Formalizing and Automating Requirements Engineering

Stephen Fickas
John Anderson
William Robinson

CIS-TR-90-03
April 6, 1990

Abstract

This paper draws together the results of three separate projects (all falling under what we call the "Kate" project) in which, over the past seven years, we have been studying requirements and specifications for complex systems. We present a set of requirements issues, then motivate them by examining three Kate projects: the critic, the planner, and the resolver.

Our results help to clarify our understanding of the issues surrounding requirements engineering, and support the idea that semi-automated tools can be built to address those issues. However, the domains in which we have been able to work effectively are relatively small. Several important specification issues remain to be addressed.

Department of Computer and Information Science
University of Oregon

Formalizing and Automating Requirements Engineering

Stephen Fickas
John Anderson
William Robinson

Computer Science Department
University of Oregon
Eugene, OR. 97403

1. Introduction

Over the past seven years, a group of faculty and students at Oregon have been involved in the study of requirements and specifications for complex systems[1]. This work, all falling under what we call the Kate project, has produced various models and tools. Until now, we have not drawn together the results of these separate projects. In this paper, we will take on that task - we will attempt to integrate the results of three individual Kate projects to gain a general insight into the requirements and specification processes.

The Kate project has often been described as one at the intersection of AI and SE. It is true that all of our work to date has borrowed heavily from the AI field, as will be seen in the following sections. The reason for this is worth noting. We did not set out from the AI field looking for problems to solve in the SE field. Instead, we started with the problem of constructing requirements and specifications, and were *inevitably* drawn to representation and problem solving techniques in the AI field. There is an explicit claim here: when attempting to formalize and automate high-end SE processes (requirements elicitation, specification de-

[1] We will use the term *complex system* repeatedly in this paper. Our definition of a complex system is one that involves physical sub-systems, social sub-systems, and software sub-systems. In essence, we consider problems where a software system is embedded in an environment that includes physical and social components. Given this definition, specification complexity does not spring from the size of the resulting implementation, but from the difficulty of modeling the environment. We note that we have found no reliable correlation between the two, i.e., complex specifications (by our definition) do not necessarily lead to complex implementations, nor do simple specifications necessarily lead to simple implementations.

sign, specification validation), one must confront problems studied by AI. If one has a formal specification in hand, then it might be argued that non-AI techniques suffice for constructing and verifying an implementation. But producing a ready-to-implement specification is *inherently* an ill-defined task, involving multiple (often conflicting) points of view, only a fuzzy notion of success and failure, and problematic issues of completeness.

This leads us to a second claim. We conclude from our results that current specification languages are both too meagre and too low a level to represent a client's needs and preferences. Our claim is that large efforts poured into tuning and incrementally improving languages at this level will pay only modest dividends. The large payoffs will come from languages that allow us to formally represent what lies above current specification languages. Whether these new languages are extensions of existing specification languages (see, for instance, the wide spectrum philosophy of Gist [London&Feather 1982]), or separate languages that are translated or transformed into existing specification languages (the Draco system is an example of this narrow spectrum approach [Neighbors 1984]) is irrelevant at this point - we have yet even to define the pre-specification concepts which must be attended to.

This brings us to our last point: the requirements versus specification debate. This is a special issue on requirements engineering, so the reader might suspect that there is a process (and associated field of study) of requirements that is separate from specification. Unfortunately, we have found that it is difficult to know, in any useful sense, where *requirements engineering* leaves off and *specification engineering* takes over[2]. This difficulty is not surprising to us, for we have the same problem differentiating specification from implementation. A specification *is* an implementation from a high enough view, i.e., specification component = implementation of requirement/need. Attempting to keep these concepts separate becomes particularly bothersome when dealing with wide-spectrum languages that often thoroughly blur the distinction between concepts like specification and implementation. Given this, the most accurate description of our work would be that it addresses issues in formalizing and representing pre-specification languages and processes. However, we will allow conve-

[2]There does exist an IEEE standard for a requirements document. However, it has proved of little use in our attempts to formalize and automate the requirements engineering process.

nience and conventionality to win over accuracy - *pre-specification language* will become *requirements language*, and *pre-specification processes* will become *requirements engineering* in the following discussion. Figure 1 represents this graphically[3].

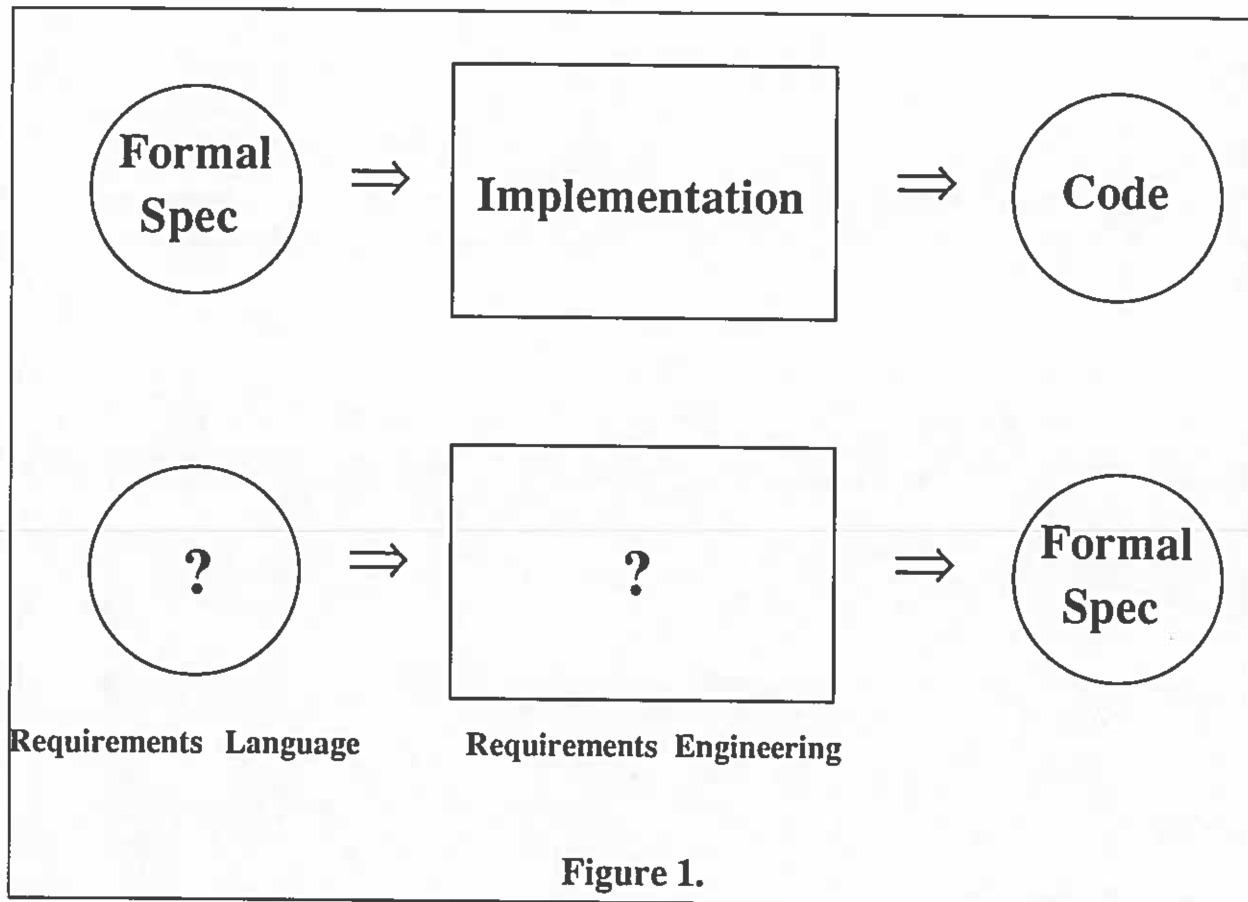

Figure 1.

In the remainder of the paper, we will first present a set of requirements issues, and then motivate them by looking at three specific Kate projects: the critic, the planner, and the resolver.

2. Issues in Requirements Engineering

The importance of identifying the issues of system *specification* as a prelude to designing new specification languages is taken up in [Balzer et al 1979]. We believe the same reflection on *requirements* issues is needed in the design of new requirements languages. In support of this belief, we present a set of requirements issues. These conclusions are reached,

[3]Note that by giving the top box in figure 1 a label of *implementation* (as opposed to a question mark) we are in no way implying that this is a well formalized and automated process. As discussed in [Fickas 85], its form is still an open research question for many formal specification languages.

much like those by Balzer et al, both from observing humans doing requirements engineering and from attempting to build tools, such as the critic, planner and resolver, that formalize and automate some portion of the requirements engineering process.

Issue 1: The requirements language used by clients[4] in complex domains is often ambiguous.

One can take the view that this ambiguity is bad: that we must spend more effort in tightening down domain definitions and needs. We take the opposite view: clients should be allowed to state general, often ambiguous needs, and then proceed to negotiate their meaning and utility. What we do believe is bad under current practice is that none of this disambiguation process is supported or captured in any usable way.

Issue 2: A single agent cannot be relied on to state non-conflicting requirements.

It is naive to believe that an agent will refrain from stating goals that cannot simultaneously be achieved, even if so directed. Quite the opposite, we believe one should encourage an agent to state each goal independently of its interaction with other goals. In particular, we do not believe an agent can foresee the outcome of the complex negotiation process that is carried out when goals conflict. For example, an agent may avoid stating a goal G1 because the agent believes it would conflict with another goal G2. We want both G1 and G2 stated, even if in conflict. The agent may want to state *a preference* for G2 over G1. In any case, a clever negotiator may find a way to achieve both G1 and G2, *in effect.*

Issue 3: Agents are not homogenous.

Multiple special interest groups or agents typically are involved with the construction of complex systems. At least at some preliminary step, their separate concerns should be represented.

Issue 4: Two separate agents cannot be relied on to state non-conflicting requirements.

This issue is corollary to the second issue. Again, we do not believe it is useful for one to attempt to limit conflict between separate agents during the requirements process.

Issue 5: Agents have preferences and bounds they are willing to live (and bend) within.

The representation of goals as all-or-nothing is too restrictive. This rigid posture leaves no room for negotiation. A requirements language should capture the bounds of what is acceptable to an agent. In some cases this may be nothing less than 100% satisfaction, but in other cases there may be a range of acceptable achievement. Further, an agent may wish to state preferences among goals, e.g., prefer a small staff over security in a library.

---

[4]We will use the term *client* and *agent* interchangeably. An agent will refer to one or more outside entities with shared concerns. When we use the term agents, we will be referring to two or more groups with differing concerns. Finally, we note that an agent of a system is not limited to human group. For instance, the agents of a patient monitoring system might include ones concerned with safety, reliability, or even cost, in contrast to the patients, doctors and hospital administrators [Feather 1989].

Issue 6: Requirements engineering involves a "dialog" between a) the goals and constraints of the client and b) the methods that can be provided by existing technology.

The goals determine what methods will be considered; the methods that are available determine how easily the goals can be met. The dialog between client and analyst corresponds roughly to the dialog between goals and methods. First the client states his goals at a high level and the analyst points out potential conflicts and difficulties, based on his knowledge of existing methods. The client then clarifies or revises his goals and the analyst points out additional problems. Hopefully this process will eventually lead to a consistent set of goals and constraints that can indeed be realized by a system.

Issue 7: Conflicts lead to negotiation and resolution.

As corollary to issue 6, any complex artifact is the result of negotiation among competing concerns. If the negotiation process is *not* captured as part of the documentation of the artifact, then the intertwining process so eloquently argued for in [Swartout&Balzer 1982] is problematic.

Issue 8: We need to model the system environment.

Frequently client goals cannot be achieved by the system alone, but require cooperation of external agents. Or worse, external agents may have the ability to thwart the achievement of the goals of the system. The ability to model the environment, including the actions of external agents, is a difficult but necessary part of validating a system. In some ways this issue is simply further elaboration of footnote 1.

Issue 9: The notion of completeness of a domain model is strictly relative.

There is no notion of an *absolutely* complete domain model. For instance, for some problems in a domain we must worry in great detail about the behavior of the potential users of the resulting system. In other problems *in the same domain*, such behavior can be abstracted to a much less detailed level. Given the relative nature of completeness, one needs the ability to reason at various levels of abstraction.

As a corollary to issue 6, it may be possible to reason at an abstract level to sustain a dialog. For instance, we might propose an abstract method for achieving a goal well before we actually work out the details of its implementation. In essence, we are using abstraction to temporarily overcome a hole in our domain model.

Issue 10: Cases or scenarios seem to be an important tool in understanding how, and validating that, a set of requirements has been met.

While it is nice to reason in general about a set of high level goals, and even their links to actual implementation components, we find that clients do not gain confidence in a system until they see a pertinent set of system behavior.

Further, it may be possible to "model" a domain effectively through anecdotal cases. In particular, when a domain is too complex to represent formally, this may be our only means of reasoning.

In the remainder of this paper we will discuss the contribution of three Kate projects -- the critic, the planner, and the resolver -- to both the formation of these issues, and the amount of formalization and automation each tool provides. Viewed another way, the three projects were chosen to highlight a particular aspect of the missing pieces in figure 1.

To allow us to concentrate on the results of the three systems, discussions of the implementation aspects will tend to be high-level. We will refer the interested reader to the literature for further details.

3. The Critic

The first system, called the *critic*, was built around a set of recorded observations of analysts and clients discussing a software system to be built. The critic introduces the notion of a requirements language being synonymous with a goal language. It allows conflicting goals to be stated, thus representing the often conflicting requirements of a client. Goals are linked to specification components through a set of goal-achievement techniques (A.K.A. cases) found useful in the domain. Thus, the system employs a form of case-based reasoning (CBR) as a basis for analysis.

In essence, the critic addresses the left portion of figure 2, where a client proposes a system specification and an analyst critiques it. As discussed in [Fickas&Nagarajan 1988a], the right-hand process in figure 2, that of the analyst proposing and the client critiquing, is also prevalent in our observations of specification construction. However, we did not attempt to model it in the critic..

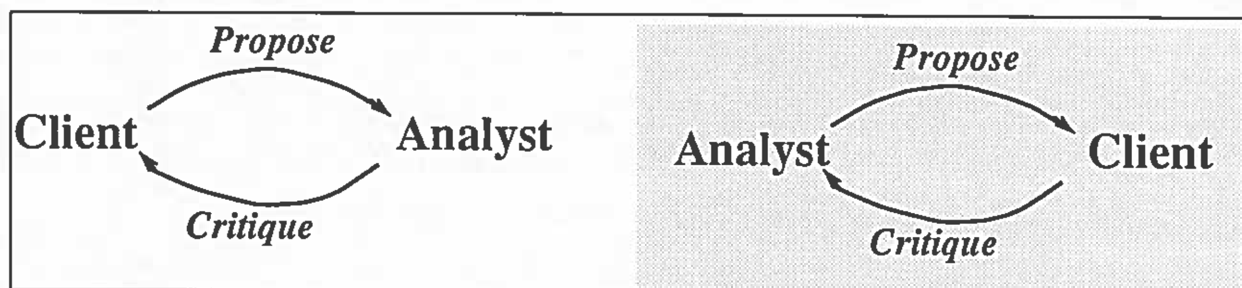

Figure 2.

One might employ two general approaches to the specification validation task: 1) use domain-*independent* heuristics to find likely holes in a specification, or 2) use domain-*dependent* information (A.K.A. knowledge, as in knowledge-based approach) to do the same[5]. Work by Swartout [Swartout 1983] and Yue [Yue 1988] is representative of the first approach; the critic is representative of the second.

[5] Although we list the two approaches as alternatives, clearly a combination of the two is also possible and likely desirable.

The first approach, that of using domain-independent heuristics, is both the more general and the more tractable of the two. However, its use is limited to the analysis of low level processes, e.g., if there is a process for creating an item, shouldn't there be a process for destroying it. While there is no doubt that this style of critique can be useful, we saw a completely different style of criticism when observing expert human analysts. In particular, our findings (as reported in [Fickas&Nagarajan 1988b]) were as follows:

- Analysts expert in the domain[6] referred often to both the successes and failures of similar past specifications when analyzing the current specification.
- Analysts expert in the domain were aware of *domain issues* that needed to be addressed by the client(s). The client was asked to place a relative weight or importance on each such issue.
- Analysts expert in the domain were aware of the different agents or user groups of a system. They were careful to take into account differing views and needs of the system among agents.
- Finally, analysts expert in the domain were aware of possible compromises among agents with conflicting needs, typically taking into account the issues (and weights) associated with the conflict.

We used these results to build a model of specification critiquing. We implemented the model as a computer-based analysis tool, and tested the tool in an informal comparison with an analyst expert in the domain. We discuss our tool in detail in [Fickas&Nagarajan 1988a], and will simply summarize the details here.

The critic provides a client with three things:

1. A formal specification language based on an extended form of Petri-net representation.
2. A goal language to state requirements.
3. A catalog of interesting past cases in a particular domain.

The input to the critic, as supplied by the client, is 1) a formal specification, and 2) a utility value for each goal in the goal language. Goal utilities can take on the value important, unimportant, or unknown.

The output of the critic is a critique that takes one of three forms (typically represented as an animated scenario):

1. The description of a potential behavior allowed by the specification where an *important* goal is supported.
2. The description of a potential behavior allowed by the specification where an *important* goal is obstructed.
3. The description of a potential behavior allowed by the specification where an *unimportant* goal is supported.

---

[6] We use the rather cumbersome term *analysts expert in the domain* to differentiate analysts not expert in the domain, i.e., generalists. As discussed in [Fickas et al 1987], our study included both with differing results.

The next section discusses the our evaluation of the critic as it applies to our more general set of requirements engineering issues.

3.1 Discussion

We have run the critic tool on the problem description in figure 3. We supplied the tool with

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:

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

Figure 3.

1) a specification of the problem in our language, 2) a case-base of interesting resource management cases, and 3) our best reconstruction of the implicit goals after looking at the origins of the library example and talking to the authors of the version used in [IWSSD 1987]. In particular, we set the goals to reflect a small academic library, possibly a department library run by a secretary. While we feel confident in this interpretation after talking with the authors of the text, informal descriptions such as this are clearly a problem for any translator, human or machine, in terms of ambiguity and missing/implicit goal information. For instance, we can read at least four possible meanings for *small* in line 1:

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

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

Kemmerer's original description [Kemmerer, 1985], from which this description is at least partially taken, gives no clue; 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 opted for a combination of meanings 1, 2, and 3.

There is an important point to be made here: it is not the use of ambiguous words like *small* that leads us to trouble. In fact, we end up championing their use during requirements as a freedom from specificity. What is of concern is that the negotiation process that we went through to disambiguate such words is left implicit in systems built today. Pointing back to our example, we believe that the various interpretations of small and the final compromise outcome we arrived at must be part of the requirements process *and documented.*

In [Fickas&Nagarajan 1988b], we give a side-by-side comparison of the tool and the human critique of the library problem. As shown by the broken lines in figure 4, this comparison was

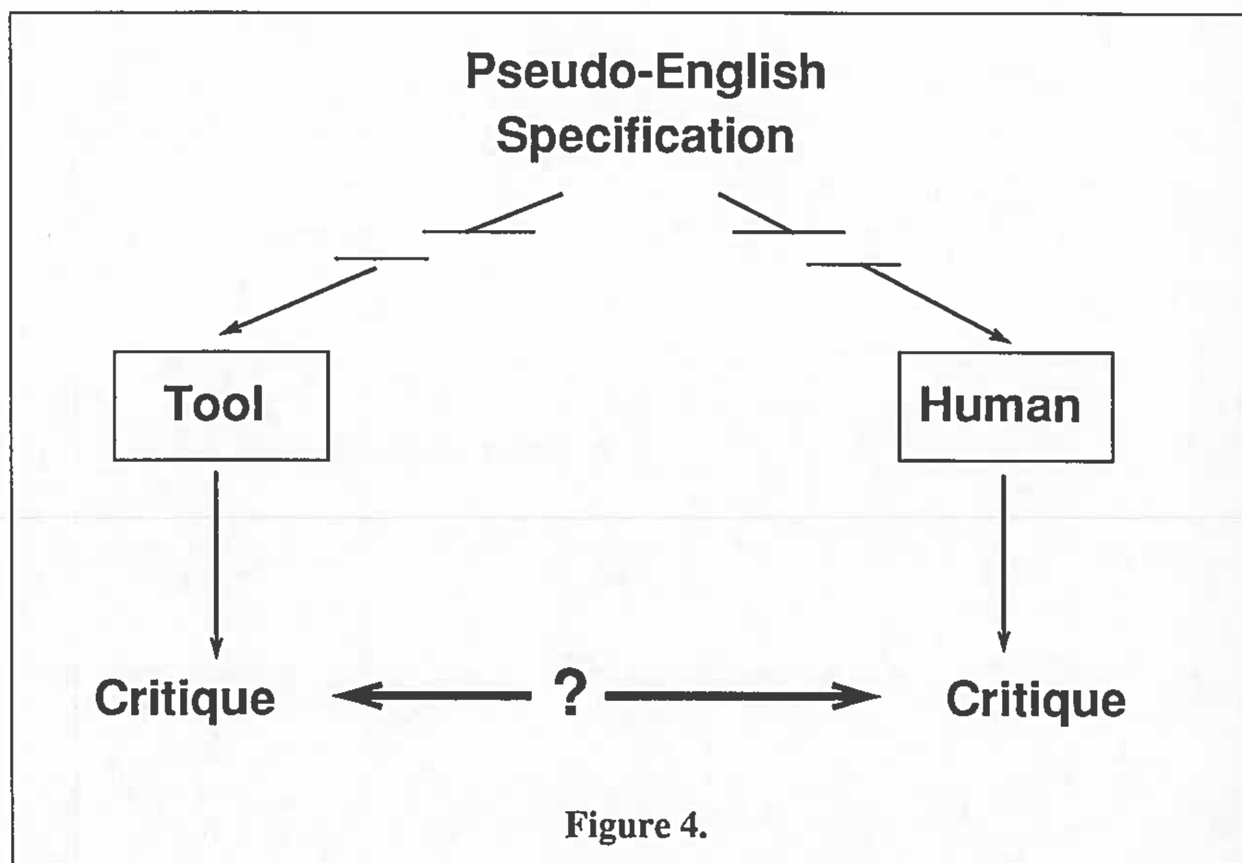

Figure 4.

informal; we had to supply several of the steps ourselves. While we do not have space to reproduce the full discussion here, we will present three results relevant to this paper:

1. We found general support for our representation of domain goals. In particular, the analyst spent the first part of the session establishing the needs of the library being described. These needs registered, in a straightforward fashion, with the domain goals we had represented for resource management systems.

2. The analyst used the importance given to the goals of the library explicitly in some places to relativize her critique, e.g., noting that the nature of a small department library required more than a binary view of the state of resources (see item 6 in figure 3). Of course it is difficult to judge what criticism she *did not* state that was similarly influenced by the goals in force.

3. The analyst relied heavily on both abstract and concrete cases to make a point. The cases represented by the critic's case-base can be viewed as a small subset of those presented by the analyst. A focus of both analyst and critic cases was the normal and deviant behavior one can expect from external agents in the environment.

4. A major discrepancy between analyst and tool was in the analyst's ability to deftly juggle competing concerns. In particular, she was able to weigh the importance of various goals, and order them when considering a particular component in the description, e.g., giving borrowers a useful working set must supersede concerns with keeping adequate stock on the shelves. This ordering sometimes leads to suggestions for compromise, e.g., adding a recall-on-demand component to the specification.

In constructing a specification critic from these findings, we eventually needed to represent three types of information not currently representable by existing specification languages; each is discussed in turn below (we also list the issues that they address from section 2).

Needs as goals. Goals such as *allow users to gain access to a useful working set and to keep until done,* and *allow users to have a large selection to choose from* have two important properties: 1) they are ambiguous - their achievement criteria will eventually have to be negotiated, and 2) they are at the right level - they are the kinds of things clients state as needs in the preliminary stages of requirements engineering. **(I1)**

Past cases. There are various means one might employ in disambiguating requirements goals and eventually mapping them to specification components. For instance, one might try a problem solving approach that reduces goals to sub-goals, and finally links primitive goals to specification components [Fickas 1985], [Mostow 1983]. One might employ a planning approach that finds operators for achieving a goal (see section 4). Our observations of human analysts expert in the domain lead us to a third approach, definition-by-example. Instead of falling back on a first-order theory of the domain, analysts seemed prone to reason about a user's needs by dredging up past anecdotal information. We call this a case-based approach, for it does not draw upon problem solving or planning, but instead retrieval of cases that look relevant to achieving a particular goal.

We argue that, in domains lacking a first-order model (i.e., domains in which the environment is too complex to formally represent), past experience can at least partially compensate. The library domain is a good example of this, given the need to represent the complex sociology of its patrons. Lacking a formal representation of the environment of a library (e.g., the "university" of a university library, the "city" of a city library), we fall back on what have emerged as important, recurring cases to consider. **(I8, I10)**.

Conflict and compromise. Current specification languages do not tolerate conflict. One cannot (soundly) state that both x and ~x are desired, or even further, that one would prefer x but will accept ~x. But in our observations, such a statement is exactly what users can be relied on to request. Typical conflicts for us were among Working-Set versus Stock-on-Hand, Privacy versus Necessary-Information, Security versus Budget, and Security versus Ease-of-Access.

The critic allows representation of goal conflict as it manifests itself in a specification. For instance, if a client has specified a policy of unrestricted borrowing in a library, and at the same time marked goals of providing-a-useful-working-set and having-a-good-stock-on-hand as both important to achieve, the critic will show the policy simultaneously supporting one

while obstructing the other. Unfortunately, the critic provides no way to state goal preferences or relative utility, e.g., that a useful working set is more important than having stock on hand. We will take up this issue as we discuss the resolver in section 5.

What the critic does do is to begin the goal clarification process, part of the more general dialog process we argue is necessary to support in requirements engineering. In particular, it presents the client with various aspects to consider in deciding just how far he or she is willing to go to achieve a goal. **(I1, I2)**.

4. The Planner

The second system, called the *planner*, uses STRIPS style operators to represent and reason about requirements. The underlying notion is that a functional requirement can be stated in terms of preconditions and effects. For example, the "check-out" operator shown in Figure 5 can be used to represent the requirement that a library include a "check-out" function. Other high level functions associated with a library include returning books, locating documents given a partial description (e.g., author), and recording overdue books and fines. Each function can be defined by its preconditions and effects, that is, as a STRIPS operator.

check-out (Patron, Book)

Preconditions: available(Book), registered(Patron)

Add: has(Patron, Book), record(Patron,Book)

Delete: available(Book)

Figure 5.

A second motivation for the planning view is the intuition that mapping goals onto methods for achieving them is a fundamental part of requirements engineering. From a broad perspective, any process that maps goals to methods can be considered a type of means-ends analysis. Issue 6, concerning the "dialog" between client goals and methods for achieving them, is directly addressed in the planner.

A third source of inspiration for using planning in requirements engineering is the observation mentioned in the previous section that analysts rely heavily on cases. Operators can be used to represent individual actions in a specific plan (i.e., a case). For example, an instance of the check-out operator in Figure 5 could appear in a plan which achieves patron P's goal of obtaining book B. Thus in the planner, operators are used to state requirements, map from goals to requirements using means-ends analysis, and reason about individual cases represented as plans.

To relate the work on the planner to that already presented on the critic, we provide the following mappings:

critic's goal language $\Longrightarrow$ a STRIPS style state description language

cases $\Longrightarrow$ STRIPS style operators

matching goals to cases and cases to specifications $\Longrightarrow$ means-ends planning

The goals of the client become problem descriptions - states to be achieved by a set of operators or functions provided by the system. The planner represents constraints on a system by allowing a client to state avoidance goals, i.e., undesirable states. Neither goals nor constraints are rigid in the planner: both are reassessed when conflicts arise, e.g., the conflict of an operator that allows both a goal to be achieved *and* a constraint to be violated.

As we will see, we have overcome the difficult problem of matching cases to specification in the critic by replacing it with a more tractable planning technique. Unfortunately, we have introduced a new and difficult problem as well: that of representing complex domains in a STRIPS style representation.

4.1 The model and the tool

Figure 6 presents an overview of the planner. The planner plays three distinct roles in requirements engineering: generating, critiquing, and modifying requirements. The ovals represent the inputs and outputs for each of these stages. The domain analyst provides two sets of operators: one, the set of *environment operators*, represents the actions that are available with or without the artifact being specified. For example, actions like "pick-up-object" and "walk" are environment operators. The second set of operators, the *candidate operators*, serves as a catalog of possible functions that could be provided by a particular type of artifact. For example, "check-out" and "recall" are candidate operators for a library.

The client provides a set of goals to be achieved and a set of constraints that are to be met by the target artifact. Here we see the two sides of the dialog between goals and methods: the domain analyst supplies knowledge of the methods that are available for achieving goals in a domain, the client provides a particular set of goals to be achieved and additional con-

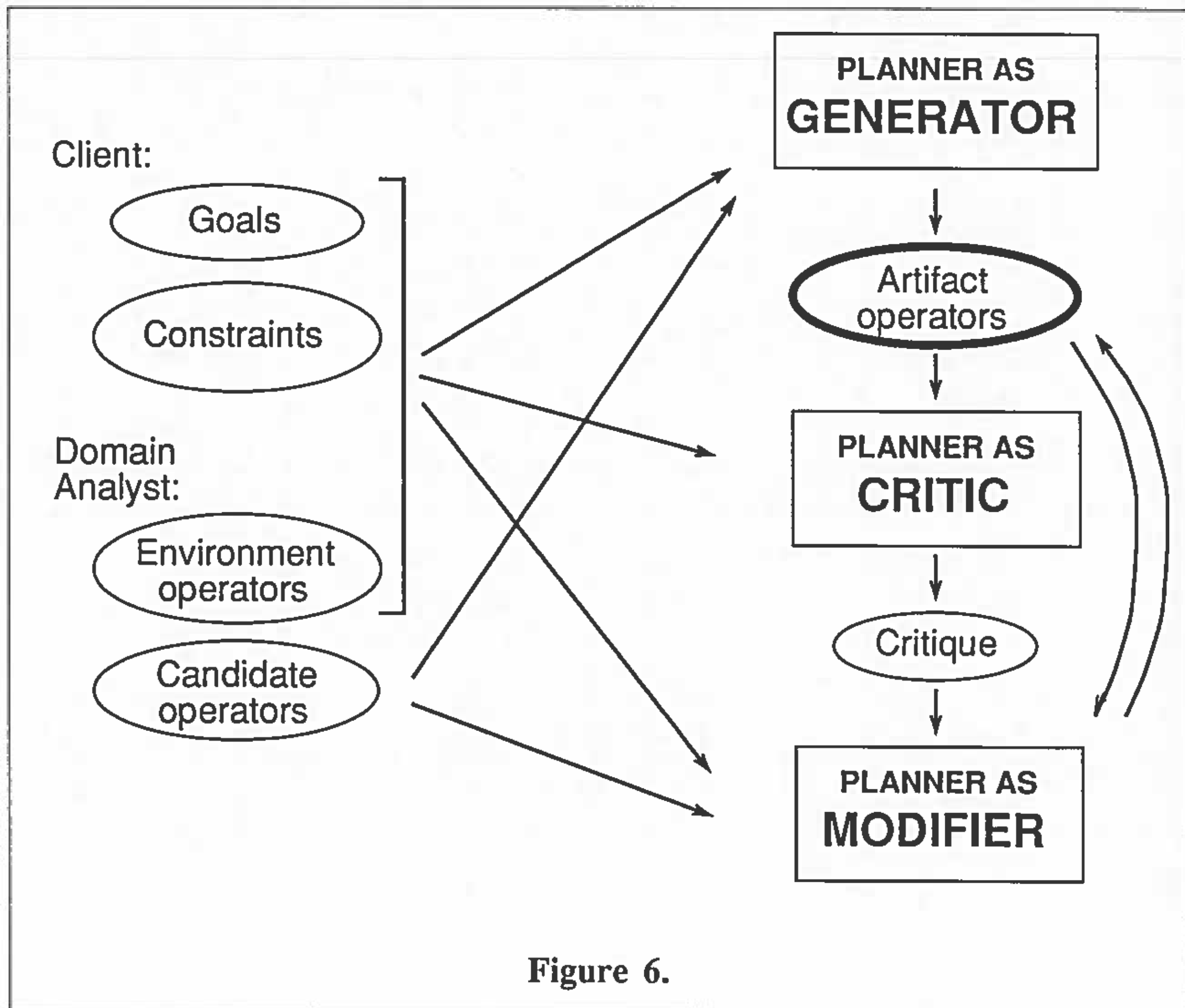

Figure 6.

straints to be met. Requirements engineering might be viewed as a process of finding a match between the two. Thus, many conflicts arise not from the goals and constraints themselves, but from the necessity of using existing technology to achieve those goals.

The input to the generator are the goals and constraints of the user and the environment and candidate operator sets. The goals are stated as problems such as "patron P wants book B given situation S". The constraints are represented as states to be avoided, e.g., "patron P has book B but book B is not checked out to P."

The output is a subset of the set of candidate operators. This set of operators, called the *artifact operators*, represents the set of requirements for the artifact. This set is found by solving the goal problems and collecting the set of operators used in the solutions. This set represents the functions that must be provided by the artifact in order to allow users to achieve their goals.

The input to the critic is similar to the input to the generator. The only difference is that instead of the entire set of candidate operators, only the artifact operators are used. The goals, constraints and environment operators are input as before.

The output is a critique of the artifact set, listing a) the input problems which cannot be solved, b) the input avoidance goals which can be achieved, along with the plans that achieve them, and c) any superfluous operators which are not needed to solve the input problems.

Let's examine the ideal case first: the planner reports that all goals are achieved and no constraints are violated. In this case, the operators contained in the artifact set are ready to be mapped to a specification language S. Given a suitable state-transition form of S (e.g., Gist [London&Feather 1982]) to match our STRIPS style representation, this mapping should be routine. The result is what can be viewed as a functional specification.

In the non-ideal (and unfortunately typical) case, either one or more goals cannot be satisfied, one or more constraints are violated, or both. Figure 7 lists examples of each of these cases plus the case where irrelevant or redundant operators appear to have been included.

1. Goal cannot be achieved

Book is already checked out ⇒ ? ⇒ Check out Book ⇒ Patron has book and book checked out

2. Constraint is violated

Enter Library ⇒ Pick up Book ⇒ Leave Library ⇒ Patron has book but book not checked out

3. Redundant operators are included

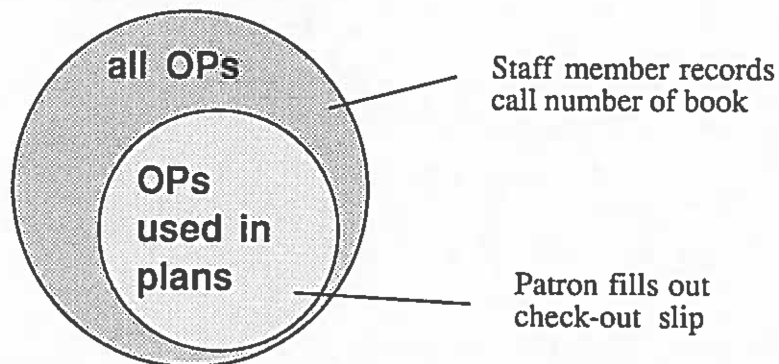

Figure 7.

The input to the modifier includes the goals, constraints and environment operators; the entire set of candidate operators; the artifact set, and the critique provided by the critic. The output is either a new artifact set which solves the problems found by the critic or a list of problems that could not be solved. If a problem cannot be resolved by altering the artifact set, it means that the goals and/or constraints will have to be modified.

The details of each stage of the process are beyond the scope of this paper; the interested reader is pointed to [Anderson&Fickas 1989]. To avoid overstating what has been accomplished, we note that the problems that the planner has been able to solve so far have been very small and carefully designed. In addition, the output of each stage must be manually translated into input for the next stage. Finally, the modifier is extremely limited in power.

4.2 Current work

The notion of partial commitment is fundamental to our current work on the planner [Anderson and Farley 90]. The basic notion is that a general operator can be used to represent a large class of related operators, in particular, all operators that achieve the same results. The general operator is an abstraction in that it only contains the preconditions and effects that are common to all of the specific operators. For example, suppose all check-out operators require that the book being checked out be in the library. The need to recall a book can be seen as an issue without having committed to one of the specific check-out operators.

We are exploring the possibility of combining the notion of an operator hierarchy with ideas from derivational analogy. The hierarchy would include both taxonomic (class/sub-class) and partonomic (part/whole) links. Such a hierarchy would allow us to store complete solutions, partial solutions, and/or generalized solutions. Such a hierarchy provides a means for organizing operators with the same effects, along with rules for selecting among them. This is useful in assembling a solution from scratch as well as adapting a previous solution.

In the modification stage, an operator taxonomy would be used as described in [Alterman 88]. The idea is that if a particular operator appears in both a positive and a negative plan, a conflict exists. The conflict may be resolved by replacing the operator with another operator. Since the operator appears in a positive plan, it cannot be removed without introducing a goal violation. However, it may be possible to find an operator that works in the positive plan but fails in the negative plan. For example, the use of electronic detection equipment at the exit to a library allows patrons to leave without books or with checked-out books, but not with unchecked books. An operator taxonomy helps the planner find a different operator which can be substituted into a plan when the constraints on the plan are changed. We describe the relation of this work to that of the planner in [Anderson&Fickas 1989]. We are in the process of implementing a similar method for use in the modifier.

4.3 Discussion

In the planner we emphasize the issues of modeling the environment, model completeness, and the general dialog model of control we embrace. We discuss several other issues as well.

I1: The requirements language used by clients in complex domains is often ambiguous.

The planner requires a specific initial state and goal state; clients generally describe a class of problems in a vague, general way. In particular, the initial conditions tend to be left out. One reason for this is that in the ideal case, the goal should be achieved regardless of initial conditions, e.g., patron P will be able to check out book B no matter what. Part of the dialog between client and analyst (issue 6) involves clarifying the preconditions that restrict actions which achieve the goals.

I2: A single agent cannot be relied on to state non-conflicting requirements.

Frequently, there are no obvious conflicts when looking only at the goals and constraints of the client. It is not the goals themselves, but the limitations of the available methods of achieving those goals that are in conflict. For example, allowing patron P to check out a book when patron Q already has the book is a conflict in the case of paper books, but not with electronic books.

I5: Agents have preferences and bounds they are willing to live (and bend) within.

Traditional planners do in fact treat goals as all-or-nothing. If a goal cannot be achieved as stated, the planner fails. Thus, a separate module is required to modify goals and constraints if modifying the operator set does not resolve a conflict.

I6: Requirements engineering involves a "dialog" between a) the goals and constraints of the client and b) the methods that can be provided by existing technology.

As implemented, the planner performs a monolog instead of a dialog. Given a complete set of goals and constraints, the planner uses means-ends analysis to find methods that achieve the goals. The next version of the planner, currently being implemented, uses abstract operators to allow the dialog to become two-sided. Abstract operators can be used to represent very vague goals and constraints; as the operators are specialized, the client will generally have to provide more details so that the planner can choose between alternative methods for achieving the same goals.

I7: Conflicts lead to negotiation and resolution.

Two general techniques can be used for overcoming a problem reported by the planner: 1) manipulate the goal/constraint set of the system; 2) manipulate the candidate operator set of the system. In the first case, we are saying that we are willing to forego (or restate) a goal or abandon (or restate) a constraint. In the second case, we are saying that we are willing to add to or delete from the set of goal achievement techniques available to us. Stated broadly, the first says we are willing to modify our needs, the second that we are willing to modify the technology that we make available for achieving our needs. As mentioned earlier, the planner itself does not assist in modifying goals and constraints. The planner, like the critic, does not represent relative worth of goals or constraints, but simply notes specific conflicts.

I8: We need to model the system environment.

Very few goals are achieved by the artifact in isolation. More typically, the patron uses the artifact as a instrument in achieving goals. Without modeling the actions available to the patron in the environment, the planner would be unable to connect the client's goals with the artifact operators.

I9: The notion of completeness of a domain model is strictly relative.

One of the central focus points of our current work on the planner involves the use of abstract operators as initial, partial commitments to the requirements. In generation, partial commitment at an abstract level is useful as a way of avoiding detailed design work until the general framework is established. As already mentioned, this allows the client to state their goals in very general terms. The planner can then request additional information as needed.

In critiquing, the ability to critique abstract plans eliminates unnecessary work. If a constraint violation occurs at an abstract level, then it can be detected and fixed before extra work has been done to specialize the requirements.

I10: Cases or scenarios seem to be an important tool in understanding how, and validating that, a set of requirements has been met.

If we view a plan as a case, then reasoning about cases is frequently not only useful but necessary to reveal conflicts between goals. Many conflicts between high level goals cannot be detected without looking at individual plans. Reasoning about client goals such as "allow patrons access to a useful working set" is too vague; for example, there is no way to note that simply having a check-out operator is not sufficient to satisfy this goal.

5. The Resolver

One of the major lessons we learned from observing analysts interact with clients is that negotiation is an inevitable piece of a model of requirements engineering [Fickas et al 1987], [Robinson&Fickas 1990]. Negotiation can come in various forms: specifying the meaning of ambiguous goals; deciding on the methods one is willing to employ to achieve goals; deciding on what is minimally acceptable as goal achievement; compromising when conflicts arise. This leaves us with two questions: 1) how can we formalize the negotiation process so that we at least capture it as part of the documentation of a specification, and 2) can we automate some or all of the process.

The resolver addresses each of these questions for the negotiation tasks associated with conflict and compromise. Further, the resolver recognizes that complex systems may have many user groups or clients, each with their own point of view. With multiple perspectives comes the potential for yet more conflict, thus making the need to address the compromise problem all the more compelling.

The resolver model shares two basic components with the critic: a goal language that represents the requirements of a system, and an extended Petri-net language as the target or output of the requirements engineering process. An early version of the resolver replaced the cases of the critic with transformation rules that were able to generate goal achievement techniques [Robinson 1987]. The current version, which we will discuss here, explicitly lists goal achievement techniques [Robinson 1990]. Both versions represent links between goals and specification components, a vital connection in terms of our model of negotiation.

The resolver adds a further dimension to requirements engineering, the concept of multiple needs or goal spaces [Bearman&Fickas 1988], [Robinson 1987]. In contrast, both the critic and planner assumed that a single agent was stating requirements or that the goals of multiple agents had been previously integrated into a single view *prior* to running either tool. As

can be seen in figure 8, this assumption leaves implicit an important part of requirements en-

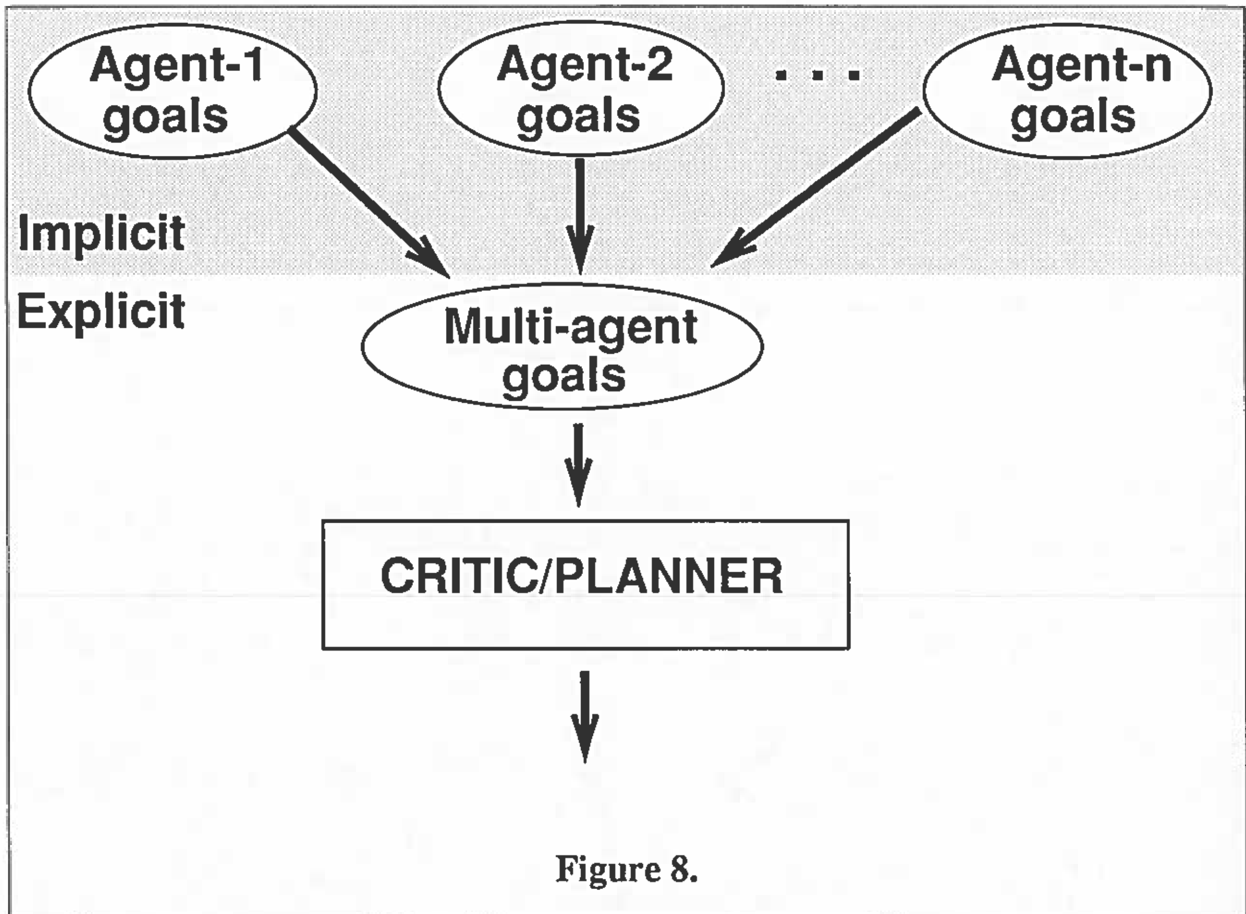

Figure 8.

gineering: the capture and integration of multiple system perspectives. As shown in figure 9,

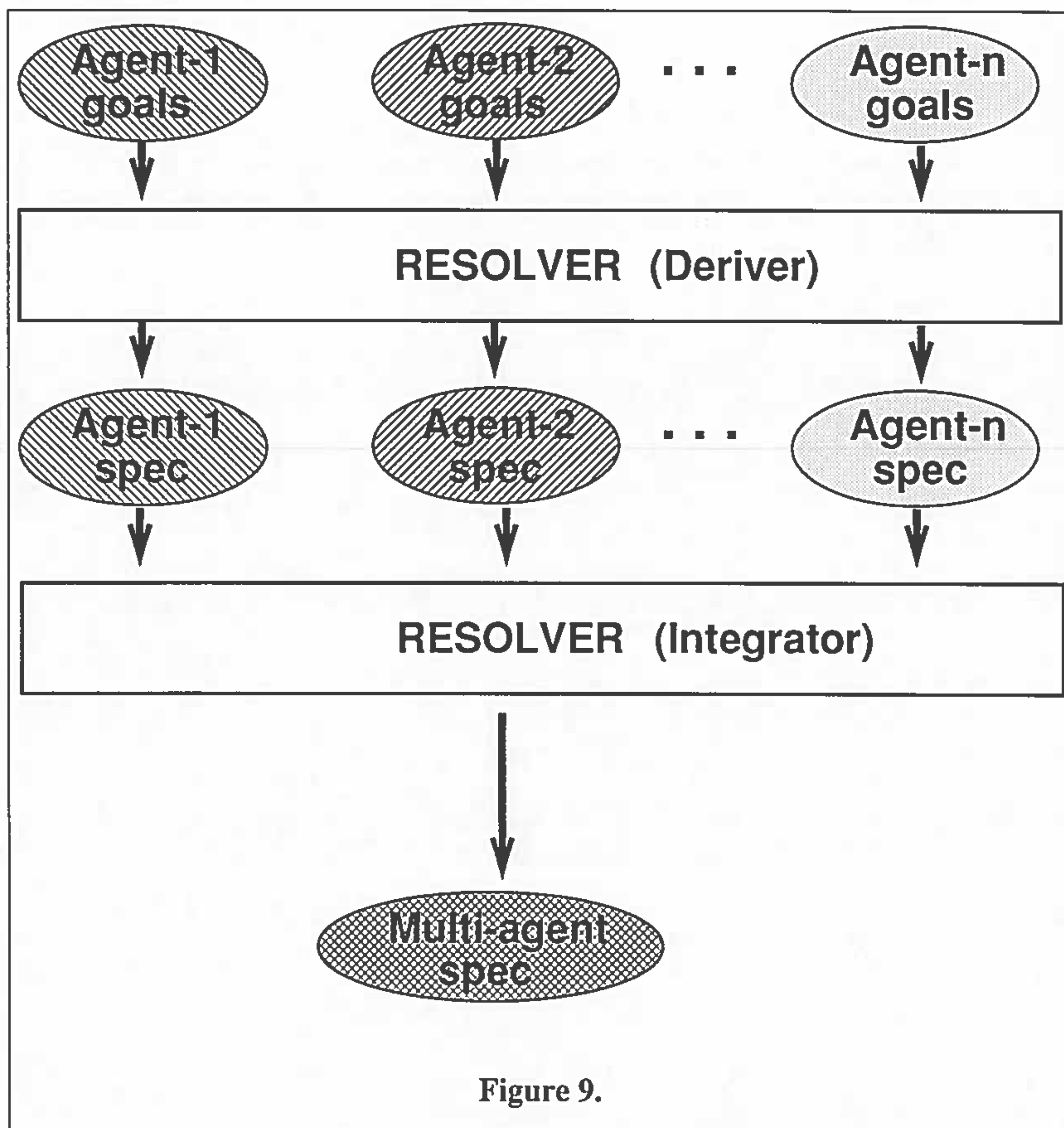

Figure 9.

the resolver attempts to address this problem by representing a perspective or goal space for each separate agent of a system (e.g., a patron's goals, the library administration's goals, the library staff's goals, etc.). From each goal space, a separate specification is derived. The final step is to integrate or join each of these independently derived specifications into a single, consistent, ready-to-implement specification.

It is important to note that each of the n specifications in figure 9 are rationalized by the goals and problem solving steps used to derived them. For example, in figure 10 we can see that two specifications have been derived by two different agents. Each specification includes the goals (and sub-goals) that lead to its final form, documentation that will become impor-

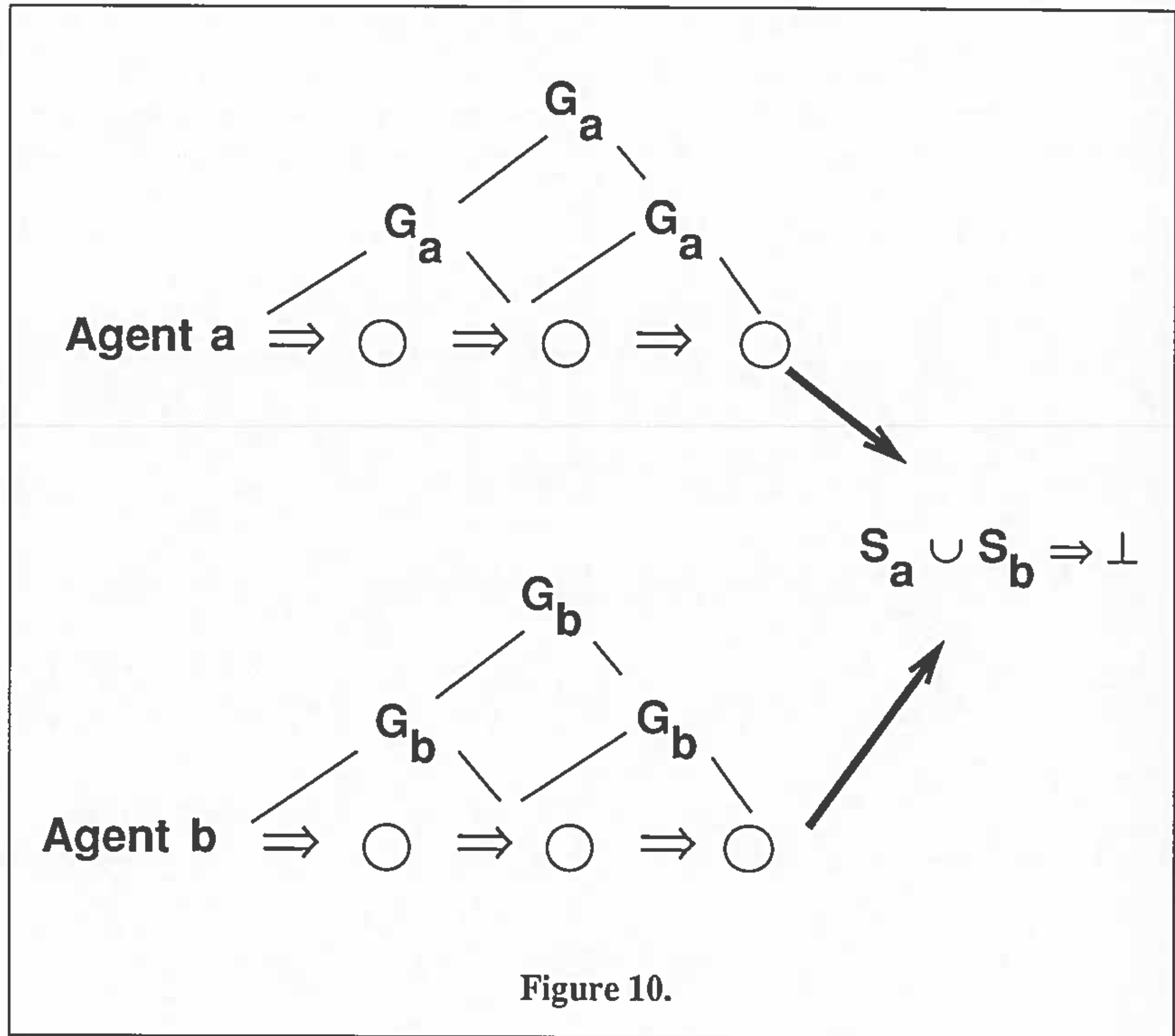

Figure 10.

tant later because of the impossibility of doing a simple union between the two.

An example may be helpful here. Suppose that we have a two agent library problem with a patron agent and a library administration agent. Using the resolver, each derives a separate specification. As shown in figure 11, the patron agent derives a library that has unrestricted

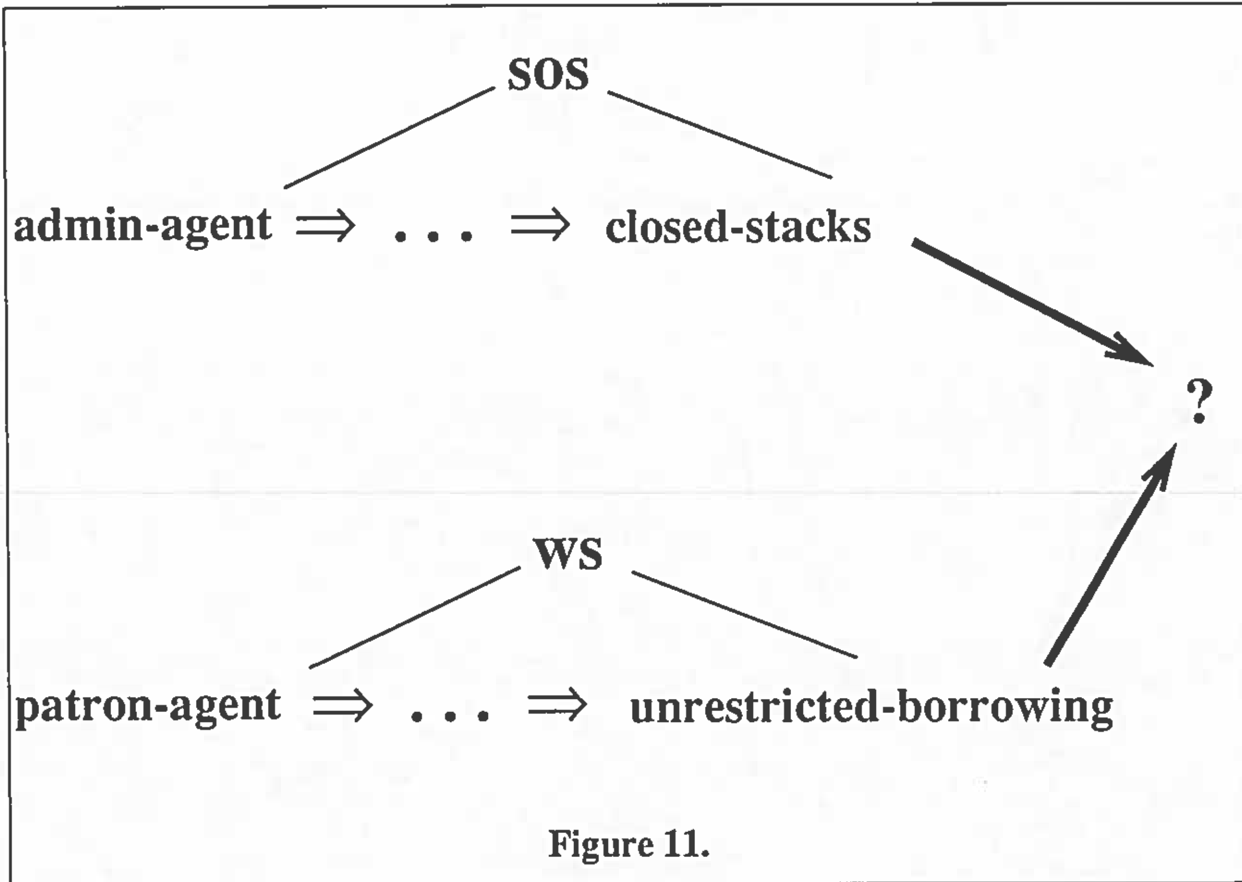

Figure 11.

borrowing. This derivation is driven by the general goal of gaining access to a large working set of library materials (shortened to the goal WS in the figure). On the other hand, the administration agent derives a library specification that has closed stacks (i.e., staff-assisted, on site, short term borrowing). This derivation is driven by the general goal of keeping stock on the shelves (shortened to SOS in the figure). A simple union of the two specifications is impossible.

One might be tempted to view the conflict as one between closed stacks and unrestricted borrowing. The resolver takes a different tack. It attempts to thread back through the rationale of both specifications to find the *goals* in conflict for each agent, in this case SOS and WS. It is from here that negotiation proceeds. Figure 12 shows this graphically.

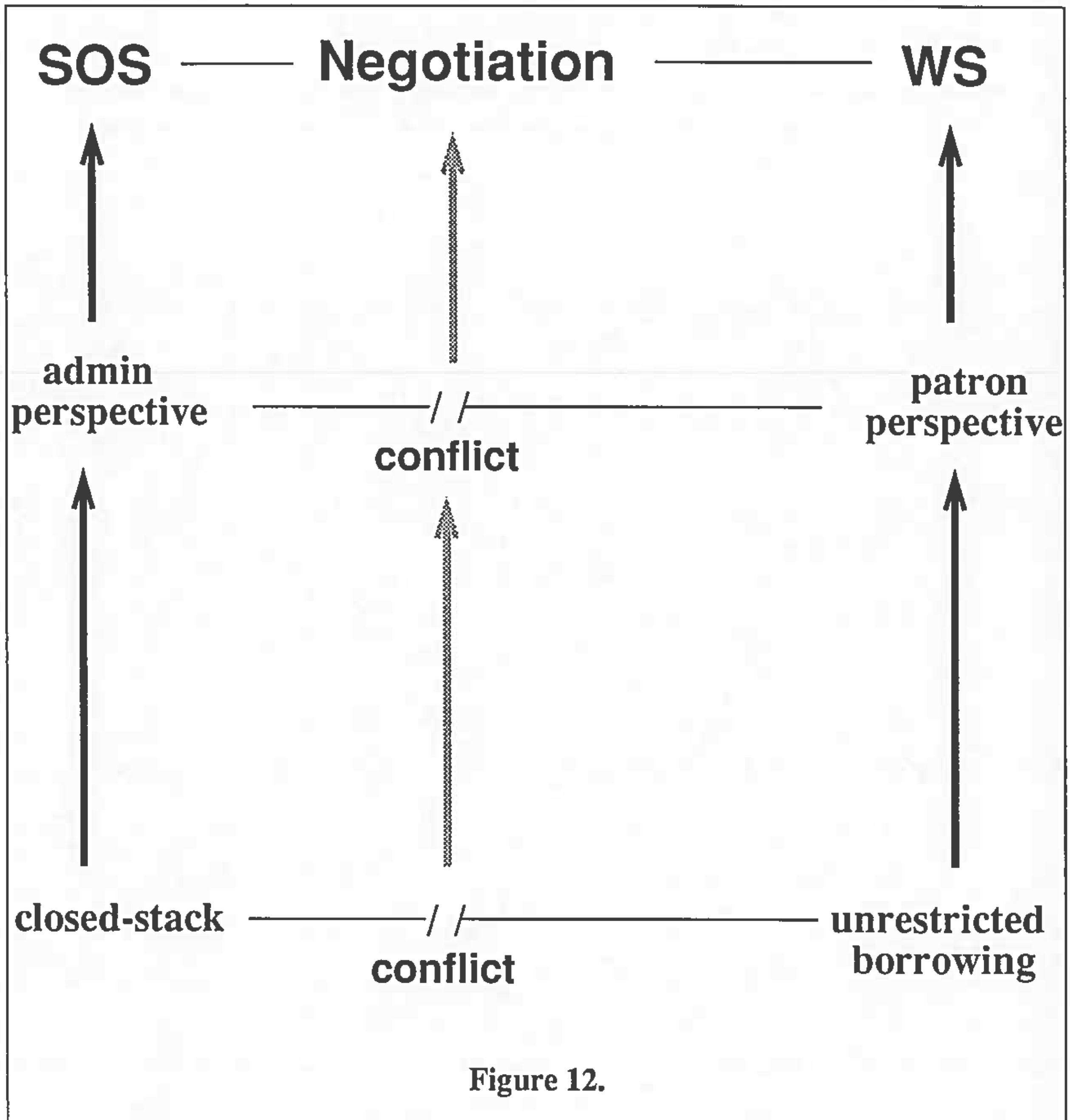

Figure 12.

Given a goal conflict, we are left to choose a negotiation style. The current resolver employs three general strategies: compromise, dissolution, and compensation. A fourth method, case-based reasoning, will be incorporated at a later date. We will discuss each in turn.

Compromise - when goals cannot be fully satisfied, consider less preferable means. For ex-

ample, closed-stack is a means to satisfy SOS; similarly, unrestricted borrowing is a means to satisfy WS. While both means may be the most preferable method of satisfying their respective goals, they can't be achieved together. Instead, during compromise we consider other (preferably non-interfering) means of goal achievement. For example, we could also achieve SOS by purchasing multiple copies of popular books. In general, agents may be willing to accept less preferable means to achieve their goals in order to resolve their conflicts.

To construct compromises, we need to represent information lacking in the critic and planner. First, we need to represent preferences among alternative goal achievement methods. Given a goal G, and a set of achievement methods A = {M1, M2, ... Mk}, we represent an ordering on the power set of A. For example, suppose we had listed two methods for the goal SOS, {closed-stacks, buy-extra-copies}. The client/agent must state a preference ordering (possibly partial) on {}, {closed-stacks}, {buy-extra-copies}, and {closed-stacks, buy-extra-copies}. Second, agents may also represent preferences between goals, e.g., SOS is preferred over WS. Finally, preferences are represented between the needs of agents. For example, a manager's goals can be preferred over a secretary's; yet, some specific secretary's goals can be preferred over the manager's. These three types of preferences (means, goal, and perspective) vary as agents consider various resolutions [Pruitt 1981], [Zeleny 1982]. The manipulation of such preferences is directly addressed in the resolver's interactive preference tool [Robinson 1990].

Dissolution - when goals conflict, consider removing the conditions causing the conflict. This negotiation technique attempts to restate the problem to dissolve the conflict. In our model, it leads to a change in problem representation. For example, in our simple library problem, only one type of resource is under consideration: books. The resolver has a dissolution heuristic for resource management conflicts that falls under the general heading of divide-and-conquer:

> If there is a WS versus SOS conflict on resource R, then try dividing the resource into sub-classes and reconsider the conflict for each sub-class.

Thus, divide books into sub-classes (rare books, reference material, journals, etc.), and reconsider the check out policy for each sub-class separately. Or, as used by a local video rental store, divide video tapes into old and new, each with a different rental policy. Expert human negotiators employ a powerful set of such heuristics [Raiffia,82], [Pruitt,81]; we have only begun to scratch the surface with heuristics like the above.

Compensation - when a good resolution can't be found, compensate the unsatisfied agents. Compensation increases the number of goals under consideration. For example, if a resolution could not be found which allowed patrons unrestricted borrowing, consider reducing late fines on patrons. While it does not satisfy the direct needs of patrons, compensation eases their burden.

Case-based reasoning - if the current conflict is similar to one previously considered, use the old case to construct a resolution. Both cases that have resulted in a satisfactory resolution, as well as those which resulted in an impasse, are useful. Successful cases can be instantiated and modified for the current context. Unsuccessful cases can prune compromises, dissolutions, and compensations from consideration. The case-base is a cache for integration and negotiation knowledge. While the resolver currently is not equipped with this method, Sycara has demonstrated its usefulness in the labor relations domain [Sycara 1988]. We al-

so note that the problems with using a CBR approach in requirements engineering are the same as those for CBR in general: how can we index and retrieve past cases; how can we reason about *differences* between the current conflict and the past situation.

5.1 Discussion

We have built a tool that supports the resolver model [Robinson 1989]. As shown in figures 12 and 15, the tool represents separate goal spaces for each agent, manages multiple specification derivations, and supports their integration.

Our initial use of the tool was to verify that a hand simulation of specification construction reported in the literature [Feather 1989] could be formalized and mechanized by the resolver. This effort is reported in [Robinson 1987]. Our current use of the tool is to test out various models of conflict resolution and, in particular, negotiation among self-interest groups or agents. This work is reported in [Robinson 1990] and [Robinson&Fickas 1990].

The resolver addresses the first seven requirements engineering issues of section 2:

I1: The requirements language used by clients in complex domains is often ambiguous.

The resolver is built around a model of goal exploration. It assumes a cycle of goal statement, method definition, conflict, goal restatement, further method definition, until a satisfactory definition of achievement has been reached. All of this takes place in a context of multiple agents, each with their own needs.

Taking an example from section 3.1, we might state a (quite ambiguous) goal of small-library. We might initially list an achievement technique of restricting acquisitions, i.e., keeping the number of books small, one interpretation of small in figure 3. However, after further negotiation, potentially brought on by conflicts with other goals and agents, we realize our real goal is to keep the number of staff small. If we can do this with a large book collection, fine. Why didn't we just come up with the right defintion to start with? Or alternatively, why didn't we better state the goal as small-staff-library? We believe that it is impossible in complex domains to be so precise, at least initially. A requirements engineering tool should expect that a complex negotiation process will evolve as goals are incrementally brought to the necessary level of specificity. Such a tool should allow freedom of ambiguity and force that ambiguity to be brought under control only when necessary, i.e., in a purpose-directed fashion.

I2: A single agent cannot be relied on to state non-conflicting requirements.

The resolver recognizes that an agent's goals will conflict, both in means and in ends. At its heart is a negotiation engine that attempts to overcome such conflicts.

I3: Agents are not homogenous.

The resolver maintains a separate goal space for all agents of a system.

I4: Two separate agents cannot be relied on to state non-conflicting requirements.

The resolver addresses this issue directly with its negotiation engine.

I5: Agents have preferences and bounds they are willing to live (and bend) within.

The resolver represents goal utility, goal preference, and preference of achievement techniques.. An agent is free to state hard and fast goals with completely fixed achievement conditions, or to state goal preferences with a wide range of acceptable achievement conditions.

I6: Requirements engineering involves a "dialog" between a) the goals and constraints of the client and b) the methods that can be provided by existing technology.

As noted under issue 1 above, the resolver's control model is based on such a dialog. In particular, one of its most effective capabilities is to show a user, in a graphic form, the effects of modifying the utility or preference on a set of goals or methods (Fickas&Robinson 1990].

I7: Conflicts lead to negotiation and resolution.

The resolver both aids in the conflict negotiation process, and documents the result. The latter, in particular, is an extremely valuable form of system documentation. For instance, why does the University of Oregon library have recall-on-demand? It is the result of a compromise between a patron's goal of obtaining a large working set and the library administration's goal of keeping adequate stock on the shelf. This answers the question of when we should be allowed to change the recall-on-demand policy: 1) when either of the two goals change, 2) when the preference between the two agents change, or 3) when we can think of a better compromise.

6. Conclusions

In evaluating the results of our work on Kate, we find equal measures of optimism and pessimism. Our optimism springs from a clearer understanding of the issues that we must address. We are also optimistic that semi-automated tools can be built to support our complex view of the requirements engineering process, although no single one of our tools to date addresses all of our requirements issues.

Our pessimism arises from the relatively small nature of domains in which we have been able to work effectively. For instance, while we would not call the library domain toy -- it is a complex system by our definition -- it is small in comparison to many requirements-driven domains that we would like to study. In particular, we have not addressed several prevalent issues in current specification research, e.g., real time systems, distributed systems.

We close by noting the three themes that we are drawn to again and again in our work:

Heterogeneity. Multiple perspectives converge when building complex systems. They should be represented both separately and as they are integrated into a whole.

Freedom. In rationalizing the specification language Gist, London&Feather use the notion

Related to the issue of scenarios being an important part of requirements engineering, Benner and Johnson have noted the usefulness of using scenarios to analyze certain aspects of a specification [Benner&Johnson 1989]. Roughly speaking, their scenarios can be viewed as similar to the critic's cases.

Our view of system implementation as mapping goals to operational components is certainly not new. For instance, Mostow's FOO system [Mostow 1983] started with non-operational goals and used transformations to arrive at an operational program. As an example, for the game of Hearts FOO would accept a goal such as avoid-taking-points. Transformations were defined that mapped this type of high level goal into lower level goals. Eventually, primitive goals could be mapped to actual Lisp code. The result was a program for playing Hearts with heuristics compiled in. In essence, FOO attempted to leap from requirements to implementation without stopping at the specification level for breath.

In later work, Mostow and Voigt examined the implicit design decisions that go into building software [Mostow&Voigt 1987]. 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.

The PADDLE system [Wile 1983] also shares some of the same goal-mapping concerns as that of the resolver system. A PADDLE goal is non-operational - it derives its meaning by listing the ways it might be implemented (either through other goals or primitive text editing commands). However, PADDLE does not support multiple perspectives, conflict, or anything other than lexical preference.

The Glitter system [Fickas 1985] focused on constructing an implementation from a specification. Like the resolver, it employed a non-operational goal language. Like FOO, it used transformations to map goals to operational components. Also like FOO, it lacked any notion of multiple perspective, and did not support conflict (and hence compromise). However, it did provide an ordering on alternative goal achievement techniques through a set of choice rules. While these rules were concerned with implementation concerns, we suspect that similar rules might be used to order and prune goal achievement techniques in the resolver.

One issue raised in this paper is the freedom of *incompleteness* during requirements. The construction of a complex artifact like a formal specification obviously will follow an evolutionary process. While this is a fine-sounding proclamation, we are left with the details of carrying it out. In the planner we use abstraction to hide missing detail. Goldman [Goldman 1982] examines three of the evolutionary steps one might take to later fill in that detail:

11. Increase the *structural* granularity of the specification. For instance, we may break a monolithic object into sub-parts, revealing more structure in a state.

12. 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.

13. 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 (A.K.A. incomplete) 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 aid both in building, and later understanding, a formal specification. At least two systems, PHI-NIX and the Requirements Apprentice, seem to be at least partial existence proofs of his speculation.

The PHI-NIX project is an attempt to apply domain dependent knowledge to the software specification and implementation problem [Barstow 1985]. 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 and *ambiguous* 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 on construction, there is no notion of interacting with the user to further refine his or her problem. The relation to our work is the use of domain knowledge to fill in the details of a problem. PHI-NIX relies on rule-based heuristics to guide it to a reasonable formal representation (and later an implementation) of a informally stated problem.

In a similar fashion, the Requirements Apprentice used cliches of typical domain components to help a user fill in pieces of a high level, incomplete specification [Rich et al 1987].

We finally note that each of these systems is based on an key assumption: the goals as stated are correct. In our work in requirements engineering in general, and the resolver in particular, this restrictive assumption is not held - virtually everything may be up for readjustment during the negotiation process. In summary, the dialog model, and the intertwining process it engenders, are basic tenets of our work.

Acknowledgments

We thank current and past members of the Kate project -- Keith Downing, Brian Durney, Rob Helm, P. Nagarajan, Susan Osofsky -- for their efforts. Eck Doerry provided substantial conceptual and organizational advice. Lisa James provided valuable editing assistance. Finally, we thank the National Science Foundation under grant CCR-8804085 for support of this work.

References

[Alterman 1988] Alterman, R., Adaptive Planning, Cognitive Science **12**, 393-421, 1988

[Anderson&Farley 1990] Anderson, J., Farley, A., Partial Commitment in Planning, Submitted to *Proceedings of the 1990 AAAI Conference*, Boston, 1990.

[Anderson&Fickas 1989] Anderson, J., Fickas, S., Viewing Specification Design as a Planning Problem: A Proposed Perspective Shift, In *Fifth International Workshop on Software Specification and Design*, Pittsburgh, 1989. Also to appear in *Artificial Intelligence and Software Engineering*, D. Partridge (ed), Ablex, 1990

[Balzer&Goldman 1979] Balzer, R., Goldman, N., Principles of good software specification

and their implications for specification languages, In *IEEE Conference on Specifications of Reliable Software*, 1979. Reprinted in *Software Specification Techniques*, Gehani&McGettrick (eds), Addison Wesley. 1986

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

[Bearman&Fickas 1988] Bearman, M., Fickas, S., Issues in specification design using parallel elaborations, In *Australian Software Engineering Conference*, Canberra, 1988

[Benner&Johnson 1989] Benner, K., Johnson, W., The Use of Scenarios for the Development and Validation of Specifications, In *AIAA Computers in Aerospace VII Conference*, Monterey, CA, Oct 1989

[Feather 1987] 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

[Feather 1989] Feather, M., Constructing specifications by combining parallel elaborations, In *IEEE Transactions on Software Engineering*, Feb. 1989

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

[Fickas&Nagarajan 1988a] Fickas, S., Nagarajan, P., Being suspicious: critiquing problem specifications, In *Proceedings of the 1988 AAAI Conference*, Minneapolis

[Fickas&Nagarajan 1988b] Fickas, S., Nagarajan, P., Critiquing software specifications: a knowledge based approach, *IEEE Software*, November, 1988

[Fickas et al 1987] 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

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

[Goldman 1982] Goldman, N., Three Dimensions of Design, In *Proceedings of AAAI-82*

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

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

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

[Mostow 1983] Mostow, J., A problem-solver for making advice operational, In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 279-283, 1983

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

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

[Pruitt 1981] Pruitt, D., *Negotiation Behavior*, Academic Press, 1981

[Raiffa 1982] Raiffa, H., *The art and science of negotiation*, Harvard University Press, 1982

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

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

[Robinson 1989] Robinson, W., Integrating multiple specifications using domain goals, In *Fifth International Workshop on Software Specification and Design*, Pittsburgh, 1989.

[Robinson 1990] Robinson, W., Negotiation behavior during requirement specification, In *12th International Conference on Software Engineering*, Nice, 1990

[Robinson&Fickas 1990] Robinson, W., Fickas, S., Requirements freedoms, TR-90-04, Computer Science Dept., University of Oregon, Eugene, Or. 97403

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

[Swartout&Balzer 1982] Swartout, W., Balzer, R., On the inevitable intertwining of specification and implementation, Communications of the ACM, 25(7) (1982). Reprinted in *Software Specification Techniques*, Gehani&McGettrick (eds), Addison Wesley. 1986

[Sycara 1988] Sycara, K., Resolving goal conflicts via negotiation, In *Proceedings of AAAI-88*, 245-250, 1988

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

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

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

[Yue 1988] Yue, K., Directionality and stability in system behaviors, In *Proceedings of the 4th Conference on AI Applications*, San Diego, 1988

[Zeleny 1982] Zeleny, M., *Multiple criteria decision making*, McGraw-Hill, 1982