. In essence, Balzer's work attempts to lay the underpinnings for modifying a knowledge base. We expect that results coming from this work can be used directly in our Glitter-based refinement tool, and as a secondary aid to our symbolic evaluator.

14.3. Gist Paraphraser

Another ISI project that is of interest to us is Swartout's Gist paraphraser [44]. This system has an understanding of Gist syntax and semantics, and is able to produce a reasonable paraphrase of a Gist specification if the right mnemonic terms are used; the paraphraser has no understanding of the actual problem under specification. Our attempt with KATE is to tie the paraphrasing capability into the domain model. This will allow us to paraphrase a problem in domain terms, and use concrete examples pulled from the domain.

14.4. The PHI-NIX project

The PHI-NIX project at Schlumberger-Doll is an attempt to apply domain dependent knowledge to the software specification *and implementation* problem [6]. As with SAFE, PHI-NIX takes a batch compilation approach. High level problem descriptions are taken in, and the system first translates these into a formal specification, and then maps this specification into compilable code. During the specification process, the system's major concern is with mapping mathematical, and some sense idealized descriptions onto physical equipment and approximation techniques. The domain knowledge comes from a set of rules that describe various definitions, facts, and properties in the oil drilling world. As with SAFE, the focus is more on translation than construction. Hence, there is no notion of interacting with the user to further refine his or her problem: it appears that oil drilling problems can be succinctly and completely stated. The tie in to KATE is then in the use of domain knowledge to fill in details. PHI-NIX relies on rule-based heuristics to guide it to a reasonable formal representation (and later an implementation) of the problem. In a somewhat similar vein, we have experimented with building ORBS rules for generating interesting boundary conditions for things like registrant lists (e.g., overflow), and session placement (e.g., walking distance, driving distance). In summary, while our specific goals are different, we find it encouraging that the Schlumberger group has been able to successfully apply domain knowledge to their application.

14.5. The DRACO project

Neighbor's Draco system [35][15] shares our goal of capturing domain knowledge so it can

[15]Peter Freeman's group at UC Irvine has continued the Draco work [2].

be used in specification, but again deals more with translation than with the construction process. To capture the work of a domain analyst, and hence reuse that work later, Draco defines tools for building domain dependent specification languages. Using this model, the analyst identifies the objects and operations of the domain, and defines a language for representing them. Draco supplies a BNF notation, parsers, pretty printers, and a transformation facility for optimizing a program. The semantics of the newly defined language are defined by mapping language constructs onto other existing domain languages within Draco. We view KATE as the front end to a system like Draco. With KATE, we are worrying about the process of building a formal specification. Draco is worried about defining new formal specification languages tailored to specific domains. In many ways, this match looks a good one. If we can tailor a conference organization language through Draco, the bridge from KATE to Draco could be a short one.

14.6. Studies of the design process

Adelson and Soloway have studied the role of domain experience in software design [1]. They produced protocols of expert designers working with familiar problems in familiar domains, unfamiliar problems in familiar domains, and unfamiliar problems in unfamiliar domains. Although the Yale group's experiments differ from our protocol experiments in significant ways (e.g., they did not involve an interaction between designer and user), we believe their results lend some secondary support to our approach. In particular, we found the following of interest. Their designers' paper-sketching of the problem as a process hierarchy looks translatable into our state-transition diagrams. Their designers' behavior of gradual refinement is supported, at least partially, by our model of refinement. Another behavior they saw as common was that of construction of "slippery road" signs[16] that warned of later trouble at certain points. In KATE, these would be represented as part of the analyst's knowledge, i.e., they are part of the domain model. Finally, their results showed that mental simulation occurs most often on unfamiliar parts of the problem. We argue that this at least partially supports our notion of system-driven symbolic evaluation of new or modified portions of a problem.

The DESIGNER project at CMU [42] is studying the use of symbolic evaluation in designing geometric algorithms. Their work is similar to ours in that 1) both allow evaluation of partial problem/algorithm descriptions, 2) both are based on a graphical process language, theirs dataflow, ours state-transition, 3) execution centers on a rule-based interpreter, OPS5 in theirs, ORBS in ours, and 4) both attempt to use "errors" to further guide the process, design in theirs, acquisition in ours. The main differences are in the levels and overall goals. The DESIGNER system attempts to automatically generate a compilable algorithm given a complete specification, i.e., it is another approach to Automatic Programming. We are interested in the acquisition and construction of the

[16]Our term, not theirs.

specification in the first place, e.g., is the specification handed to DESIGNER complete, will it handle the known sticky cases in geometry problems?

Mostow's recent overview paper on knowledge-based design [34] has provided many useful insights into our work on specification construction (nee design).

14.7. The ROGET project

The ROGET system [8] perhaps comes closest to our goals in this proposal. ROGET, by system-driven questioning, allows a user to build what is referred to as the *conceptual structure* of an expert system for solving diagnostic problems. ROGET firsts asks the user to classify his or her diagnostic problem, and then presents the user with a textual description of the main tasks the application will need to invoke. After this, the user is asked, task by task, if any modifications are necessary. Next, the system warns the user of known hard implementation problems given the final task structure. Finally, ROGET generates an EMYCIN skeleton that handles the tasking specified. Even though ROGET addresses only a part of the problem of building a diagnostic expert-system, it presents interesting ideas. ROGET's knowledge of what is hard to implement in EMYCIN corresponds to KATE'S implementation knowledge. It's built in knowledge of the primitives of building tasking structures is the type of domain knowledge we are advocating. The major functional differences -- no symbolic evaluation, no notion of gradual refinement, no critique of tasking from a *domain viewpoint* -- point out differences in goals: ROGET, as it stands now, remains a tool for *translating* portions of an expert's diagnostic problem into machine terms, hence bringing it closer to an EMYCIN implementation; KATE attempts to *acquire* a problem by presenting examples, allowing problem abstraction and refinement, pointing out special cases and error conditions, actively analyzing the user's construction of the description, in essence making sure that the right problem is specified and solved.