Automating the Specification Process

Stephen Fickas

CIS-TR-87-05
December 14, 1987

Department of Computer and Information Science
University of Oregon

Automating the Specification Process

Stephen Fickas

Computer Science Department
University of Oregon
Eugene, OR. 97403

Abstract

Recent research results in formalizing and automating software specification move us closer to a computer-based specification assistant. In this paper, we review three projects that we believe are particularly relevant to this goal. For each project we describe first the underlying model, and second our efforts to study it by construction of a prototype tool based on the model. Finally, we discuss incorporating the results of our study into a knowledge-based system that assists in the construction of a formal specification.

1. Specification as a process

A specification phase is a component of almost all modern software development models[1]. However, the specification process itself has only recently come under scrutiny. In particular, relatively little attention has been given to the way a specification evolves from an often vague set of client descriptions to a formal specification of a problem. Practical techniques that do exist supply notation and guidelines that can be interpreted by an *expert* software analyst. We highlight the word expert to note that the main body of knowledge of specification construction remains with the human in these types of methodologies. A portion of this knowledge includes how to interact with a client to extract the initial requirements, how to control further acquisition and design[2], how to recognize bad designs, how to explain a specification to a client and thus validate it, and how to talk a client out of a problematic requirement, where problematic here could mean a) impossible to satisfy, b) conflicts with a more important requirement, or c) is something that is really not needed. We are interested in bringing more of this type of expert analysis knowledge into the machine. While informal guidelines are useful for many things, we believe that to successfully tackle such traditional problems of specification as ambiguity and incompleteness, we will have to have the machine

[1] Our use of the word *specification* is discussed in section 1.1.

[2] Since *design* typically follows specification in most software development models, our use of it here may be confusing. Unless otherwise qualified, we will use the term design in this paper to refer to the process of building a formal specification.

CIS Technical Report 87-05, November, 1987.

become involved. This in turn must rest on results in two areas, a formal model of the specification process itself, and a computer-based representation that will allow it to be automated. In this paper we argue that a critical mass has been reached in both areas.

Our approach will be to show that a set of connected research results have lead to the following characterization of software specification:

- *The process is evolutionary.* It is neither feasible nor desirable to take a massive transcription of a client's needs. A specification starts with a simple, often naive description of a problem, and gradually evolves, through refinement, case analysis, and even implementation [43], into a useful document.
- *The process is one of design.* In particular, the specification *product* is just as much a designed artifact as the program implementation. While different design techniques may be needed for each product, there is no technical reason to reserve the term design for the specification-to-code process. On the contrary, by viewing both specification and implementation as a design process, we may be prodded to look more closely at common design techniques (see discussion of Glitter in section 2).
- *The process often involves conflicting goals.* Almost any complex specification is the result of compromises and trade-offs between competing concerns, i.e., there is no perfect specification. Conflicts may arise between different user groups, users and administrators, performance versus budget, and a host of others.
- *The process involves collaboration between client and analyst.* Stated more strongly, clients have only a vague notion of what they want, and a narrow view of what is possible. A good analyst has expertise in pulling out the key points of a client's problem, presenting convincing arguments for and against the inclusion of certain components, and finally filling in the necessary details. All of this occurs within a rich, interactive, problem solving context that includes processes for problem acquisition, disambiguation, and critiquing.
- *The process is inherently knowledge-based.* This point is actually a summary of those above it. To design a specification through refinement, to identify the special cases, to find compromises for conflicting goals, and to argue effectively with a client about what has been wrought, requires knowledge of the application domain, knowledge of implementation possibilities, and knowledge of the production environment, *among others.*

Attempts have been made, using one or more of these characterizations as a focus, to define techniques that might automate some portion of the process. In particular, each of the following approaches has been argued as a possible basis for a computer-based tool to assist in building software specifications:

- Gradual elaboration.
- Design as problem solving.
- Parallel development.
- Knowledge-based critiquing.

In sections 2 through 4 of the paper, we will discuss each of these approaches in more detail and in the broader context in which they were initially defined. Along with each approach, we will discuss our attempts to view it as a model of design by constructing a specification tool based on it. The presentation style of each section will be to first introduce the particular specification model, next describe the tool we constructed for study, and finally discuss the results of our study.

In the final portion of the paper, we will describe a project that attempts to bring the results of our study together in a coherent form, and prescribes further research in the area.

1.1 A note on terminology

There is a confusing array of terms used to describe what goes on before one begins to make "how-to" decisions in developing a software system: *requirements analysis, specification, requirements specification, domain analysis, systems analysis, needs analysis*. We will not attempt to offer any notational unification here. However, we will describe our specific use of terms in this paper. In particular, our use of the terms 1) *analysis*, and 2) *specification* as a process are synonymous. Further, we will make no distinction in this paper between a *requirements model* and a *formal specification* (in section 7 we do attempt to offer some distinction between the two general concepts of requirements and specification). Finally, we will *not* rule out the following type of information appearing in a specification: a) limits on the development process itself, e.g., how much time and money is available for implementing the system, b) the human, machine, and monetary resources available to operate the system once delivered, and c) the organizational goals and policies in effect as design proceeds.

We will use the terms *design* and *development* synonymously, unless otherwise qualified, to refer to the construction of a formal software specification.

2. Specification design along dimensions

It is obvious that the construction of a complex artifact like a formal specification will follow an evolutionary process. Instead of springing full-borne from client to paper, the level of detail will come to light in a gradual and incremental fashion. While this is a fine sounding proclamation, we are left with the details of carrying it out. Goldman [20] examines at least some of these details by proposing that the evolutionary steps taken in building a specification can be characterized in one of three ways:

1. Steps that increase the *structural* granularity of the specification. For instance, we may break a monolithic object into sub-parts, revealing more structure in a state.
2. Steps that increase the *temporal* granularity of the specification. For instance, we may break a single action into a set of sub-actions, thus increasing the number of states modeled.
3. Steps that add new *cases* to consider. For instance, we may expand or restrict the set of possible behaviors permitted by a specification.

Goldman argues that starting with an idealized specification, these three steps can be composed into a sequence of specification changes that produce a final specification meeting a client's needs. He also speculates that these steps can by formalized and automated, leading to a tool that can both aid in building, and later understanding, a formal specification.

To demonstrate his ideas, Goldman uses a simple example of a game of baseball as a specification problem. As an initial start, Goldman provides the following specification, paraphrased here from its Gist[3] representation in [20]:

There are two teams, HOME and VISITOR. Each team has an associated SCORE which starts at 0. A GAME consists of 9 PLAY-INNING actions. A PLAY-INNING consists of a BAT by HOME and a BAT by VISITOR, in no particular order and as an atomic action (the two BATs produce only a single state change). A BAT by team T consists of incrementing T's score by a non-negative integer.

Figure 1. Initial baseball specification

Goldman argues for the above description as a good starting point along the following minimalist lines: 1) the minimal structural description should include the major components of a game, i.e., participants and scoring, 2) the minimal temporal granularity should be a snapshot after every inning (a true minimalist might argue that innings are details; take a snapshot before and after a game, producing only a single state change), and 3) start with normal cases initially.

[3]Gist is described in more detail in [27]. Briefly, it is a domain-independent specification language particularly suited to the specification of ongoing interacting processes. It models behaviors as sequences of global world states, where each state is represented in terms of relations among typed objects. Gist's constructs allow objects, relations, and behaviors to be expressed in a concise fashion.

In his paper, Goldman informally describes how the three types of development steps can be used to fill out the initial specification into a final version that can be described as follows:

There are two teams, HOME and VISITOR. Each team has an associated SCORE which starts at 0. A GAME consists of PLAY-INNING actions until the INNING is greater than or equal to 9 and the score is not tied. A PLAY-INNING consists of a BAT by VISITOR and a BAT by HOME, in that order. A BAT by team T consists of an arbitrary number of PLAY actions. A PLAY consists of incrementing T's score by a non-negative integer between 0 and 4. A GAME has an associated INNING number that counts the number of PLAY-INNING actions.

Special case 1: if the INNING is 9 and the BAT by VISITOR is complete and the SCORE of HOME is greater than the SCORE of VISITOR then the game is over.

Special case 2: if the INNING is 9 or greater and a PLAY is completed by the HOME team and the SCORE of HOME is greater than the SCORE of VISITOR then the game is over.

Figure 2. Final Baseball Specification

In closing, Goldman notes

> "Although we have not 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."

The next section describes an effort to both formalize and automate the informal and manual development description given by Goldman in [20]. From this study, we will look for answers to the following questions:

1. Is it feasible to formalize and automate Goldman's model of specification, and hence support his above conjecture?
2. If so, how much domain *specific* knowledge will be needed?
3. In what form can it be represented?
4. Is Goldman's model a sufficient one for realistic specification problems?

2.1 Model implementation

To study Goldman's specification design model, we decided to formally represent his three elaboration strategies -- temporal elaboration, structural elaboration, and case analysis -- in a problem solving system. The system chosen was Glitter [14]. The original Glitter system was constructed to automate the selection and application of transformations to move from a formal specification to a valid implementation, validity being assured by the use of *correctness-preserving* transformations. Glitter was based on an interactive problem solving approach where the user would supply implementation goals, and Glitter would find methods to achieve those goals and choose the right transformations to apply. Because more than one method would often be available for solving a goal, Glitter contained selection rules for choosing among competing methods. To help manage the development process, Glitter maintained a tree of implementation paths, where each node represented a problem solving state, and each arc one of possible many alternative methods chosen to move to the next state/node. Attached to each arc was a record of the process that lead to that arc/method being selected (e.g., the selection rules that chose it).

To apply Glitter to the new problem of specification "implementation", a Glitter shell was produced by stripping away the transformational implementation knowledge while leaving intact the general method and selection rule catalog structures, the problem solving control, and the implementation tree. To use the shell as an implementation of Goldman's model, we filled in the following items:

- *A goal language.* The highest level goals became Elaborate_Structure, Elaborate_Temporal_State, Add_Case, corresponding to Goldman's three dimensions of design. It was also necessary to fill in the intermediate goals needed to do incremental problem solving. In some sense, the construction of these goals and the methods below lead to the major result of this study: domain *specific* components would be required to carry out the development in an automated fashion.
- *A set of methods.* To define methods for achieving our goals, we were forced to formalize the informal descriptions of methods given by Goldman. To do this, we borrowed what domain independent methods we could from Goldman's paper and the original Glitter system, and additionally defined a set of domain specific methods.
- *A set of selection rules.* Deciding among competing specification strategies must rest on some decision criteria. In our original Glitter system, we relied on notions of efficiency and feasibility to choose among competing implementation methods. In our new system, we must decide not on *how* to solve a problem, but on *what* problem to solve. We found that general rules were hard to come by. We were able to identify a few domain specific rules, but feel we have just scratched the surface of the problem. In summary, Goldman did not address the problem of strategy selection in his paper, and we were able to make only a modest dent.

To avoid confusion, we will refer to the new Glitter system, which includes the goals, methods, and selection rules described above, as Glitter-S.

2.2 Development example

To provide a better understanding of the Glitter-S system, and to better evaluate the feasibility of formalizing and automating Goldman's model, we will present a portion of the problem solving transcript taken from the Glitter-S development of a baseball specification similar to that used by Goldman (see Figures 1 and 2). First we will provide a rather detailed look at one step of the development, and then talk in more general terms about the remaining steps.

2.2.1 Detailed description

In this section, we will follow the detailed problem solving process employed by Glitter-S to handle the initial development step suggested by Goldman in [20]. This step focuses on the inadequacy of the initial specification to account for tie games. In particular, tie games are not allowed in baseball[4]. The initial specification (Figure 1) simply repeats the actions BAT for HOME and BAT for VISITOR 9 times, allowing final scores to be tied. We need to introduce a new case to allow tie games to go into extra innings. In Glitter-S we can do this by posting a goal that will eventually lead to a change to the specification that handles extra innings games.

(1) Goal: Add_Case "no ties"

The special case "no ties" is a primitive of the Glitter-S system[5]. In other words, we have incorporated concepts specific to competitive events into the system. This raises the question of the feasibility of using domain *independent* concepts to develop the baseball specification. If this is possible, it would appear that these concepts would be much more general, and hence useful in many more contexts. In fact our first attempts at defining the components of Glitter-S (i.e., goals, methods, selection rules) were centered on the original Glitter model of domain independent knowledge. However, it became clear that a reliance on domain independent knowledge for the specification development process was forcing us to describe our components 1) at a lower level than was useful, and 2) in an overly verbose and complex fashion. In particular, our highest level domain-*independent* goals were similar to the rather low level editing steps of the current system. In retrospect, it was here that we made a commitment to the use of domain specific knowledge in specification design. While we still believe that domain independent methods will play an important role in specification, as will be seen shortly in the example we are following, we no longer rule out the use of domain specific goals and methods, and this decision is reflected in all of our subsequent efforts.

[4]There are obscure cases in baseball where tie games may be allowed to stand. Neither Goldman nor we included these cases.

[5]**Notation:** domain specific items (i.e., concepts specific to games in general and baseball in particular) are placed in quotations.

Getting back to the goal posted, there exists a Glitter-S method for achieving the "no ties" case by attempting to weaken the *games* halting condition, and hence extend the game until a non-tie score is achieved[6]. Other approaches are possible, e.g., do not allow an inning to end with a tie score by weakening the *innings* halting condition. The extra-innings method was selected by the following Glitter-S rule, paraphrased here for readability (see [14] for a more detailed description of the goal, method, and rule language of Glitter):

If the domain is game-playing, **and**
scores are accumulated by opponents in discrete increments, **and**
the goal is to include a "no ties" condition,
then choose a method that extends the game until a winner is found.

This rule attempts to capture the notion that it is typical to avoid ties in a game with discrete scoring by letting the game continue until a non-tie score is reached. Baseball and tennis are such games. As always, there are exceptions to the rule, for instance boxing and cricket, and compromises like football and hockey. We did not attempt to further refine our selection rules to account for such exceptions (e.g., by representing knowledge of human endurance in various physical activities), being content to rely on gross typicality rules.

The selection of the method causes the following goal to be posted:

(1.1) Goal: Weaken_Halting_Condition "no ties"

The method we will employ to achieve this goal attempts to extend the game by replacing the loop repeat-9-times with a do-until loop. Note that this goal and method are part of the domain independent problem solving knowledge that Glitter-S embodies. In particular, we find that a continuum from very specific domain methods to very general Gist transformations is needed to be effective.

The do-until loop will need a halting condition that defines "no-ties". In other words, before we can do the repeat-to-do-until transformation, we must first define the until-condition. Borrowing terminology from the original Glitter system, we call this type of sub-goaling behavior a *jittering* step, i.e., one that gets things in place for a major step. The jittering goal posted is as follows:

(1.1.1) Goal: Define_New_Condition "no ties"

At this point we must rely on the rules of baseball to define non-tieness. We could include in Glitter-S a baseball-specific method, or possibly one slightly more general for inning-based games, that would match on this goal and proceed with the development process automatically. We chose not to define such a method for this example. Instead, the user will be called on to help unblock the problem solving process by supplying the missing method. This type of interactive problem solving development is very much in the Glitter tradition, and extends naturally to Glitter-S.

[6]This method actually incorporates two separate steps in Goldman's informal development: 1) ruling out tie games, and 2) extending into extra innings.

(3.) Goal: Add_Case "inevitable win"

. . .

There is another case to consider regarding inevitable wins: *during* the last half of any inning after the eighth, if the home team's score becomes greater than the visitor's, the home team should immediately be declared the winner and the game declared complete (of course it is possible to imagine games where it is legal to "pour it on" even after a win is assured).

The problem solving necessary to add this case requires that we eliminate unwanted states *within* the BAT action, i.e., states in which home players are coming up to bat after their team has an inevitable win. This requires that the BAT action be further refined (using the Elaborate_Temporal_State goal). In essence, to use one of Goldman's three development actions we will be required to use at least one of the remaining two as a type of jittering goal. For this problem, case analysis is driving temporal refinement.

A general method is chosen that breaks an action into sub-actions towards the goal of further restricting behavior. The user is called on to supply the sub-actions of a team's at-bat, namely players' at-bats. The BAT action now acts as a generator of player at-bats.

We must also worry about the amount a single player's at-bat can add to the score. The range is zero to four. Using interactive problem solving, we use Glitter-S to refine the original score-incrementer associated with a team's at-bat to an incrementer associated with a player's at-bat, and we further restrict any single increment to be in the range 0 - 4.

Finally we note that to remain consistent with Goldman's development, we treated the two aspects of inevitable wins separately by posting the same high level goal twice (goals 2 and 3), and selecting different methods to achieve each. A more natural strategy would be to post a single "inevitable win" goal, and define a method that handled the various aspects – forego team's at-bat, forego players' at-bat – as sub-problems.

This ends the specification development process as carried out by Goldman. One may ask whether we can view the specification that we (and Goldman) end up with as a "complete" specification? It clearly depends on the use the specification is going to be put to. If we are only interested in generating *legal* baseball scores, then it has only a few minor omissions having to do with suspended games. If we want something that will generate *realistic* baseball scores, then we have a major task ahead of us in adding the statistical nature of baseball play. To do this, we will have to better specify the relationship between the runs a single player can produce and what's gone on before his at-bat, eventually leading to a specification of men-on-base, hits, balks, etc. This question of completeness, and its relative nature, is one we take up in more detail when discussing the Kate system in section 5.

2.3 Discussion

Glitter-S is, in fact, an existence proof of the feasibility of building a specification development tool that is 1) based on a specific model of design, and 2) implemented as an interactive, problem solving system that at least partially automates the development process. At the same time, it became clear to us in constructing our tool that much lies behind Goldman's "three dimensions" as presented in his paper. In particular, while the notions of structural development, temporal development, and case analysis appear useful guiding principles, the nitty gritty process of mapping them to actual specification editing steps is far from straightforward. A question that soon became apparent was the amount of domain knowledge specific to baseball and games that would be needed to usefully document and automate the baseball development. Unlike the more constrained world of Glitter, where a relatively limited number of techniques were applicable across domains (see [27] for a description of some of these techniques), refining and modifying what amounts to problem descriptions has no such limitations: there is an infinite variety of problems we might want to describe, and no absolute notion of what is a good or bad problem description[7]. It appears the best one can do is capture the inherent constraints on any particular domain, and use these as a guide to specifying problems in that domain.

Given that domain knowledge appears to be a required component of a specification design tool, what form should this knowledge take? In Glitter-S, we used problem solving components (goals, methods, selection rules) to embody game-design knowledge. Thus, we attempted to guarantee that the client would be happy with a specification by insuring that its generation followed game-playing principles. To carry out the generative approach successfully requires that we anticipate all possible goals of a client, and build methods that will achieve them, and selection rules that will choose the best, given the constraints of the domain. This seems unreasonable for any complex domain. A client may always introduce some twist on a problem that we have not anticipated. An alternative approach is to allow relatively unconstrained generation by the client, but provide critical analysis of the slowly evolving specification along the way. This approach has its benefits: it gives the client a freehand, and hence allows the inclusion of any concept expressible in the specification language; it uses whatever knowledge the system possesses when appropriate, giving critiques on subjects it knows about, and leaving other parts of the specification as is. The drawback is one inherent in any analysis approach: we must recognize the concepts we wish to critique in the artifact we are examining. In Glitter-S, the concepts were compiled into the goals and methods; once concepts such as GAME and PLAY-INNING and BAT were identified in the initial specification, they were implicitly tracked through problem solving refinement. Without this track, we may be left with recognizing domain concepts in specification language terms, a knotty problem indeed. The punchline is that we have moved towards a compromise between the generative approach of Glitter-S and the analytic approach discussed above. This is discussed in more detail in section 5.

Finally, we note that our rhetorical question -- is Goldman's design model sufficient for building real specifications -- is slightly unfair. Goldman makes no claims that he is describing a tool or environment. However, it is clear to us that such an environment will be needed to support the construction of formal specifications in the real world. The next two models we look at will add further components to those needed by such an environment.

[7]There are syntactic-level specification criteria we might employ, e.g., non-redundant constructs, non-superfluous constructs. In [29], such goodness criteria are grouped under "the seven deadly sins" of specification. Conversely, one might argue that anything that makes a specification easier to understand, e.g., redundant views of the same thing, should not be considered an error.

3. Specification design as parallel development

Feather proposes a model of specification construction that is based on an approach he terms *parallel development* [13]. As with Goldman's model, Feather starts with a simple description of a problem. From there, various elaborations are considered in parallel. Thus one elaboration might be to refine action A, and another elaboration to add a special case S. Both of these may be carried out *independently*, producing two separate elaboration lines, one with "simple + refined A", and the other with "simple + case S". At some point , the two separate lines must be merged, and inconsistencies and interactions dealt with. Note that this is not simply a renaming of top-down design; each elaboration leads to a different, more detailed specification. In particular, there is no notion of fixing interfaces; each elaboration line is free to change the interface or functionality it inherits in arbitrarily complex ways. Of course, each line must eventually be merged and hence made consistent with the others, leaving a single, unified specification as a product.

Feather notes that the merging of elaboration lines may span a range of effort. In some cases, two lines may be independent, and hence it may be possible to take their union as a merged state. In other cases, components of two lines may interact in complex, possibly conflicting ways, often requiring domain knowledge to choose among various options.

Feather argues for the parallel development model on the following grounds:

- Each elaboration line separates development concerns. Thus, during construction, the specification process is simplified. As one might expect, the same benefit is gained in later attempts to explain and understand the specification.
- The model promotes a gradual refinement. In some sense, one can view each separate elaboration line in Feather's model as conforming to a Goldman style development, with a finite set of specification operators gradually refining some portion of the specification. In this view, it is an extension of Goldman's model, one which addresses more of the complexity of building non-trivial specifications by attempting to better structure and simplify the process.
- The combining or merging of parallel lines forces one to explicitly consider (and document!) the interaction between various components of the specification.
- There is a potential for reuse. Feather argues that it may be possible to introduce abstract specification components (e.g., from a catalog), and use the refinement and specification-tailoring tools already present to work the component into the current problem.

To further illustrate the first three points, he describes a pencil and paper design of a specification of a patient monitoring system using the parallel development approach (see [19] for other work on the same problem). Figure 3 gives a graphical representation of the initial problem. The representation used in this figure, and throughout the remainder of this section,

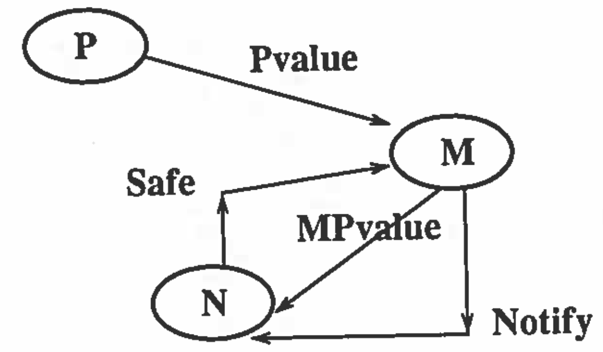

Figure 3. Feather's Initial State

is part of the Oz system discussed in more detail shortly. In words, as interpreted by Feather from the original problem statement [40] and paraphrased from figure 3,

> There are one or more patients **P**. Each has a value **Pvalue** (a crude approximation of that patient's medical status). There is one nurses's station **N**. It contains the safe range for a patient's value. There is one monitor **M**. It inputs the **Pvalue** of each patient **P**, and passes this value along as **MPvalue** to the nurse's station. The nurse's station determines whether **MPvalue** is within range, and sets the boolean flag **Safe** accordingly. If the monitor receives an unsafe signal from the nurse's station (i.e., **Safe** is false), it signals the nurse's station using the boolean flag **Notify**[8].

The monitor **M** is the component of the problem to be implemented; the other components are considered as part of the environment, and are included to provide a *closed specification* (see [3, 12] for arguments in support of such specifications).

[8]We are using Feather's original problem statement as is, even though it seems a bit odd that the nurse's station cannot ''notify itself'' when it detects an unsafe value.

The final specification developed by Feather is shown graphically in Figure 4. To paraphrase it,

> There are one or more patients **P**. Each has a value **Pvalue**, which in turn is broken out into a set of factors **PFvalue**. A device **D** reads these factors, and passes on the values in a form acceptable to the monitor **M**. Because the device may fail, it passes on a separate flag to the monitor describing its current condition. The monitor samples the values of **D**, i.e. **Dvalue** and **Fail**, beginning at **StartTime** and for a length **Period**, both stored in a database **DB**. The value read by **M** during this period is stored in the **DB** as **Store**. There is one nurses's station **N**. It contains the safe range for each factor. The monitor **M** passes the **LastValue** of each factor to **N**. The nurse's station determines whether **LastValue** is within range, and sets the boolean flag **Safe** accordingly. If the monitor receives an unsafe signal from the nurse's station (i.e., **Safe** is false), it signals the nurse's station using the boolean flag **Notify**.

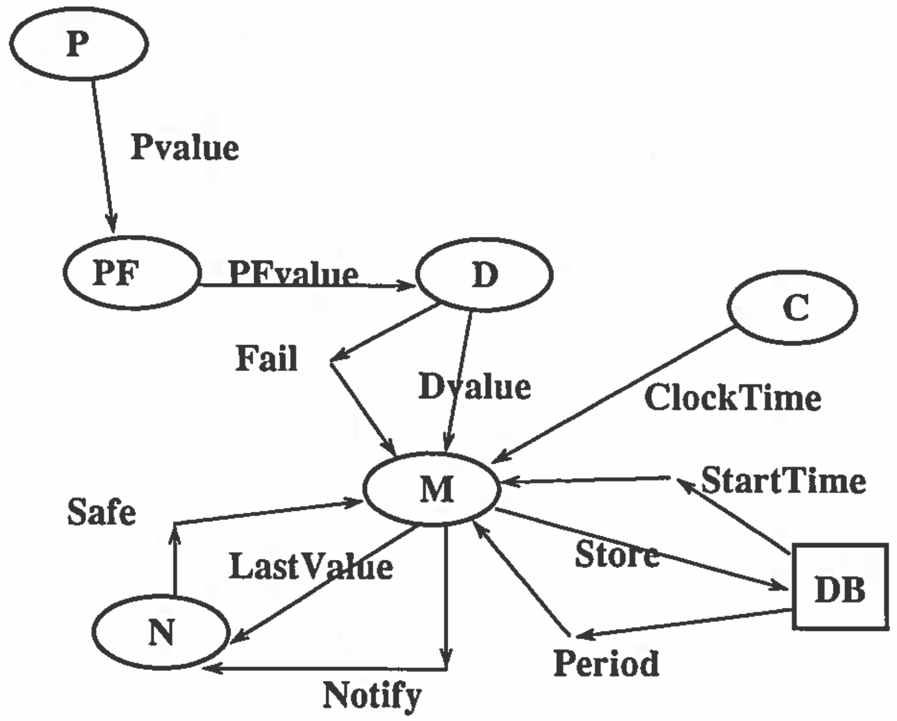

Figure 4. Feather's Final State

Figure 5 shows the top level elaborations that were used to arrive at this specification. In particular, the initial specification in Figure 5 was elaborated along 4 separate lines -- *devices, factors, clock* and *safety* -- as follows:

- *Devices.* The monitor cannot read the patients' value directly, but instead must rely on an intermediate A-to-D device to read **Pvalue**. This introduces **D** into the diagram.
- *Factors.* A patients' health can be measured by various vital signs. Thus, **Pvalue** should be factored into **PFvalue**, which represents a vector of values for each patient. Note that both **P** and **PF** are modeling the patient now.
- *Clock.* It is unrealistic to assume that the monitor can read and react instantaneously to patient values. Instead, we will have the monitor read values periodically, using a clock to synchronize.
- *Safety.* Instead of a single boolean flag **Safe**, we need a safety flag for each patient. In Gist terms, we will change **Safe** to a parameterized relation on **P**.

Once each of these four top level elaborations have been identified, further refinement can be carried out along each line, e.g., specify that devices may fail, specify that values will have to be stored between clock ticks.

The final step is to combine all of the separate elaboration lines back into a single state/specification. As Feather notes in [13], there are three interesting merge interactions to consider in the patient monitoring problem:

1. In combining *devices* and *factors*, what is the mapping from factors to devices, i.e., one-to-one (one device for each different factor), many-to-one (one device per patient), many-to-many (some combination)?
2. In combining *factors* and *clock*, what is the mapping from factors to sampling period, i.e., one-to-one (a different sampling period for each different factor), many-to-one (same sampling period for all factors), many-to-many (some combination)?
3. In combining *device failure*[9] and *clock*, should one worry about moving from a continuous monitoring of a device for failure to a periodic one? In other words, we introduced the clock because the monitor did not have the processing power to continuously monitor **D**'s output of patient values. Using this same analysis, shouldn't we worry about the monitor's ability to check **D** continuously for failure?

Given the apparent advantages of Feather's model, we looked for support of the model's claims through the construction of a tool that would allow us to 1) recreate the example development in a more formal manner, and 2) explore the details of the model that were missing in the informal description provided in [13]. The next section describes this effort.

[9]Device failure introduces the notion that the devices **D** are imperfect, and that the monitor can determine when one is malfunctioning. The Notify flag is further parameterized to show what type of problem is detected by the monitor, i.e., malfunction or patient value unsafe.

3.1 Model implementation

After studying the hand development provided by Feather, we were interested in three questions: 1) could a tool be built that would manage the parallel development design process, 2) could some parts of the process by automated ala Glitter-S, and 3) given that a merge process is inherent in the model, what were the issues in combining parallel lines beyond those brought out in Feather's paper? A system called OZ was constructed by Robinson to explore these questions [38]. The OZ system contains five major components:

Component 1.	A simple specification language based on a subset of Gist. The patient monitor figures shown in this section are a diagrammatic view of this language.
Component 2.	An elaboration manager that maintains parallel development lines.
Component 3.	A set of editing commands for refining a single elaboration line.
Component 4.	A specification maintenance component that attempts to clean up other parts of the specification affected by an editing command.
Component 5.	A "merge editor" for managing the merger of parallel development lines.

To illustrate these components in more detail, we will look at a portion of the development of the patient monitoring specification as carried out by OZ. To provide context, Figure 5 is a snapshot of a set of elaboration lines and states generated during the development of the patient monitoring system. In this figure we can see the four initial parallel lines suggested by Feather, along with further line splitting and merging. The diagram in this figure is a paraphrase of the graphics generated by a component of OZ that helps a user track and maintain the elaboration graph (see component 2 above).

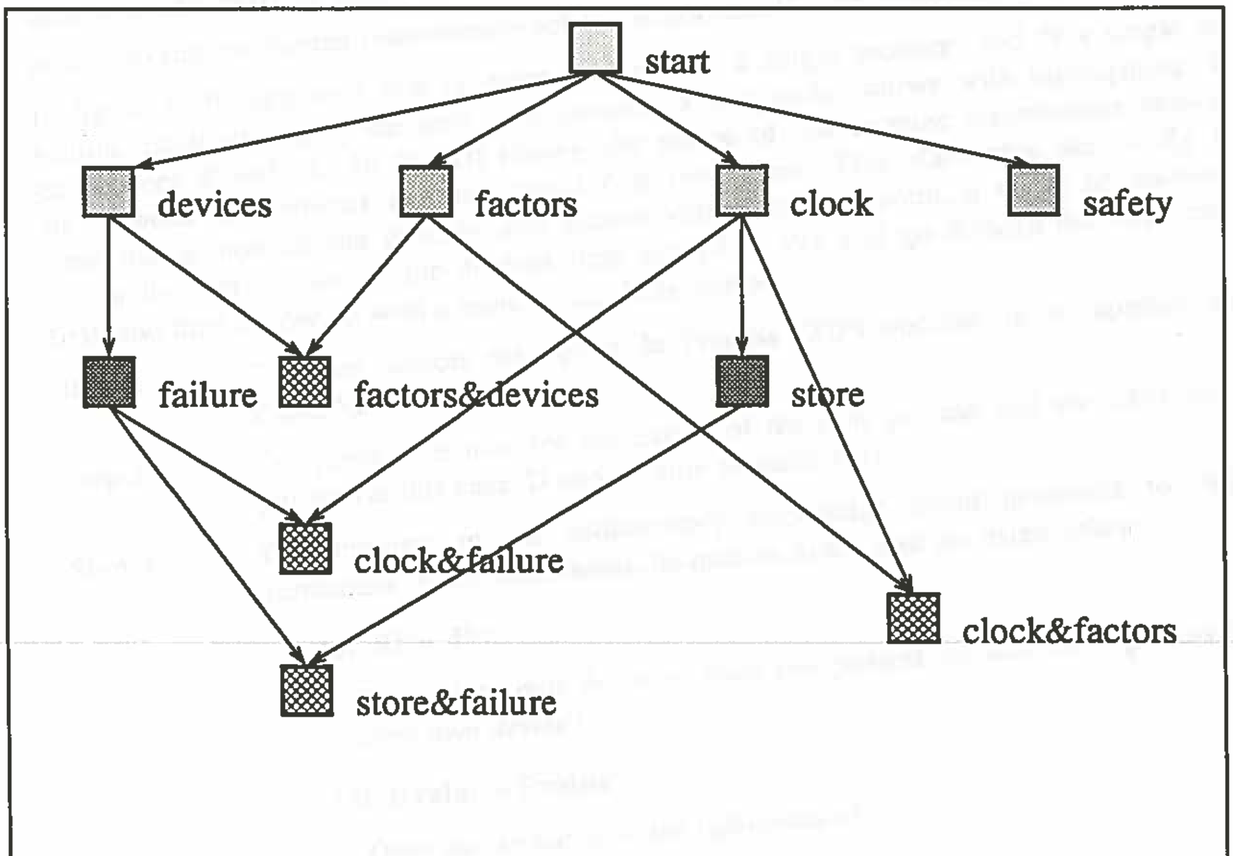

Figure 5. Development snapshot

Next, we will look at a subset of the states in figure 5 in a finer grain of detail. Figure 6 is an enlargement of the root state and two of its siblings, devices and factors, both taken from Feather's original development. The diagrammatic view of each state is a pretty-print of components of the OZ specification language:

- Nodes represent Gist processes (demons or procedures). OZ *does not* represent the full functionality of each process, but instead just the data dependencies between input and output.
- Labeled arcs represent newly inserted database relations. Not shown in the diagram, but represented by OZ, is further information about the cardinality and argument types of each relation.

In summary, an OZ representation can be viewed as a type of data-dependency graph of a Gist specification. While this simplified language captures only a modicum of the full Gist language, it nevertheless has been useful in at least two ways: 1) it is understandable by the user, and 2) it is understandable by the machine. For the latter case in particular, OZ con-

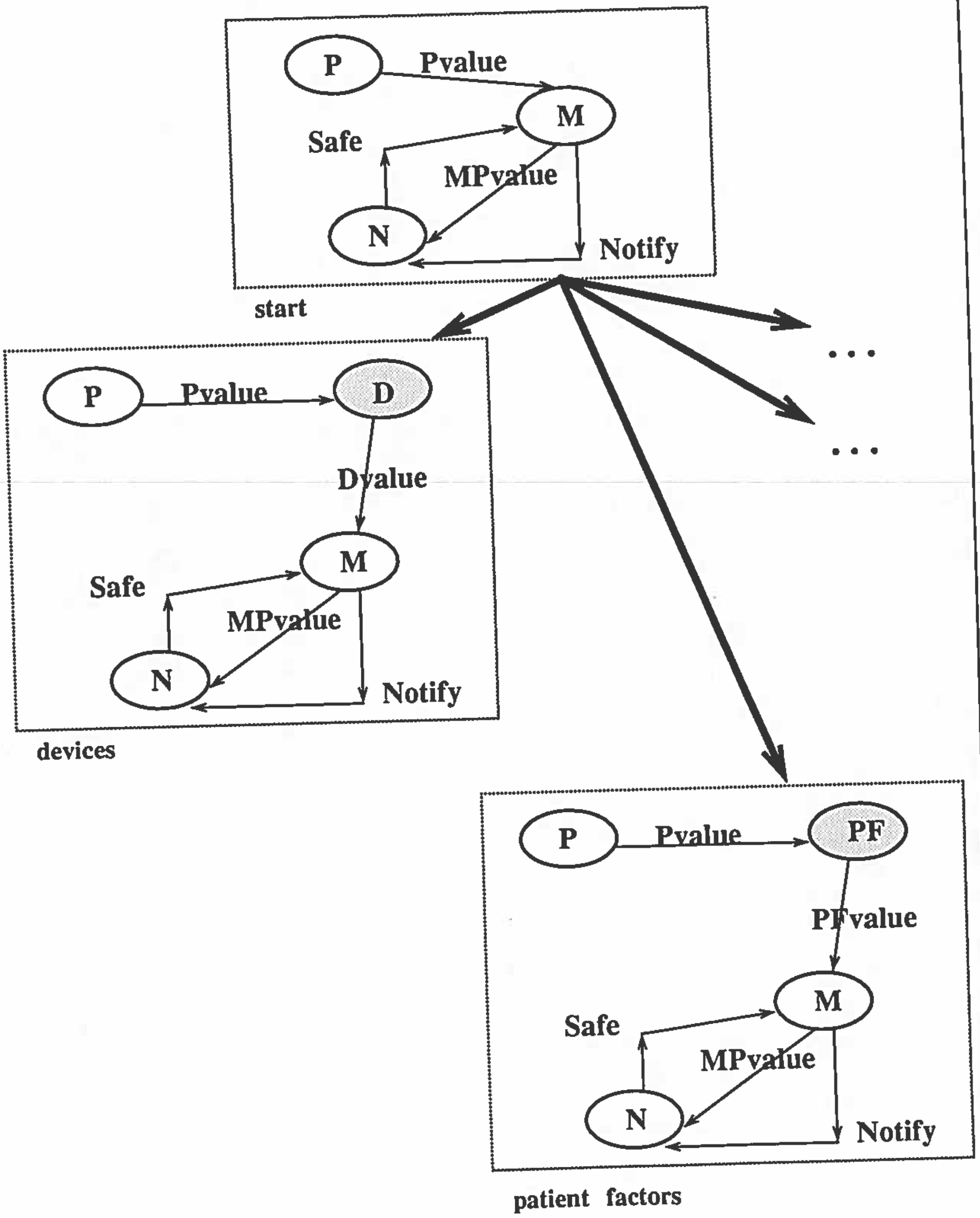

Figure 6. Development-state enlargements

step 3. This leaves us with the first query to account for. An operator in OZ can be implemented by more primitive operators. In the case of SIP, two primitive operators, Add_Process and Add_Arc, are called on to implement a part of the operation. Each of these operators in turn may have a rule-set associated with it. In particular, Add_Arc has a rule that worries about process cardinality, prompting the first question in step 3.

For the right sibling in figure 6, the same basic 4 step process is followed using an operator similar to SIP called UNPACK. For this elaboration, we note that a patient actually has a number of factors (i.e., vital signs) that are important to monitor. We model this by using a patient-factor generator **PF** to unpack the single patient value (in this view, **P** and **PF** are both components of the patient). As with the application of SIP, attached rules will query the user for more information. In particular, because UNPACK calls SIP, we will see the same three queries (in a slightly different form) as in the first elaboration. The UNPACK operation itself has an attached rule for acquiring the new composite set. Thus, **PFvalue** in figure 6 represents a set of arcs.

The last component of OZ that we will look at is the merge editor. We will use the merge of the **devices** and **patient factors** states just constructed as an example. Looking at Figure 7, we can see that there are three major problems confronting us. First, we must decide which processes and arcs in each state are in correspondence. Second, we must identify any conflicts. Third, we must resolve those conflicts. OZ finesses the first problem by relying on the user to explicitly select corresponding objects from each state. In this case, the user specifies that the components **P, Pvalue, M, MPvalue, N, Notify** and **Safe** in both the left and right states in Figure 7 are representations for the same objects.

The second problem is to find conflicts; if none exist then a simple union of the two states may suffice. In our example, the following problem arises: given that two different processes have been spliced between **P** and **M**, which should go first in the merged state, or should they go in parallel? Oz provides no help in answering this question; it is up to the user to explicitly tell OZ how to arrange the splices.

Given the user chooses to splice **PF** and **D** as shown in figure 8 (and as done by Feather), should there be separate device for each patient (as in **devices**), or a device for each patient's factor, or a device for a single factor of all patients, or a device for a subset of factors? Oz does indeed query the user for this information. It does so through the mechanisms we have already seen in applying an operator to elaborate a line. In particular, the construction of the new merged state in figure 8 is carried out by operators SIP, UNPACK, etc.,. The rules associated with these operators generate queries about the new state that force attention on two key aspects 1) how do the new components in each state interact with one another, and 2) what are the side-effects of the newly concatenated processes on the remaining processes in the specification?

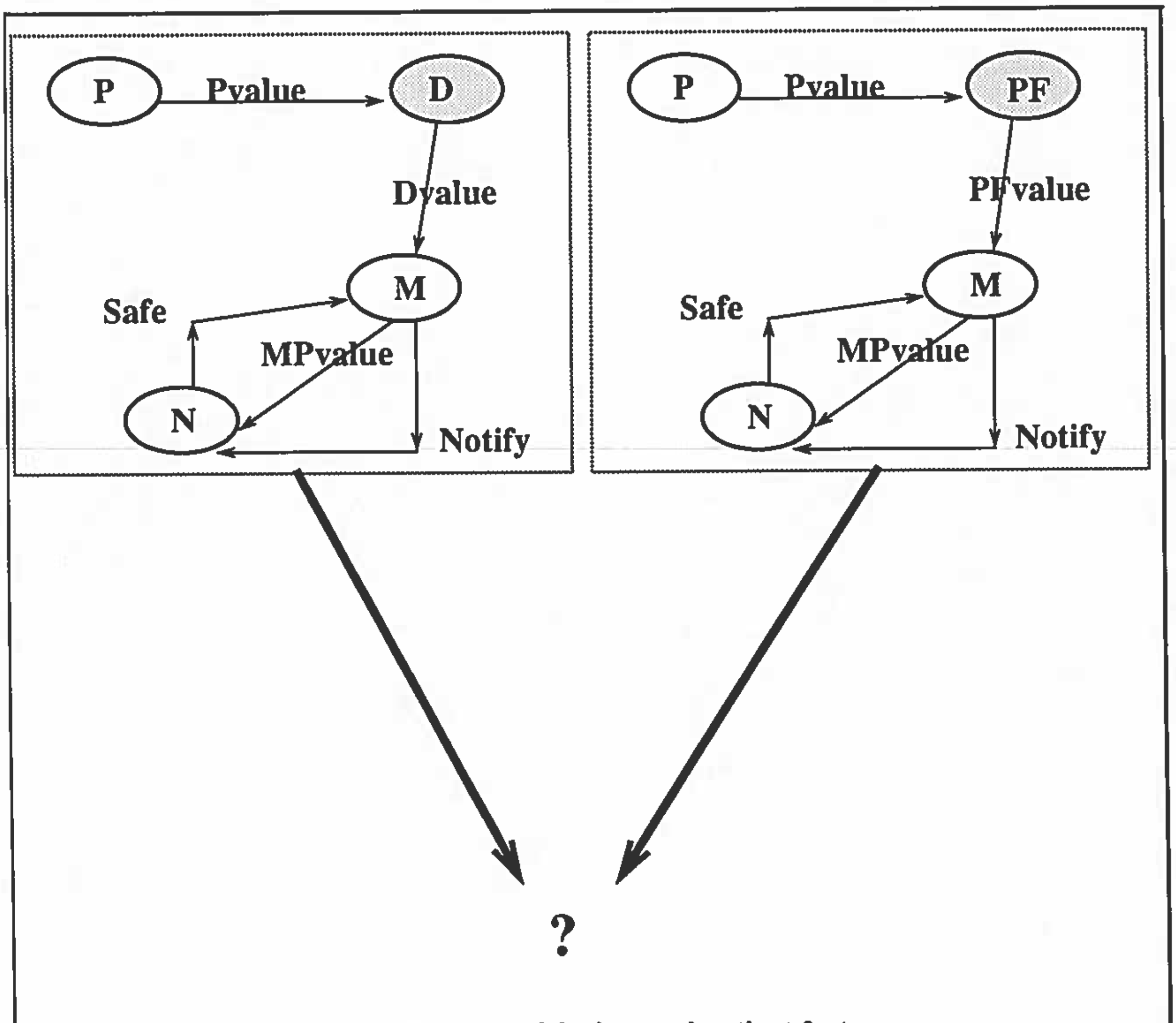

Figure 7. Merge of devices and patient factors

Figure 8 shows the resulting state.

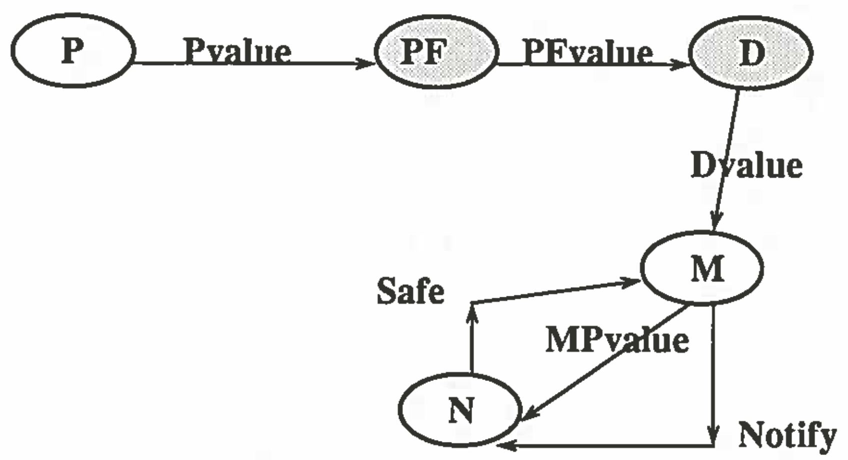

Figure 8. Merge results

3.2 Discussion

To reiterate, our three questions of interest were 1) could a tool be built that would manage the parallel development design process, 2) could some parts of the process by automated ala Glitter-S, and 3) given that a merge process is inherent in the model, what were the issues in combining parallel lines beyond those brought out in Feather's paper? The results of the OZ effort provide useful answers to each, but raise still more questions.

First, the OZ system is an existence proof for question 1. However, our attempts to apply the parallel development model to a more complex problem, and our careful recording of the actual design steps carried out, have convinced us that there is a major omission in OZ, that of realistic design or what we call dirty design. In particular, there is no representation of the dead-end paths, alternative designs, nor cut-and-paste behavior that seems inherent in any non-trivial design problem. The Glitter system provided a rudimentary form of an exploration tree, and it appears to us that something similar will be required here. We discuss this problem in more detail in section 5.4.

In regards to automation, we would argue that OZ was successful in the application of elaboration operators in the limited context in which it was applied. Robinson was able to define a simple action-oriented ''goal'' language (e.g. SIP, UNPACK), and provided simple implementation of goals in terms of other more primitive operations (e.g., ADD_ARC, ADD_PROCESS). While he did not provide a jittering component ala Glitter, he did build a cleanup or maintenance mechanism, in the form of attached rules, for leading the user through a set of potentially interesting post-elaboration questions.

Of course, the simplicity of the OZ specification language must be considered. Operating on and maintaining an extended data-dependency graph is a far cry from dealing with the full complexity of a language like Gist. On the positive side, not a small amount of useful automation was gained with this simple representation. This appears to be an encouraging result: we may be able to gain immediate automation benefits if we are willing to allow our tools to work with simpler representations of our full languages, leaving the more complex analysis to the human user.

The automation of the merge operation in the parallel development model remains a key research problem for our group. In particular, given unlimited splitting of new lines, and unconstrained divergence of each of those lines, can we hope to bring all lines back into a single, coherent, specification? Robinson noted the three hard problems: finding what is common in the two states, finding what is in conflict, and finally, resolving the conflict. The OZ system leaves each of these problems in the user's hands. Once the user has answered each question, the system does use a clever form of post-merge analysis by reusing the same cleanup rules used in normal elaboration. Our most recent efforts in addressing the merge issue are discussed in section 5.3.

3.3 Other questions raised

The construction and testing of OZ raised a set of new questions regarding the parallel development model:

Choice of the initial specification. This remains an open question in the pure elaboration model. However, it has not been a difficult task, at least in the small number of problems we have studied, to come up with a minimal description. Thus, while we might later wish to catalog a group of root nodes for various application domains, the lack of mechanized support for choosing the starting specification does not appear a crucial problem at this point.

Choice of elaboration lines. There may be many means of viewing a problem within the elaboration model. The patient monitoring problem was simple enough where this was not a particular difficulty. However, our study of a larger problem, a university library system, made it clear that choosing the break-out of the problem was important. At the very least, we would like support for exploring various elaboration options.

Choice of operator. The question here is the degree of automation one expects from a computer-based assistant. In OZ, the user is responsible for stating rather specific elaboration goals. Hence, it is up to the user to do the more demanding problem solving necessary to decide how an elaboration line will proceed, and then find the OZ commands that will bring it about. In the Glitter system, as introduced in section 2, the system was able to make some of the elaboration/refinement choices automatically, given a general development goal posted by the user. The question is whether we can expect to come up with a user development language such as Glitter's, i.e., one at a high enough level to allow useful problem solving by the system, in the domain of specification design. For instance, we might like to tell an OZ-like assistant "Fill in the details of devices that fail", and expect the assistant to choose what elaborations are necessary to extend the specification, using its knowledge of domains that contain imperfect physical devices. While this command can be viewed as residing along the "case analysis" dimension suggested by Goldman, the knowledge necessary to carry it out is clearly moving away from Goldman's notions of *domain-independent* refinements.

We also note the issue of elaboration order, i.e., is the model overly sensitive to the order in which elaborations are carried out. If so, then we may spend all of our time attempting to find the right elaboration *sequence* as opposed to the right elaborations. While we cannot offer a formal proof of Church-Rosser properties here, our experience with the patient monitoring problem and a library problem suggests that the parallel development model is relative insensitive to elaboration order (this claim was also made by Feather in his paper [13]).

Support of abstraction. Feather's model is based on the notion of evolutionary development coupled with parallel, focused views. While both of these ideas give us a powerful type of complexity management, we have not addressed the more traditional notion of design simplification based on abstraction coupled with step-wise refinement. It can be argued that the step-wise refinement approach is founded on having a *complete* (but abstract) specification to start with, and thus would naturally *follow* the design of the specification. Or alternatively, that abstraction is a language issue (see section 6.1). However, we believe these are glib dismissals: we have also argued that specification is a design process, and it seems clear that abstraction is a useful design technique. The question then becomes its role in the parallel development model, or more broadly, its role in an evolutionary model, i.e., a model with an ever changing specification.

4. Critiquing a specification

The third specification system we will look at in detail is one that attempts to debug or critique a given specification. It is based on the following three assumptions: 1) the client may have only a vague idea of what he or she wants, 2) domain knowledge will be a required component of the critiquing process, and 3) the system will need some knowledge of the development process as a whole, including the design, coding, and maintenance processes, resource requirements, and constraints on the runtime environment. These three assumptions run counter to the view of analysis as a process of *translating* or *rephrasing* client intent to formal or semi-formal documents. In the translation view, the human analyst is taken to be expert in interviewing and model building skills, but ignorant of the domain. Lacking domain knowledge, the analyst focuses on syntactic error detection techniques similar to those one might find in a good compiler, e.g., missing inputs and outputs (parameters), interface mismatches (typing errors), unused data (dead variables), disconnected processes (dead code).

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

As we will discuss in more detail in section 5, the system we describe here is a limited prototype of a larger project. Our goal is to provide assistance to a software analyst in producing a formal specification. The project, called Kate [15], focuses on the following 3 components:

Component 1.	A model of the domain of interest. This includes the common objects, operations, and constraints of the domain, as well as information on how they meet the types of goals or policies one encounters in the domain.
Component 2.	A specification construction component that controls the design of the client's emerging specification. The Oz system is one of the tools we are studying to fill this role.
Component 3.	A critic that attempts to poke holes in the client's problem description (in the form of the specification being constructed).

In the next section, we describe a system that focuses on the first and third components, the domain model and the specification critic.

4.1 Model implementation

Our attempt to build a critic has taken us through several iterations. We will discuss the three major versions that came out of our efforts.

4.1.1 Critic-1

We constructed a tool made up of two pieces, a specification language and a rule-based critic. The framework of the tool consisted of a **model** part, an **example** part, and correspondence links between components in **model** and **example**[11]. The use of **model** was as a representation of a particular application domain D. The use of **example** was as the representation of a particular problem specification in D. Our specification language, used by both **model** and **example**, can be viewed as an augmented Petri-net representation that allows computable predicates on both arcs and transitions. The language also supports abstraction through a class hierarchy in a manner similar to Greenspan's RML language [22]. Initial versions used NIKL [24] as the basis for implementation; more recent versions use KEE/SIMKIT [25].

To use the tool, several manual steps had to be employed. First a representation of a particular domain had to be coded in **model**. As our initial domain, we chose the domain of resource management. We justify this choice elsewhere [15, 16], but note here that the domain covers a wide range of applications including management of human, physical and information resources, and forces one to deal with human users, staff personnel, resource overflow and underflow, security, and a host of other ''interesting'' problems (see, for instance, [9]).

[11] Because an example-model link is often hypothetical, each such link is made in a separate context. In this way, alternative mappings from example to model can be maintained, reflecting the frequent ambiguity in a client's description (for example, see section 5.5.2).

Second, the specification to be critiqued had to be represented as the **example**. The specification/**example** we will discuss in this section is that of an automated library system. The problem has become a standard one in discussing specification research. The particular incarnation we will use comes from the problem set handed out prior to the Fourth International Workshop on Software Specification and Design [19]; it is produced verbatim in Appendix A.

Finally, correspondence links were forged between components in **model** and those of **example**.

To reiterate, all three of these steps were carried out manually, i.e., by members of our group. Once **model**, **example**, and the correspondence links were in place, a rule-based critic, implemented in the ORBS language [17], was used to find components of **model** that were not linked to components in **example**. Running the tool on the problem description in Appendix A (after translation into **example** form), we were able to produce warnings such as the following:

- Missing actions: *mark-as-lost, add-to-group, remove-from-group*
- Missing constraint: *borrowing-time-limit*

4.1.2 Critic-2

At least one major problem we found with critic-1 was that there may be problematic components in **example** that did not show up in **model**. Specifically, **model** was meant to represent the typical components of a resource management system, and thus give advice such as ''X is a good thing to have, and you're missing it.'' It seemed we needed a representation of the way systems can also go wrong as well: ''X is a bad thing to have, and you've got it.''

Our solution was to divide **model** into good components (what we had originally) and bad components, still using correspondence links to tie components of both types into **example**. This approach allowed us to produce a new set of warnings:

- Problematic action: *who-has-what-query*
- Problematic constraint: *book-borrowing-limit*
- Problematic object: *borrowing-history-record*

4.1.3 Critic-3

It became clear by talking with expert analysts that there is really no such thing as an inherently ''bad specification'', only one that does not conform to policies in force. In particular, the goodness or badness of a component in **example** can only be judged *relative* to the goals or policies of the client[12]. Thus, the two-part partition of **model** in critic-2 is overly rigid. What we really need is 1) a broad representation of the basic components of the domain, 2) a

[12] We will henceforth use *goal* and *policy* synonymously. Further, we will include resource constraints, both on the development of an implementation and on the operational environment, as representable as policy decisions, e.g., ''minimize operational staffing costs''. See section 5.2 for further discussion.

representation of the policies of the domain, and 3) a way to show the relationship between domain components and policies. It is only after we have this third component that we can point to errors in the specification.

We built a tool that captured all three components listed above. This involved combining the two parts of **model** in critic-2 into a more general, single model. Further, a simple, hierarchical representation of policies was defined based on discussions with domain experts and a study of the resource management literature. Each policy can be in one of three states: *important, unimportant, unknown.* Finally, **model** components were linked to policies. Such a link can take on one of two values: *positive* (the component supports the policy); *negative* (the component thwarts the policy).

The critic's rules can now be more tightly classified into three types:

> **Type 1**: A policy marked as important or unknown is linked *positively* to a **model** component. No corresponding component can be found in **example.**
>
> **Type 2**: A policy marked as important or unknown is linked *negatively* to a **model** component. A corresponding component can be found in **example.**
>
> **Type 3**: A policy marked as unimportant is linked *positively* to a **model** component. A corresponding component can be found in **example.**

Type 3 is a weaker type of critique than the first two, and is based on the notion that components added to a specification in support of an unimportant policy will tend to add unnecessarily both to the complexity of the system, and to the cost of its operation and maintenance.

It is important to note that it is possible, and in fact typical, for the same **model** component to have both positive and negative links to some set of policies. That is, it may positively support policy A and negatively effect policy B. Further, while we would like to think that A and B are never both simultaneously marked as important, it is again typical for a client to describe a conflicting set of goals or policies, i.e., "I want it all." One critical process in specification design is then coming to grips with conflicting policies through various forms of trade-off and compromise. We will discuss this further in sections 5.2 and 5.3. Our critic currently does *not* note these conflicts, but simply presents the three types of warnings given above.

The tool, eventually dubbed Skate (for Small Kate), was run on the library problem; we discuss the results in the next section.

4.2 Discussion

As an assessment of Skate's performance, we presented a university library analyst with the same problem description as in Appendix A, and asked her to critique it outloud. We recorded this session in both audio and video form; an analysis of the transcript can be found in [18]. There were three things worth noting about the results. First, much of the session was taken up with establishing "the ground rules", i.e., the policies in effect. In Skate's critique, we left all but one of the policy nodes marked as of unknown importance to reflect the meagre information given in the text description. In our session with the library analyst, we felt compelled to fill in at least some of the policy details of the problem. The negative side of this is that we are no longer comparing exactly the same problem. However, there is a positive side as well: the policy questions raised by the analyst were at least superficially equivalent to ones in Skate's representation. Thus, we find support 1) for the general notion that a critique of a specification must take into account the overall goals and available resources, and 2) for the specific use of policies to capture this information in Skate.

The second thing to note from our results is the actual critique given by the analyst. First we will summarize the major findings of Skate's critique of the problem description in Appendix A (henceforth, WS), and then compare them with that of the library analyst. We remind the reader that Skate's criticism is given in light of only a single component-policy link; other links that might be used to counter the criticism are not taken into account.

1. Actions 4 and 5 in WS may cause problems with giving out user-confidential information. In general, any action that gives out information about a user's borrowing record, whether now or in the past and whether to the same user, or to someone else, is linked negatively to the policy of maintaining user privacy.
2. The WS constraint that the check out action must be monitored by a staff person[13] goes against the policy of minimizing operational staff. This policy was marked as important by at least one interpretation of *small* in the first line of the description.
3. The actions of adding and removing a book (see item 2 in WS) can be viewed in finer grain, e.g., remove-lost, remove-stolen, remove-damaged, replace-lost, replace-stolen, replace-damaged. These type of actions are linked positively in **model** to the policy of accounting for human foibles.
4. A borrowing limit (part of WS) is linked negatively to the policy of giving users a sufficient working set.
5. A borrowing time limit is linked positively to the policy of maintaining an adequate stock on hand. There is no corresponding component found in the description of WS.
6. The division of users into groups (staff and ordinary borrowers in WS) is without corresponding actions to add and remove members from a group. These actions in **model** are linked positively to a policy of recognizing the human dynamics of group membership.

The analyst produced criticism similar to 1, 3, and 4 above. Later questioning of the analyst about the other three warnings produced by Skate (2,5,6), and missing from her critique, can be summarized as follows:

[13] There is actually another interpretation of the constraint: only staff can check out books. While no human reader used this interpretation, we did run it through Skate. The outcome was a warning that the policy of giving library users access to needed materials was unsupported in the case of non-staff, i.e., ordinary borrowers.

- (2) She assumed that no library could be effective without some type of control on check out and check in. Thus, she was assuming that if we needed to minimize cost, it would not be through removing staffing from circulation. The message for us was that we would need some type of prioritization ordering on policies.
- (5) Her assumption was that you must give users both an adequate set of resources (prompting her criticism akin to Skate's in 4), and an adequate time to keep them out. Again, there seemed to be an ordering of policies that placed this above concerns of keeping adequate stock on hand.
- (6) These seemed so obvious that she assumed that they were omitted just as a matter of detail, and that any competent designer would include them (!).

In summary, the comparison of Skate's analysis with that of the human analyst reassures us that the representation of policies in a specification critic will be a key component. Our findings also point to the need for a more refined view of policies, their interaction, and their connection to domain components. This is discussed in more detail in section 5.2.

Finally we note that critic-3 has no automated means of extending **model**; this currently must be done by manual editing. While we are interested in a more automated acquisition process, success here would seem to rest on progress in the learning field in general [10].

5. The Kate project

In the proceeding portions of this paper, we have described three projects that have attempted to better formalize or automate the specification process. We have taken the results from each, and are using them in our larger project, Kate, to further formalize and automate software analysis. This section discusses the Kate project in more detail.

5.1 Use of the parallel development model

We have chosen the parallel development model as the organizational tool for building FrameNet[14] specifications. The bases of this decision were the results of the OZ project discussed in section 3, and the subsequent development of a much more complex specification, that of a university library system. In the latter development, described in [6], we kept a careful record, over a three month period, of the paper-and-pencil use of the parallel development model for building a FrameNet specification of an automated library system. We can summarize the positive results of this effort under two headings:

Choice of elaboration lines. We did find at least informal guidelines for breaking out elaboration lines. Initial elaboration of the root could usefully be tied to each of the users' perspectives. Thus, specifications would be built initially from each user's selfish viewpoint, e.g., a

[14]We have settled on a compromise between a process-based and frame-based representation for our specification language in the form of a Numerical Petri Net (NPN) [46] extended to include frame-type tokens. This frame-based NPN language, which we have dubbed FrameNet, has been brought up on top of the KEE/SIMKIT system [25].

line for the selfish borrower, a line for the selfish library administrators, a line for the selfish library staff. Once these lines were brought together, subsequent elaboration lines followed functional lines, primarily as a means of simplifying the problem.

Design as compromise. The merge of "selfish lines" was clearly where much of the action was. Virtually all of the library concepts that we could find could be traced to compromises between various special interests. The merge operation provided two important functions here: 1) it gave a focal point where conflict became apparent, and various conflict resolution strategies could be discussed, e.g., what compromises are available, what selection criteria should be used in choosing among them, and 2) it forced an explicit record to be kept of the decision process. As an example, when attempting to merge the selfish borrower line and the selfish library (administrators) line, a conflict arises between the borrower's wish to keep a book indefinitely, and the library's wish to have a needed book instantly available to a potential customer. The choice of resolution strategy here will give different types of libraries, e.g., *reference library* only if the administration is given priority, *department library* if the borrower is given priority, *university library* if a compromise such as recall-on-demand is chosen, *city library* if a compromise such as fixed-time loans is chosen.

In summary, we find Feather's claims are still substantially supported:

- Each elaboration line separates specification concerns.
- The model supports gradual refinement.
- The combining or merging of parallel lines forces one to explicitly consider (and document!) the interaction between various components of the specification.
- There is a potential for reuse. We will discuss this further in section 5.4.

However, there were negative results as well. In particular, we saw a lack of support in two important areas:

Merge assistance. The merge operation must be better formalized and automated. Part of this problem is necessary progress on the three sub-tasks pointed out by Robinson. Part is also identifying the various kinds of merge operations that one may want. For instance, in our design of the library model, functional elaboration lines often required communication between them, primarily to avoid wildly divergent work, but also to promote sharing of analysis. We dubbed this type of communication *crosstalk*, and found the need for a special type of incremental merge that would handle it.

Dirty design. The carefully recorded design steps of the library specification [6] pointed out the need for a more extended view of design management tools. In particular, the OZ system captured all of the elaborations, operators, and merging of a design, but only for a single solution. The Glitter system, viewing design as a problem solving process, captured the AND/OR tree, and thus multiple solutions, but left backtracking to the user and did not support reuse. Our experience with the library design convinces us that we need the full solution history kept by OZ, *and* a means of exploring and recording alternative designs ala Glitter, *and* a means of reusing parts of the design when backing out of false paths or constructing portions of the model that are similar to others.

Each of these two problems has spawned research efforts within our project. They are discussed in sections 5.3 and 5.4.

5.2 Representation of Policy

We argued in section 4.2 that our representation of policies seem to capture the type of goal or context information employed by human analysts, but lacked a notion of policy interaction. For instance, many components in Skate's **model** were linked to *conflicting* policies. It is clear from our protocol studies that each analyst had at least a partial ordering on the policies of the domain, an ordering that was used to choose among various components needed in the final specification. Our initial approach to handling this problem has been direct: define a partial ordering on policies, and pass this information along both to the critic and to the merge assistant in Oz. Unfortunately, we suspect that this is rather a naive view of a complex goal structure, and we will eventually need to look at more sophisticated representations such as the meta-goal and meta-plan notions proposed by Wilensky [48].

Besides concern with the interaction between policies, we see the need for further study of our representation of the policies themselves. In particular we believe, in pursuit of generality, we might be overloading the policy notion. At present, we use policies to represent a) the various goals of the users and administrators of the system, b) the resource constraints of the system, and c) a simple model of human behavior in resource management applications. For the latter in particular, concepts such as borrowers forgetting, borrowers stealing, borrowers gaining and using information illicitly, tend to interact with specification components in complex ways, sometimes requiring something more akin to a script-like representation. In fact, we are currently extending Skate to include the notion of a *usage scenario* to capture this knowledge. A usage scenario consists of 1) a set of FrameNet components that represent a particular type of user transaction with the system (TAXIS scripts [4] use a similar approach), and 2) an optional list of policies that are positively or negatively reinforced by the scenario. We employ usage scenarios to capture the way users and staff typically use and misuse a system in a particular domain. Because usage scenarios are tied into the components of **model**, i.e., they are instantiated with the objects, actions, constraints and policies represented in **model**, they rest by definition on domain knowledge. Some examples may be useful here.

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

> "A user U_j has checked out a resource R; U_i is in a state in which he or she can identify himself or herself as U_j; U_i queries the system for the resources checked out by U_j; U_i moves to a state in which he or she is aware of the identity of R."

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

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

When extended with this information, Skate is able to offer more complex critiques of a specification. For instance, the inclusion in Appendix A of a query to allow users to find out what books they have borrowed, and an assignment of value *important* (or *unknown*) to the privacy policy, will now prompt Skate to present the first scenario. If the user decides to remove the query, the second scenario will be presented.

Skate also issues warnings on partially matched usage scenarios. For instance, if we parse Action 1 in Appendix A to mean only staff can borrow, scenario 2 above has a partial match: the query is present for ordinary borrowers (from Action 4), but the check out action is missing for this group. In other words, ordinary borrowers can find out what they have checked out, but cannot check anything out! We'd like to claim that the system is being clever here. However, what is compiled out of such a critique is a representation of why the scenario is useful, or how its steps necessarily characterize it. Thus, we cannot generate the perhaps more satisfying explanation

> "Useless queries are bad. Any query that returns known information is a useless query. Any query that returns a known constant is one that returns known information. Transaction 4 [query own books] always returns "nil", and it is known that it will return nil. Transaction 4 is a useless query."

In other words, while we do capture the notion of a partially matched usage scenario representing a potentially incomplete or inconsistent view of the problem, it is really of a scenario-syntax nature. Overall, however, this expanded view of component interaction in support of policies seems to be a useful one, and worth further exploration.

5.3 Formalizing and automating merge

Our study of the library problem reaffirmed our notion that the merge operation in the parallel development model was of prime concern. There are three problems in merging specifications A and B: 1) finding the common components of A and B, 2) finding the conflicts between A and B, and 3) resolving the conflicts.

The first problem is currently finessed by our system: we rely on the user to provide the registration between the two specifications. While we can conjecture that the operator history of each line, along with their common ancestor, may provide help here, we have yet to attempt to use this information in our system.

For the second problem, we are attempting to extend work in the version control area to conflict detection. In particular, Horwitz, et al have devised an algorithm for merging *non-conflicting* code modules in a version control system [23]. As part of this process, they must determine if conflict exists. Robinson has found a way to extend this work to identify conflicts for simple types of merges in the parallel development model. As part of his thesis work, he is studying means of generalizing his algorithm to handle more complex cases.

For the third problem, there are two questions to consider: 1) what conflict resolution strategies are available, and 2) what criteria can be used to select among them. Given that the type of conflict has been classified in the previous step, our goal is to have a set of strategies cataloged in Kate to resolve it. Thus, if a specification A has a process defined for gaining access to a resource, and specification B has no such process, we might consider the following strategies:

1. Prefer A over B, i.e., allow unlimited access.
2. Prefer B over A, i.e., allow no access.
3. Compromise between A and B, e.g., allow limited access.
4. Add a new component not found in either A or B that satisfies the concerns of both, e.g., add new resources as they become depleted.

Clearly the choice among these must depend on the rationale for the existence of an access function in A and the lack of one in B. In our view, we are back to issues of policy and the ordering among policies. For instance, the access function in A may be motivated by a policy of giving users what they need to perform a task. The lack of an access function in B may be motivated by a policy of maintaining a useful stock on hand. Various types of resource management schemes can be specified by the choice of a strategy above, each making a commitment to the weight they give to the policies of A and B. Note that while strategies 3 and 4 treat A and B as of equal weight, strategy 4 introduces a new problem of producing new resources on demand; its selection must depend on yet still other concerns with the ability to acquire such resources (e.g., buy multiple copies of popular books, find funds to replace lost or stolen books) when the system is put into operation.

There are two approaches that we are exploring to address the issues above. First, we are extending Oz to include the representation of policies discussed in the previous section. As we have argued, we will need this information as the basis for selection among merge strategies.

Second, we are using the Glitter model in representing conflict resolution as a problem solving process. Thus, an initial Glitter goal might be of the form *resolve-conflict(X,Y)*. Conflict resolution strategies are stored as Glitter methods. Each strategy/method may post further goals (e.g., *choose(X), choose(Y), choose-compromise(X,Y)*), finally leading to actual elaboration operators. Selection criteria, e.g., *priority-ordering(X,Y), maintain(X), maintain(Y)*, find a home as Glitter selection rules. Finally, Glitter maintains the problem solving history as a permanent record, thus documenting the design decision that lead to a particular merged state being created.

Of course, questions abound here: can merge strategies be defined in domain-independent terms (our example is tied to resource management problems); given a general strategy, how are its details implemented (e.g., how is *choose-compromise* achieved in terms of lower level methods and goals); are there a tractable number of strategies and a feasible way to organize them? In summary, while Glitter appears to provide a useful representation, we still must decide what to represent. As part of his thesis work, Robinson is looking at these issues.

5.4 Dirty design

Our careful record of the use of the parallel development model on both the patient-monitoring specification, and an automated library specification carried out later, brought to light the following types of design processes:

- *Generative design.* The designer gradually extends the specification by choosing elaborations and operators, and generating new states until the specification is sufficiently defined. This is the ideal type of design process implicit in specification development systems such as those of Goldman [20] and Feather [13].
- *Alternative design.* The designer recognizes that the current specification, as designed, may be inappropriate (for any number of possible reasons), and an alternative is spawned.
- *Design patching.* A design previously thought to be satisfactory is later recognized to have deficiencies. Instead of starting an alternative design, missing pieces are patched into the existing design at appropriate places. In his Paddle system [47], Wile addresses some of the issues involved with patching a transformational implementation, which is not dissimilar to the elaborations of Goldman and Feather.
- *Design reuse.* To extend the current state, the designer pulls a piece of design from another place, and appends it. This other place can include another elaboration line of the same design or a piece of an alternative design (a type of design salvaging).
- *Analogical design.* More than once we found ourselves looking at the design of the patient-monitoring specification for guidance on the library problem. While we did not explicitly attempt to map pieces from one to the other, we speculate that with the proper set of support tools, this could have been a powerful form of design.

Most tools support only the notion of generative design. The assumption is that the designer will make all the correct choices along the way, and that duplication of effort is to be tolerated. We would like to support a more realistic view that takes into account the other design processes we have listed above, collectively what we are calling *dirty design* to contrast it to the pure, generative model.

As a start, we have extended the Oz system with a representation of alternatives and a means to move about them. In a related research effort, we are attempting to characterize and support the type of copy-and-paste and patching processes that were observed so frequently in the design of the library specification. We are currently able to support a very simple type of copy operation that takes a starting and ending state (and the operations in between) from one design alternative and attaches it to the leaf state of another (the user must supply the correspondence between objects in the design being copied and the design being extended or patched). If conflicts arise after attachment, for example an operation from the old design does not apply in the new context, the user is asked to manually repair the state. Our goal here is to remove the user from the repair process by relying on the system to ''fix up'' the current state so that copied or patched designs can be integrated. However, even a primitive copy operation provides, if nothing else, a more explicit record of the ''real'' design process that we observed taking place in our construction of specifications.

In a longer term view, we have become interested in reusing designs not only within a specification, but across specifications and eventually across domains. John Anderson is exploring the concept of analogical mapping of designs as part as his Ph.D. thesis.

5.5 Interactive Design

Our long term goal is to produce an interactive assistant for acquiring a problem from a client and designing a formal specification. Towards this end, we have begun to integrate our work on Oz and Skate, giving us a specification tool with the ability to critique an evolving design. However, there is much to be done to move from what is basically an extended editing paradigm to a model of active design assistance. Our preliminary studies have allowed us to identify three of the processes that we believe will be necessary in such a model: problem acquisition, disambiguation, and example-based critiquing, . We discuss each briefly below.

5.5.1 Example-based critiquing

The results of the Skate project convinced us that a knowledge-based critic could play an important role in debugging a specification. The results of studying expert human analysts provided clues on how a critique might effectively be delivered. In particular, the library analyst that we studied was able to give a much more detailed analysis of a problem than Skate, and produce a seemingly unlimited number of examples on demand. In our view, she had 1) a deep understanding of the ''first principles'' of library science, 2) a knowledge of specific libraries and thus specific examples, and 3) the ability to generate hypothetical examples to illustrate some abstract principle or policy. A student on our project, P. Nagarajan, is attempting to integrate the Skate work with this finding. Specifically, his interest is in building a critiquer that is able to present a critique by way of examples. His approach, similar to Rissland's in the legal domain [35, 36, 37], is to catalog a set of interesting cases in a generic form. Attached to each case is the type of bug it represents, and ways of generating an example that demonstrate the bug to the client. To critique a particular state S in Oz, Nagarajan's system first matches FrameNet components of S against cataloged cases[15]. If a case C matches, an example is generated in S that can include either concrete or abstract frame tokens for execution purposes.

Nagarajan has built a small prototype critic system that can analyze simple resource underflow problems. An example may be illustrative here. An underflow case is stored in the system as 1) a FrameNet *place* that holds a limited number of FrameNet resource *tokens* (nee objects)that are checked in, 2) a *place* that holds one or more borrower *tokens*, 3) a *place*

[15]As with the merge problem, the matching task here is difficult. Currently, the system uses simple name correspondence when possible, and otherwise, queries the user.

that holds zero or more resource *tokens* that are checked out, and 4) a FrameNet *process* that takes as input *tokens* from checked in resources and borrowers, and moves the resource *token* to the checked out *place*. Figure 9 represents the case.

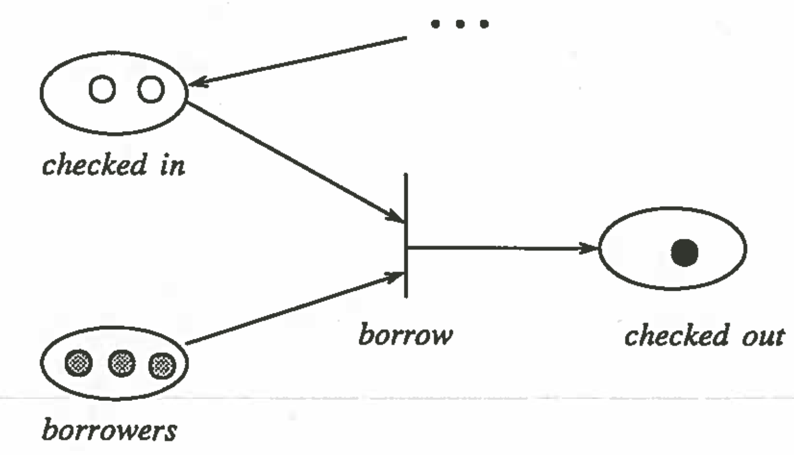

Figure 9. Resource underflow case

Attached to this case is a scenario for showing its dire consequences, i.e., that available resources can soon be depleted. The scenario includes the initial placement of tokens to start the scenario, and a directive to the non-deterministic control mechanism of the net to fire the borrow process whenever possible, i.e., constantly until no more resource tokens are available. Along with the execution trace of the scenario (using SIMKIT's animation tools), the system also generates a canned text summary of the problem for the user.

Note that Nagarajan's system is attempting to demonstrate an extreme case here. For instance, other processes might exist in S for replenishing resources either by checking them back in or buying new ones (as represented by the elision of an input arc to *checked in* in Figure 9), or even removing borrowers from the pool. However, at least in terms of FrameNet's non-deterministic control, an unrestricted borrowing process *could possibly* lead to the case shown. It is the analyst's and client's decision whether it is worth building in further restrictions (e.g., by attaching predicates to the borrow process) to prevent the case. Nagarajan is also looking at the representation of best-case and average-case scenarios, thus extending the type of examples that can be generated, and coming closer to Rissland's notion of a continuum of cases to consider for any specific issue [36].

5.5.2 Disambiguation

A high level problem description, such as the one in Appendix A, is of mixed benefit. It clearly reduces the amount of detail that must be specified. For instance, there is no need to define what the words *library*, *check in*, *check out*, *add*, *remove*, *query*, or *book* mean; they are all assumed to be obvious to the reader. However, this opens the door to ambiguity and misunderstanding. Specifically, client and analyst may each maintain a different interpretation of the problem while under the illusion that they share the same interpretation. For instance, the following ambiguities from Appendix A lead to different interpretations by various readers of the text:

> **Ambiguity 1**: The word *small* in the first sentence can be interpreted several ways, e.g., the library collection is small, i.e., a small-library database, the database will remain small, i.e., a small library-database (which is at least partially subsumed by meaning 1), the supporting environment is small, i.e., a small-time operation, or small is synonymous with simple, i.e., a simple problem involving a library database[16].
>
> **Ambiguity 2**: When the class *users* is broken into *staff* and *ordinary borrowers*, does *staff* refer to library staff or organizational staff? We parsed it to mean library staff; the library analyst parsed it to mean university staff as a whole (i.e., there are faculty, staff, and students at a university).
>
> **Ambiguity 3**: A constraint on the *check in* and *check out* actions states that only *staff* are allowed to perform these actions. Does this mean literally that *ordinary borrowers* cannot borrow? Or does it mean that a staff member must be in charge of these actions?

Using formal specification languages instead of English text only partially addresses the ambiguity issue; someone must still interpret the client's description and map it into formal terms, e.g., our mapping of Appendix A into a Skate **example**. Expert human analysts seem to be aware of this problem. From our protocol studies [18] we have found that they use a variety of means to insure that a consistent understanding is maintained. Below are three exemplary samples from our transcripts:

1. **Analyst**: [*discussing meaning of removal of book*] OK, and then removing it would be, say, it gets old, or it gets torn up or it's missing pages [*client agrees with meaning*].

[16]Kemmerer's original description [26], from which the workshop problem was taken, gives no clue since it reads "The example system considered in this paper is a university library database", obviously not a small-library database, nor a small library-database, nor a small-time operation, nor a simple problem. We chose to take the meaning small-time operation in terms of staff, i.e., we marked the policy of minimizing staff as

2. **Client**: Everything [*related to book ordering*] is centralized.

   Analyst: OK. So they order things for you,

   Client: Um-hmm.

   Analyst: and then process them,

   Client: Yes.

   Analyst: and so you get a book that's been processed and has got a book card in it,

   Client: Um-hmm.

   Analyst: a check-out card, and it has the set of cards and you file those.

   Client: Right.

3. **Analyst**: How many, what's your patron count? How many people use the library?

   Client: 1300.

   Analyst: About 1300 in the school. OK. [*Goes on to find the actual usage.*]

All three examples above show the power of domain expertise within an analyst. The first two rely on knowing the details of domain terminology, while the third is based on an expected value range. For the latter in particular, our supposition is that the analyst expanded the client's ''1300'' response because it was a surprising answer taken literally, i.e., that 1300 students of a high school were library users. More likely the client was giving the total number of potential users.

5.5.3 Problem acquisition

Problem acquisition must be at the heart of any specification design tool. We observed three types of acquisition techniques that were common across our expert analysts: 1) short-answer questions, 2) example-based questions, and 3) summarization-based questions. For the first, typical behavior was to open a session with a set of stock questions. The ''patron count'' dialog in the third example above exemplifies this behavior.

Example-based questions were also typically of a short-answer variety, but were lead off with a scenario to set a context. Several examples are given below:

1. **Analyst**: OK, how about on-line, public access, so that a student could come in and rather than go through the card catalog, could search your holdings on a terminal [*do you want this feature*]? ...

2. **Analyst**: Since we are in a campus setting, talking about a departmental library, I would have to ask what ordinary borrowers are we distinguishing between. For instance, a senior working on a honors thesis may need the same type of borrowing privileges that a graduate student needs [*should we refine the notion of ordinary borrower*?]. ...

One thing in particular is evident from interactions like those above: the analyst is often driving the design. To model this behavior, Nagarajan is studying the use of his catalog of cases as the basis for a synthesis tool. Thus, in a similar way to that proposed by the Programmer's Apprentice project [45], the same cases (or plans) can be used for both analysis and synthesis. The goal is to use the generic FrameNet cases to generate examples of potentially useful specification components to be confirmed or denied by the client (see both examples above).

The third acquisition technique, summarization-based questions, was typically directed towards completeness checks and further refinement, as shown by the following two examples:

1. **Analyst**: And so a student's file isn't considered complete until they have all the things. Are there other things other than letters of recommendation, test scores, statement of goals, transcript, and application [*Summarizing file contents discussed to date.*]?

   Client: No.

2. **Analyst**: OK. Um. I don't know how far you want us to take this, at this point. I think I know basically what the process of graduate student admissions and application is.

   Observer: Would you summarize it?

   Analyst: Yeah. Um. Basically the process involves completing a file for each graduate student. That file contains letters of recommendation, test scores, statement of goals, transcripts, and the application itself. [*continues summary*] ... and then you send the file to . . . well, and you also tell them when it's going to appear before the committee. So, is that the same time each term or how does that . . .?

   Client: Within a few weeks, yeah.

   Analyst: [*This begins a whole new round of discussion, one focused on filling in details that are necessary to complete the summary started above.*] ...

There is at least anecdotal support in [41] for the enumeration technique shown in the first example as being a means of jogging a client's memory. The second example goes beyond a simple enumeration or paraphrase of a specification component, and begins to address control issues. In particular, while the summarization in this example was prompted by a member of our group observing the session, and hence may be viewed as artificial, its ability to drive problem refinement seems an important one. For instance, we might speculate about a tool that could do timely sweeps over the current design, summarize it, note missing pieces, and propose further refinements.

We close this section by noting the large gap between our goals for an interactive design assistant and our results to date on achieving them. Both our formal and informal observations of expert analysts have given us enough information to identify important components of a specification design model, and in some cases, enough information to build tools to test them. What we are lacking are 1) the details of the majority of these components, and 2) a clear understanding of how they interact and are effectively controlled.

6. Related work

Related projects, besides those listed in the preceding sections, are discussed below. Note that the work we list here is directed toward the knowledge-based *specification* problem; a discussion of research that addresses broader issues of knowledge-based software development can be found in [15, 33].

6.1 Related specification concerns

While relatively little work has been performed in the study of the specification *process*, this is not to say that other portions of the specification task have been neglected. For instance, a growing amount of research can be found in the definition of formal specification languages, for which a good set of references can be found in the proceedings of the International Workshop on Specification and Design, e.g., [19]. Given a formal specification language, we can analyze it, use it as a document to verify an implementation, and even in some cases, execute it [8, 44].

Related to execution, the rapid prototyping approach suggests that we build a working implementation that a client can directly interact with. The argument is that while such an implementation may only be partial, in some cases only incorporating the interface, it will prove useful in further validating the client's requirements. In general, it would appear that executable specification languages in tandem with a rapid prototyping approach provide a powerful means of validating a specification.

Are these two techniques -- formal specification languages, and validation through execution -- enough to effectively support the specification process? We argue no on two grounds. First, a specification language provides a representation for a *finished* product. While this representation may provide constructs that ease design, for instance by allowing levels of abstraction, freedoms of expression or forcing us to attend to certain specification concerns (initialization, error checks), the design process itself remains outside of its scope.

Second, execution of a specification or implementation seems clearly useful in validating certain aspects of a system, e.g., the interface, performance, certain functionality. However, pushed to the extreme, this might be characterized as hackerism versus analysis. Specifically, what test cases should be run on the prototype to validate intent? What constitutes an error? Just as we do not encourage a bang-on-the-keys approach to testing an implementation for development errors, it seems reasonable to eschew a similar approach when testing a specification against intent. Our observations tell us that a good analyst knows what key points to bring up with a client in a particular application domain, e.g., what is typically required, what are the special cases, how have similar development efforts failed. It is these points that are used to "test" a client's specification. In summary, a prototype without a principled testing plan would give little confidence that anything other than the vanilla functionality of the program would be explored.

Finally we note that our observations of systems analysts working with clients has shown us an interactive problem solving process as opposed to a transcription process. In particular, an analyst that is expert in an application domain would not only listen to the client's stated requirements, but would also add new ones, force the client to justify others, and argue for exlusion or modification of still others. In essence, a view of systems analysis founded on the old saw that the customer is always right would appear to be almost always wrong.

In summary, we argue that new languages and execution techniques will give us more powerful tools to work with, but neither provides any magic bullet for representing the knowledge needed to automate the specification process itself.

6.2 Design goals

In [31], Mostow and Voigt examine the implicit design decisions that go into building software. Their approach was to produce a rationalized design that would lead to an existing piece of code, given design recollections from the original implementors of the code, and a theory of interacting design goals and methods for combining them into a coherent design. We believe results from this work can benefit us in two ways: 1) it may provide some guidance on the order in which parallel developments and elaborations should be carried out, and 2) it may suggest various strategies for the merge operation.

6.3 Specification evaluation

It seems clear that some form of "operationalization" is necessary to find intention bugs in specifications. The rapid prototyping approach provides this by building a source code program from the specification, or using the specification itself if it is executable, and running it against intent. This latter approach often rests on what is commonly known as symbolic evaluation. Cohen [8] and Swartout [42] have built a symbolic evaluator and behavior explainer for Gist that serves as a good reference point. 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. Another system sifts through the trace to summarize the results. 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 [42]:

> "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 had some notion of what was interesting, it could avoid lengthy and unproductive paths. Our arguments in this paper 1) support Swartout's conjecture, and 2) extend it along several lines. 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 have proposed is to explicitly represent what is interesting about the domain, again part of the knowledge we see an expert 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 analyst is to recognize and show the user the ramifications of his or her actions, e.g., adding a new branch to a library, extending borrowing limits of certain library users, allowing staff to borrow without restriction, 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 or elaboration system. Given this, we see the following roles for operationalization in a specification system:

- The role of *validator*, as used by Swartout [42] and Cohen [8]. Here, *the user* selects test data and an action or actions to be confirmed; the system executes the action, and presents the results.
- The role of *validator*, as used in section 5.5.1. Here, *the system* selects test data and an action or actions to be confirmed by the user.
- 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 [39]), and is discussed in section 5.5.3.

For the first role, while the user can set up and run a specification test in the critic system discussed in section 5.5.1, we currently do not provide the analogue of Swartout's explainer; the user must interpret the results manually.

6.4 Specification as problem solving

The PHI-NIX project at Schlumberger-Doll is an attempt to apply domain dependent knowledge to the software specification and implementation problem [5]. PHI-NIX takes a batch compilation approach: high level problem descriptions are taken in, and the system first translates these into a formal specification language, and then maps this specification into compilable code. During the specification process, the system's major concern is with mapping mathematical, and in some sense, idealized descriptions onto physical equipment and known approximation techniques. The domain knowledge comes from a set of rules that describe various definitions, facts, and properties in the oil drilling world. Because the focus is more on translation than construction, there is no notion of interacting with the user to further refine his or her problem. The relation to our work then is the use of domain knowledge to fill in the details of a specification. PHI-NIX relies on rule-based heuristics to guide it to a reasonable formal representation (and later an implementation) of a informally stated problem. Using the Oz/Glitter system, we are exploring the translation of high level specification design goals into a set of primitive specification construction operations. Like PHI-NIX, we too have found the need for domain dependent problem solving knowledge in automating specification construction.

6.5 Reuse of analysis

The Draco system [1, 32] shares our goal of capturing domain knowledge so it can 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.

6.6 Reuse of plans

Several projects have studied the use of pre-stored ''specification plans'' that can be used to construct a specification by component composition [28, 34]. Our only effort along these lines is Nagarajan's catalog of problem cases for critiquing a specification. As discussed in section 5.5.3, we expect that these may be used as a means of synthesis as well as analysis, but have yet to explore this approach.

6.7 Design in general

There is a growing body of work on formal models of design, both within Computer Science and without. A general survey is beyond the scope of this paper. However, we do note that Mostow's survey [30] of AI design paradigms has proven particularly useful to our work. We also note that the protocol studies conducted by Dietterich of Mechanical Engineering designers [10] have given us further insight into the problem acquisition process.

6.8 Knowledge acquisition

Recent work in the Expert System field has concentrated on the problem of extracting the expertise of a human expert so that it can be incorporated into a computer program (e.g., a rule-based shell). Portions of this research that focus on interviewing techniques (see for instance [7]) appear to have some relevance to the work discussed in section 5.5. In particular, while appearing disparate on the surface, it would seem that problem specification and knowledge acquisition have an interesting set of common goals.

7. Summary

We have presented three models of specification design: Goldman's characterization of gradual elaboration; Feather's parallel development model; Fickas' knowledge-based critic. The central theme of the paper is that the set of research results obtained from 1) these three models, and 2) attempts to formalize and automate them, point the way towards a useful computer-based specification assistant. We have argued that the bases for such an assistant must rest on the following views:

- The specification process is evolutionary.
- The specification process is one of design.
- The specification process often involves conflicting goals.
- The specification process involves collaboration between client and analyst.
- The specification process is inherently knowledge-based.

As we discussed in section 5, we are studying these issues along two broad fronts: 1) exploration of the interactive design process that leads to a useful specification, and 2) identification and representation of the type of knowledge underlying such a process. The tasks that we believe important for further research in the area can be outlined as follows:

Task 1. Produce a model of *interactive* problem acquisition and specification construction, one based on a view of cooperative problem solving versus syntactic transcription. We have discussed some of the components we believe will be necessary in such a model; much work remains.

Task 2. Characterize the complex ways conflicting goals come into play during specification, and means of recognizing them, documenting them, and resolving them. We have argued that Feather's model provides a basis for studying these problems; the work by Mostow&Voigt [31] clearly fits here as well.

Task 3. Come to grips with ''requirements''. There seems to be a growing belief that the separation of 1) requirements and 2) a formal specification is not a useful one, at least partially on the lack of a precise definition of either. However, it is clear to us that what are calling policies are capturing some useful information, information that is typically extraspecificational. By what ever name they go by, they are turning out to play a key role in every facet of our study of specification design. In particular, our current belief is that policies/goals/requirements have much more to do with the construction process leading to a product than with the product itself. Thus a view that we translate requirements into a specification seems wrong headed: we use ''requirements'' to guide and constrain the design process, a process that must wade through a potentially infinite number of specifications.

Task 4. Extend the top-down design model currently in practice. Specifically, we have argued that exploration of alternatives, patching and reuse are all common design techniques employed by humans when building a complex specification. At the least, our tools should manage these processes for the human designer.

Unlike the rather optimistic estimates in [21], we believe that we are in for a much longer effort to achieve a ''knowledge based requirements assistant'', at least for one that meets the goals above. We hoped to have shown, however, that the problems we face are becoming clearer, and that even partial solutions can lead to useful tools along the way.

Acknowledgments

We would like to thank Sarah Douglas and Barbara Korando for their assistance in carrying out our protocol experiments. We also acknowledge the support of the National Science Foundation under grant DCR-8603893.

References

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

[2] Balzer, R., Automated Enhancement of Knowledge Representations, In *Proceedings of 9th International Joint Conference on AI*, 1985

[3] Balzer, R., A 15 year perspective on automatic programming, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

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

[5] Barstow, D., Domain-specific automatic programming, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

[6] Bearman, M., Fickas, S., Parallel development of software specifications: An Example and Critique, Technical Report 87-05, Computer Science Department, University of Oregon, Eugene, OR. 97403, 1987

[7] Boose, J., Personal construct theory and the transfer of expertise, In *Proceedings of AAAI-84*, Austin, 1984

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

[9] Corbin, J., Developing Computer-Based Library Systems, ORYX Press, 1981

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

[11] Dietterich, T., Ullman, D., Stauffer, L., Preliminary Results of an Experimental Study of the Mechanical Design Process, Technical Report 86-30-9, Computer Science Department, Oregon State University

[12] Feather, M., Language support for the specification and development of composite systems, *ACM Transactions on Programming Languages and Systems*, Volume 9, Number 2, April 1987

[13] Feather, M., Constructing specifications by combining parallel elaborations, To appear in *IEEE Transactions on Software Engineering*, Available from author at USC/ISI, 4676 Admiralty Way, Marina del Rey, Ca. 90292

[14] Fickas, S., Automating the transformational development of software, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11 Nov. 1985

[15] Fickas, S., A Knowledge-Based Approach to Specification Acquisition and Construction, CIS-TR 85-13, Computer Science Department, University of Oregon, Eugene, Or., 97403

[16] Fickas, S., Automating Analysis: An Example, In *Fourth International Workshop on Software Specification and Design*, Monterey, 1987, Available from author at Computer Science Dept., University of Oregon, Eugene, OR. 97403

[17] Fickas, S., Supporting the Programmer of a Rule Based Language, *International Journal of Expert Systems*, Vol. 4, No. 2, May 1987

[18] Fickas, S., Collins, S., Olivier, S., Problem Acquisition in Software Analysis: A Preliminary Study, Technical Report 87-04, Computer Science Department, University of Oregon, Eugene, OR. 97403

[19] Fourth International Workshop on Software Specification and Design, IEEE Computer Society, Order Number 769, Monterey, 1987

[20] Goldman, N., Three Dimensions of Design, In *Proceedings of AAAI-82*

[21] 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

[22] Greenspan, S., Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, Ph.D. Thesis, Computer Science Dept., Toronto, 1984

[23] Horwitz, S., Prins, J., Reps, T., Integrating non-interfering versions of programs, Technical Report #960, Computer Sciences Dept., University of Wisconsin, 1987

[24] Kaczmarek, T., Bates, R., Robins, G., Recent Developments in NIKL, In *Proceedings of AAAI-86*, Philadelphia, 1986

[25] KEE Reference Manual, Intellicorp, Palo Alto, Ca.

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

[27] London, P., Feather, M., Implementing specification freedoms, *Science of Computer Programming*, Number 2, 1982

[28] Lubars, M., Schematic Techniques for High-Level Support of Software Specification and Design, In *Fourth International Workshop on Software Specification and Design*, Monterey, 1987

[29] Meyer, B., On formalism in specifications, *Software* 2, January 1986

[30] Mostow, J. Towards better models of the design process, *AI Magazine*, Spring 1985

[31] Mostow, J., Voigt, K., Explicit incorporation and integration of multiple design goals in a transformational derivation of the MYCIN therapy algorithm, *IJCAI87*, 1987

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

[33] Readings in Artificial Intelligence and Software Engineering, Rich, C., Waters, R., (eds), Morgan and Kaufman publishers, 1986

[34] Rich, C., Waters, R., Rubenstein, H., Toward a Requirements Apprentice, In *Fourth International Workshop on Software Specification and Design*, Monterey, 1987

[35] Rissland, E., Hypotheticals as Heuristic Device, In *Proceedings of AAAI-86*

[36] Rissland, E., The Ubiquitous Dialectic, In *Proceedings of ECAI-84*

[37] Rissland, E., Constrained Example Generation, Technical Report, COINS, University of Massachusetts

[38] Robinson, W., Towards the formalization of specification design, Masters Thesis, Computer Science Dept., University of Oregon, 1987

[39] 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

[40] Stevens, W., Myers, G., Constantine, L., Structured Design, *IBM Systems Journal* 13, (2), 1974

[41] Swartout, W. The Gist English generator, In *Proceedings of the National Conference on AI*, 1982

[42] Swartout, W. The Gist behavior explainer, In *Proceedings of the National Conference on AI*, 1983

[43] Swartout, W., Balzer, R., On the inevitable intertwining of specification and implementation, *Communications of the ACM*, 25(7) (1982)

[44] Terwilliger, R., Campbell, R., PLEASE: A language for incremental software development, *Fourth International Workshop on Software Specification and Design*, Monterey, 1987

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

[46] Wilbur-Ham, M., Numerical Petri Nets - A Guide, Report 7791, Telecom Research Laboratories, 1985, 770 Blackburn Road, Clayton, Victoria, Australia 3168

[47] Wile, D., Program development: formal explanations of implementations, In *Commun. of ACM*, 1983

[48] Wilensky, R., Meta-planning, In *Proceedings AAAI-80*, Stanford, 1980

Appendix A

Consider a small library database with the following transactions:

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

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

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