The Kate Project:
Supporting Specification Construction

Stephen Fickas, John Anderson
William Robinson

CIS-TR-90-24
December, 1990

Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

The Kate Project: Supporting Specification Construction

Stephen Fickas
John Anderson
William Robinson

Computer Science Department
University of Oregon
Eugene, OR. 97403

Abstract. Over the past seven years, a group of faculty and students at Oregon have been involved in the study of requirements and specifications of complex systems, ones that involve physical, social and software subsystems. The focus of this work, all falling under what we call the Kate project, has been finding support for the specification process. Our concentration has been on two related research efforts: 1) formalizing the specification construction process, and 2) building computer-based tools based on this formalization. The outcome of these two efforts has been a specification methodology and a set of tools that support it. In this paper we will 1) present the methodology and the issues it raises for formalization and automation, and 2) describe the tools that we have built and their results as they relate to the methodology and issues.

1. Introduction

Formal specifications are hard to construct. There have been two major research directions proposed to address this difficulty. In the first, the problem is perceived as that of language design; if we make the specification *product* language easy to program in (we use the verb *program* loosely here), or easy to reason about, or free from implementation detail, then we will see formal specifications being built faster, with fewer bugs, and by the masses (as opposed solely by the research group who defined the language). The majority of work in software specification centers on this view.

In the second view, one attempts to come to grips with the construction process itself. We must study how specifications are constructed from requirements, and provide the methodologies to guide the process and the tools to support it. While in the minority, there are research projects in software engineering that take this process oriented view. However, such projects typically center not on the specification process, but on the implementation process, i.e., the process that *starts* with a fully constructed formal specification. Our project, called Kate, is one that attacks the *specifica-*

tion problem from a process oriented view. In this paper, we will discuss our attempts, over the last seven years, to formalize and automate the specification construction process as part of the Kate project.

The paper is laid out as follows. Section 2 presents the specification construction process as a two-step cycle. It is this model or methodology that we use as the heart of further discussion. This section also presents three issues that have come to the forefront in our research efforts. For the most part, they remain open research problems for us. Sections 3, 4, and 5 have a common purpose of presenting diverse Kate projects in an understandable, unified way. Section 5, in particular, discusses what approaches we have taken in formalizing and automating our methodology. Section 6 discusses the pluses and minuses of these approaches in relation to the overall goals of the project.

2. A specification construction methodology

We propose a model of specification construction that is based on an incremental cycle of validation and modification, as shown in figure 1. A development state consists of 1) a set of goals to be

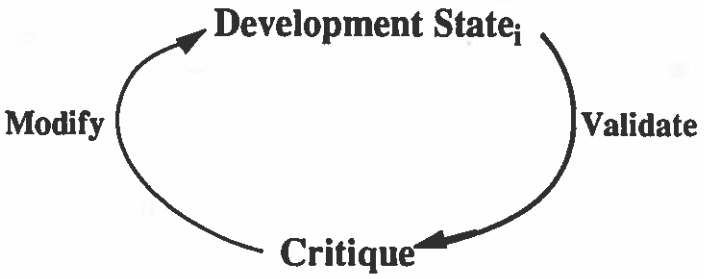


Figure 1

achieved, 2) an operational model that has been proposed for achieving them, and 3) links between the goals and the model. The validation process checks for three types of problems in a development state:

Goal verification. Does the model effectively achieve the goals? Goals represent states to reach or avoid. Are those states reachable or avoidable given the model?

Model validation: Is the model realistic? If it takes a naive view of the environment, then goals which appear to be met in the specification may not be met when the system is actually delivered.

Goal validation. Even if the model is valid and the goals are achieved, are they the right goals? Do they allow behavior that might be deemed undesirable in typical systems encountered in the domain?

The answers to these three questions lead to a critique of the current development state. From this critique, certain model or goal modifications may be suggested. This leads to a new development state, and the cycle is repeated. Development halts when either the critique is empty, or the client

accepts the consequences of the critique, i.e., the cost of ignoring a critique is less than the cost of addressing it. The final development state reached contains the specification, in the form of the model[1], and the specification rationalization, in the form of links to the clients goals.

The result of this cycle is a sequence of development states that form a history of the specification construction process (see figure 2).

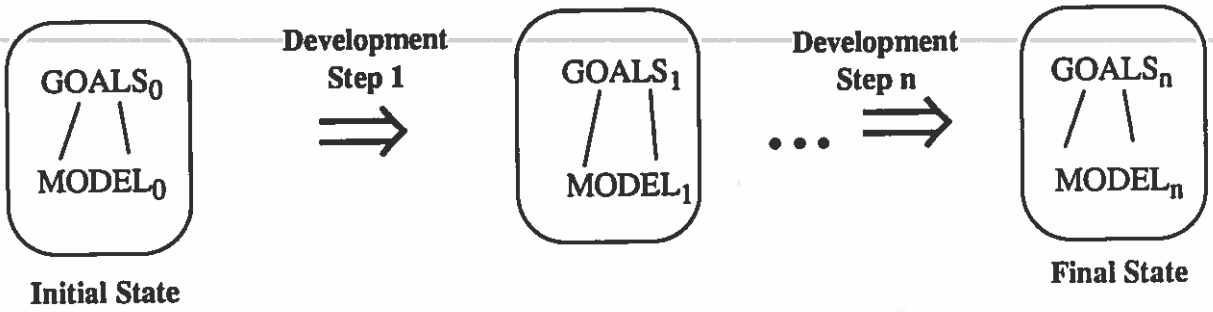


Figure 2

The following properties may hold for any development:

- $Model_0 = \varnothing$. It is not required that we start with a non-empty initial model. Typically we do so to reflect certain initial conditiions of the problem. However, we can also derive all that we need by noting goal failure problems (likely to be total with an empty model), and use the modification step to gradually build up the model.
- $Goals_0 = \varnothing$. It is not required that any goals be given in the initial state. In this case, any critique produced would focus on *typical* goals one finds in the application domain, noting their absence in the current development (see the example in section 5.1.1).
- $Goals_0 \neq Goals_n$. The original goals proposed by the client are modified during development. This type of modification may include either strengthening or weakening the goal set.
- $Model_0 = Model_n$. The model first proposed is deemed satisfactory as the final specification. All of the development steps, if any, are focused on the goal space (see above).

In a typical development, one would expect to start with non-empty goals and a non-empty model, and produce modifications to both the model and the goals. Further, in a typical development one would expect that alternative modifications might be considered at each development step. Figure 2 shows the final or ideal development path, one that does not illustrate the alternatives considered and rejected. We expect to capture these alternatives, and the criteria for selecting among them, as part of the development history as well.

---

1. In the context of our representation of a development state, we will use the terms specification and model synonymously throughout the rest of the paper.

2.1 Issues arising from the methodology

The methodology we have proposed is based on a cyclic validation-modification process. Our efforts to formalize and automate this process have raised three key research issues:

Issue 1. There is a tension between formalization and closed specifications. Useful specifications of complex systems must be closed; they must model both the artifact to be built and the environment that it interacts with. This implies the need for models of complex subsystems within the environment, e.g., human users, physical devices. Such models, and effective procedures to reason about them, push the boundaries of formal representation.

Issue 2. Specifiers must be freed from unrealistic constraints. Certain restrictions must be enforced in a *final* specification if one has any hope of a correct implementation being produced. These restrictions include completeness, consistency, and non-ambiguity [Meyer 1985]. However, attempting to enforce these restrictions during the specification *construction* process is futile. Clients lack a complete model of their proposed system, state inconsistent goals, and use highly ambiguous language. We view these as *requirement freedoms* that must be supported. In particular we expect that the road to a complete, consistent, non-ambiguous final specification must lead through incomplete, inconsistent, and ambiguous development states. Representing and reasoning about such states again pushes our formal methods.

Issue 3. A single perspective of a system must give way to multiple viewpoints. In complex domains, there are typically multiple interest groups, each with their own priorities and agendas. Engineered artifacts in these domains are a mixture of the goals of these groups. One must represent both the "requirements space" for each group and the negotiation process that leads to a common agreement. Thus, figure 2 is a simplification: we need a separate development line for each interest group we identify, and another process that integrates these into a final whole.

These three issues are problematic in isolation. Their interaction only exacerbates the problem. The tools that we report on in this paper do not individually address all three issues, nor does the union of our tools fully cover all issues. However, they do set useful boundaries on what is possible in terms of both formalization and automation.

3. A Common Example: the Library Problem

To illustrate the tools we have produced and the issues they address, we will use a common example problem, that taken from an automated library system. A typical description of a library system, as seen in the literature, is given in figure 3.

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.

[IWSSD 87]

Figure 3

The construction of a specification of a library system from such a description may appear simple, contradicting our claim to be using complex problems as specification examples. In particular, specification may be viewed as nothing more than taking a tracking or database application off the shelf (or finding the right database-tracking schema in a catalog as in [Rubenstein&Waters 1989]). Wing summarizes an opposing view:

> "Real libraries are not simple. They involve more than just people, books, and a database. They have policies according to who the borrower is, what kind of book it is, what time of year it is, and, of course, exceptions to all of these policies."

We concur. Libraries try to satisfy the conflicting concerns of users, staff and administrators; they employ many mechanisms to deal with both errorful and irresponsible behavior; they involve complex responsibility assignments among agents. In summary, libraries are complex systems which require sophisticated analysis to derive adequate specifications. Hence, it is not surprising that the issues that we identified in section 2.1 show up here.

Models of closed specifications. A closed specification of an automated library system may eventually involve the modeling of a complex environment, e.g., the "university" of a university library, the "city" of a city library. Formal models of even simple sub-pieces of such environments may be difficult to represent or reason about.

Requirement freedoms. Consider the request by a client for a *small* library (line 1 in figure 3). Among other things, this may be a constraint on the size of the book collection, the size of the patron set, the size of the operating budget, or some combination of each. Is this ambiguity harmful? Certainly it is in the final specification. However, during the specification construction process, we

may be able to leave *small* in its ambiguous state until demanded to disambiguate it. In particular, *small* can be viewed as an abstract concept that has known implications no matter what its eventual specialization entails. Further, the specialization process can be gradual. For instance, we might specialize small to mean small operating-budget. As before, this may suffice for most purposes. Eventually we would expect to provide hard data, e.g., less than $30,000 annually.

Shifting attention to incompleteness, clearly figure 3 does not include all of the objects and operations we will need [Wing 1988]. However, it is naive to believe that a client can generate a complete specification full borne. As shown in figure 1, we expect an initial model (like that in figure 3) to go through a validation-modification cycle, a process that *expects* incomplete models and is tailored to deal with them.

Multiple perspectives (and freedom of inconsistency). A real library is a complex, engineered compromise among competing concerns. Patrons that want to acquire library material have different goals than those already having acquired what they want. The goals of the library administration and the library staff may differ. Patrons themselves can be broken into different classes with different goals, e.g., undergraduates, faculty, general public.

While we cannot see any inconsistent or conflicting goals directly in figure 3 (in fact, we can't see any goals at all!), we can see the resulting residue left from conflicts and negotiation:

- A book limit (or a loan period) can be viewed as a compromise between patrons and administration. Patrons have a goal of gathering together a useful working set, no matter how large. The administration has a goal of having adequate stock on the shelves for patrons to use. A model that incorporates a form of limited check out satisfies a little of both.

- Restricting access to the borrowing records can also be viewed as a compromise between patrons and administration. Patrons have a goal of finding the book they want, no matter where it is. The administration has a goal (often a legal obligation) to protect the privacy of others. A model that allows a patron to view her own borrowing record, but not that of others is a compromise - it protects privacy in the main, but does leave the door open for illicit access by use of false ids or passwords.

As with other types of requirement freedoms, we view inconsistency as a natural part of developing specifications. Clearly, certain types of syntactic and logical inconsistency serve no useful purpose, and may be expunged immediately [Rubenstein&Waters 1989]. However, the knowledge intensive process of characterizing and reformulating conflicts, employing multiple resolution techniques, and searching through alternative goal achievement methods is all part of a larger specification design process, one that must address multiple perspectives and the inconsistency that is likely to develop among them.

4. A common representation

The tools that we have developed each use a slightly different representation, one that reflects their particular research focus. Instead of presenting each representation separately, we abstract to a level that allows us to view a single, unifying representation. While this is somewhat misleading -- it implies an actual unification that we have not formally carried out -- we believe it is outweighed by simplifying our presentation. The precise representation details of each tool can be found in our references.

In figure 2, the representation of a development state consists of zero or more goals to be achieved and a model (possibly empty) that has been proposed to achieve them. We look at first the representation of goals, and then the representation of the model.

4.1 Goal representation

Goals are represented as state descriptions. They can be broken into two classes: 1) descriptions of states we wish to achieve, and 2) description of states we wish to avoid. As an example, we might expect the following goals in a library system[1]:

- *achieve*: has-possession(Patron, Book)
- *avoid*: misshelved(Book)

There is no restriction on stating conflicting goals, either of the form[2]

- *achieve*: p(X)
- *avoid*: p(X)

or of the form

- *achieve*: p(X)
- *achieve*: q(X)

where p and q are mutually exclusive, or of the form

- *avoid*: p(X)
- *avoid*: q(X)

where ~p(X) and ~q(X) are mutually exclusive. We expect each of the above type of conflicts to arise, and we have developed tools to deal with them.

We use a library example in this paper, and hence, each tool can be expected to represent goals from the following general classes (broadened slightly from libraries to the more abstract concept of borrowing systems):

1. Allow users to have a large selection to choose from.
2. Allow users to gain access to a useful working set and keep it as long as necessary.
3. Maintain the privacy of users.
4. Recognize the human dynamics of group (patron, staff, administration) membership.
5. Account for human foibles, e.g., forgetting, losing items, stealing.

---

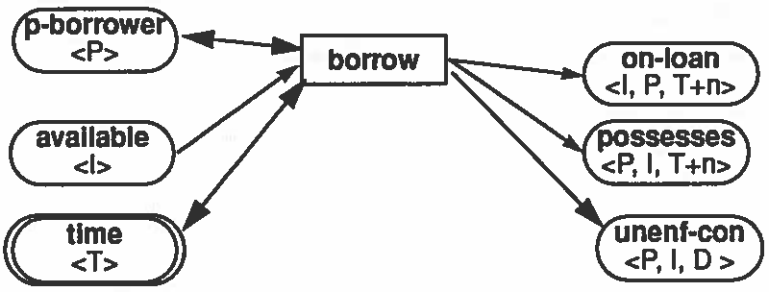
1. For simplicity, we present a single-agent perspective. In reality, the achieve goal might be that of a patron, but not that of the administration. The avoid goal might be shared by the patron and administration, but not by the staff.
2. We call the first form *direct* goal conflict. All other forms are called *indirect* goal conflict.

6. Account for development resource limitations, e.g., money, staff, and time available to develop the system.
7. Account for production environment limitations, e.g., money, staff, and time available to run and maintain the delivered system.

Not every one of these goal classes is equally important in every case. For instance, a client specifying a small reading room for an academic department might view 2, 6 and 7 as important. A client specifying a city library might consider 1, 3, 4 and 5 as important.

4.2 Model representation

The model is represented as a state-transition language. It takes the form of an extended Petri-net that includes object-based tokens, predicated transitions, and side-effects on token slots. We will use both a Petri-net and an operator form in presenting model components. As an example, figure 4 gives both representations for a simple borrow action with due date. In the graphical representa-



borrow (Patron, Item)

Uses:	potential-borrower (Patron), time(T)
Consumes:	available(Item)
Produces:	possesses (Patron, Item, T+n), on-loan (Item, Patron, T+n), unenforced-contract(Patron, Item, Due)

Figure 4

tion, ovals represent places. A double oval represents a place under an external agent's control. Tuples (e.g., <I>) represent the form of object tokens (with associated slots) that appear in a place. Boxes represent transitions. Unification is carried out among input tokens to the same transition. A double headed arc represents the use of a token without consumption or side-effect.

The example we will use throughout the rest of the paper is that of a borrowing loan-period. Figure 5 shows a model of a simple-minded return action with loan period to go along with the bor-

row action of figure 4. As can be seen, there is a non-deterministic choice between returning and

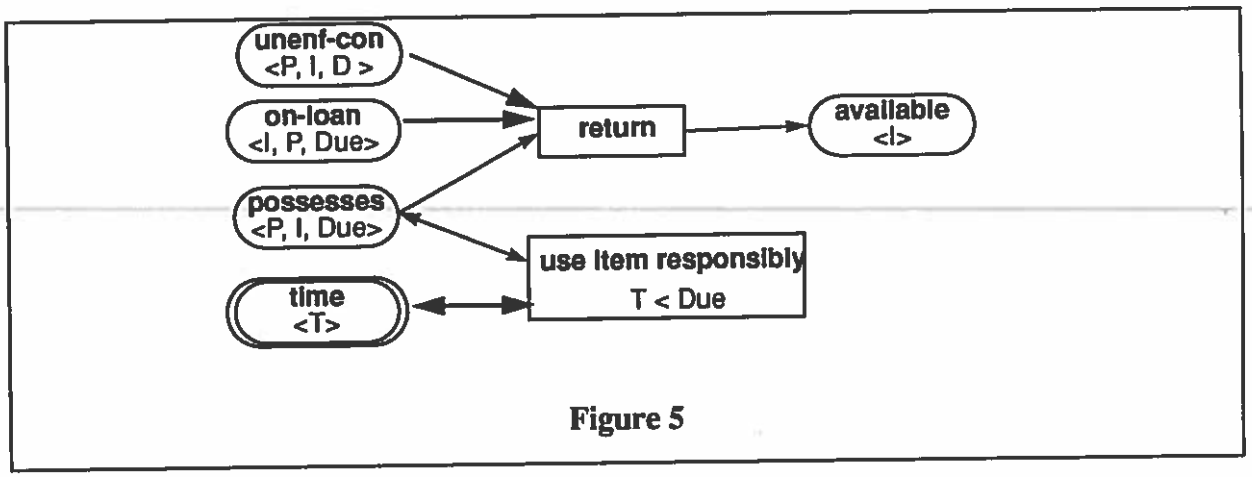


Figure 5

using an item, and hence, the model specifies a range of behaviors including 1) returning an item immediately after getting it (i.e., before using it), and 2) using it up to the last minute, i.e., returning it just before it becomes overdue. Unfortunately, this model is not realistic: for most borrowing systems it is unlikely that we can guarantee that borrowers will never use an item beyond its due date. We will take up this problem as we begin to discuss validation techniques.

5. Tools that support the methodology

There are two major processes in our cyclic methodology: validation and modification. In this section, we look at the tools we have developed to support each.

5.1 Validation

We have explored three different approaches to validation. The first uses stored examples of known problems to expose potential difficulties in a new specification. The second uses means-ends analysis to try to find sequences of actions that lead to goal violations. The third detects conflicts between two specifications of the same artifact, i.e., conflicts among separate perspectives or agents. In this section, we discuss each of these approaches, and the tools that support them.

5.1.1 Cased-based validation

Based on observations of human analysts, we noticed that known problem cases are frequently used to criticize a new specification [Fickas et al 1987]. We defined a case-based critic (cb-critic) to represent this type of criticism [Fickas&Nagarajan 1988a]. The cb-critic stores away both typical and boundary case scenarios that represent the behavior of agents of an environment (an agent may be a user, nature, or a physical component). These scenarios are used as abstract test cases for a system specification. For example, the cb-critic's test cases for a library specification include the following:

- A "run" on some part of the library's books (given unrestricted check out).

- Hoarding of some part of the library's books (given unrestricted check out).
- Stealing of library material (given unsupervised check out).
- Losing library materials (given any type of check out).
- Forgetting what one has borrowed (given any type of check out).
- The need to keep books a long time to complete a project (given a loan period).
- Gaining access to the borrowing records of others (given access to the borrowing records in general)

In the cb-critic, each case is supported by one or more scenarios illustrating the behavior in similar systems familiar to the analyst. For instance, one analyst we observed used the following scenario to illustrate the hoarding case [Fickas&Nagarajan 1988b]:

> "A secretary in Biology checked out all of the library's Biology books, and set up what amounted to a sub-branch of the library in their department."

The same analyst used two different scenarios to support the case against unsupervised reading of the borrowing record:

> "Government agencies have wanted to find out who's checked out the books on building hydrogen bombs."

> "Often within a larger setting, for instance, even the Science Library, faculty say 'Nobody, none of us have secrets from each other and we should all be able to know who has what, etc.' On the other hand, there have been many graduate students who say they don't like getting called Saturday morning at 8:00 saying bring in this book, by a person who could affect their future."

Our goal in the cb-critic was to represent general cases (and their associated supporting scenarios), and use them to find deficiencies in a system specification (i.e., the *model* of figure 2). To do so, we defined and implemented four major cb-critic components:

1. A fixed, predefined set of goals which acts as the index into a case catalog. From this set, the client chooses which goals are important for any particular problem. This chosen subset represents the goals of figure 2.
2. A case catalog. Each case is made up of 1) one or more goals that index the case, 2) an abstract sub-net that represents a pattern to be found in the client's specification, and 3) illustrative scenarios to be run. A case is *active* when one of its corresponding goals has been marked as important by the client.
3. A matcher that links active cases to specification components. The matcher incrementally matches a case pattern to pieces of the specification. It relies on the user to both confirm the matcher's proposals and to supply equivalence relations when they are not obvious.
4. A scenario enactor. Once a match is found for a case, the cb-critic will play the corresponding scenarios. Each scenario provides an initial marking and constraints on non-determinism to demonstrate the specific behavior required, e.g., repeatedly firing 'check out' but not 'return' to show a run on a library.

We illustrate the cb-critic by looking at a loan period example. Suppose that the cb-critic is given a development state that contains the model shown in figure 6 (where B = Book, D = Day), and a goal[1]

avoid: overdue(Patron, Item)

The cb-critic contains a case of unenforced-loan-period that is indexed by this goal, and hence, the case becomes active in the current development state.The case's pattern has been shown previously in figure 5.

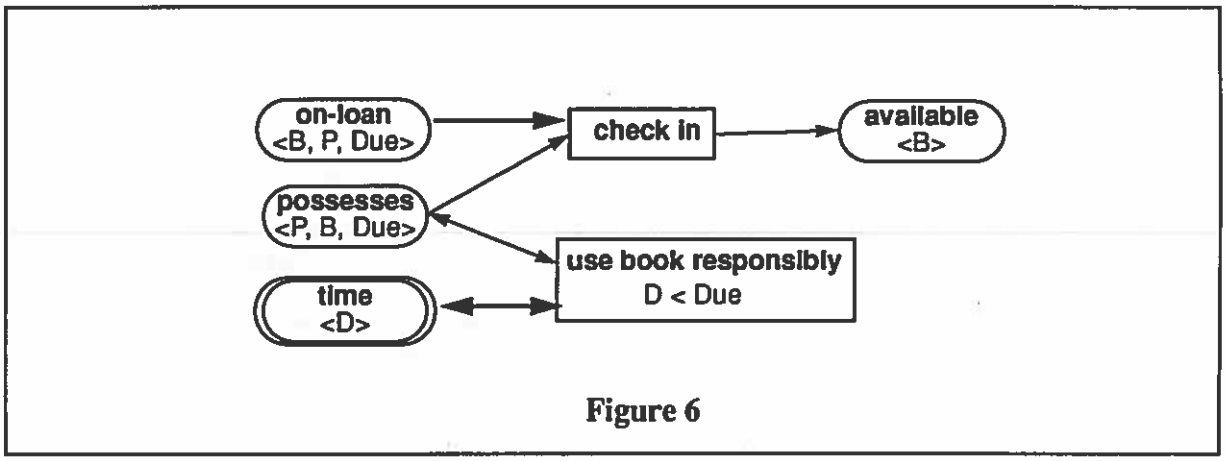


Figure 6

The human analyst assists the cb-critic in unifying the pattern in figure 5 with the model in figure 6.

Once the case is matched, the cb-critic considers the scenarios associated with the case. A scenario is one behavior out of many that the model is capable of generating. For the model in figure 6, the focus is on the non-deterministic choice between *check in* and *use book responsibly*. The behavior set ranges from a borrowed book being immediately returned, to a borrowed book being used until the very last moment.The cb-critic attempts to 1) demonstrate interesting landmark or boundary behaviors out of this set, or 2) extend the set by introducing new, plausible behaviors that are not being considered in the current model. The scenarios associated with the unenforced-loan-period case focus on the latter in particular. They represent the notion that borrowers may behave error-

1. overdue(P,I) is defined as possesses(P, I, Due) and time(D) and D > Due.

fully or irresponsibly: a borrower may forget when an item is due, decide to continue to use an item, irresponsibly, past the due date, or even discard an item after it is no longer useful. Each of these behaviors is represented by a scenario in the unenforced-loan-period case.

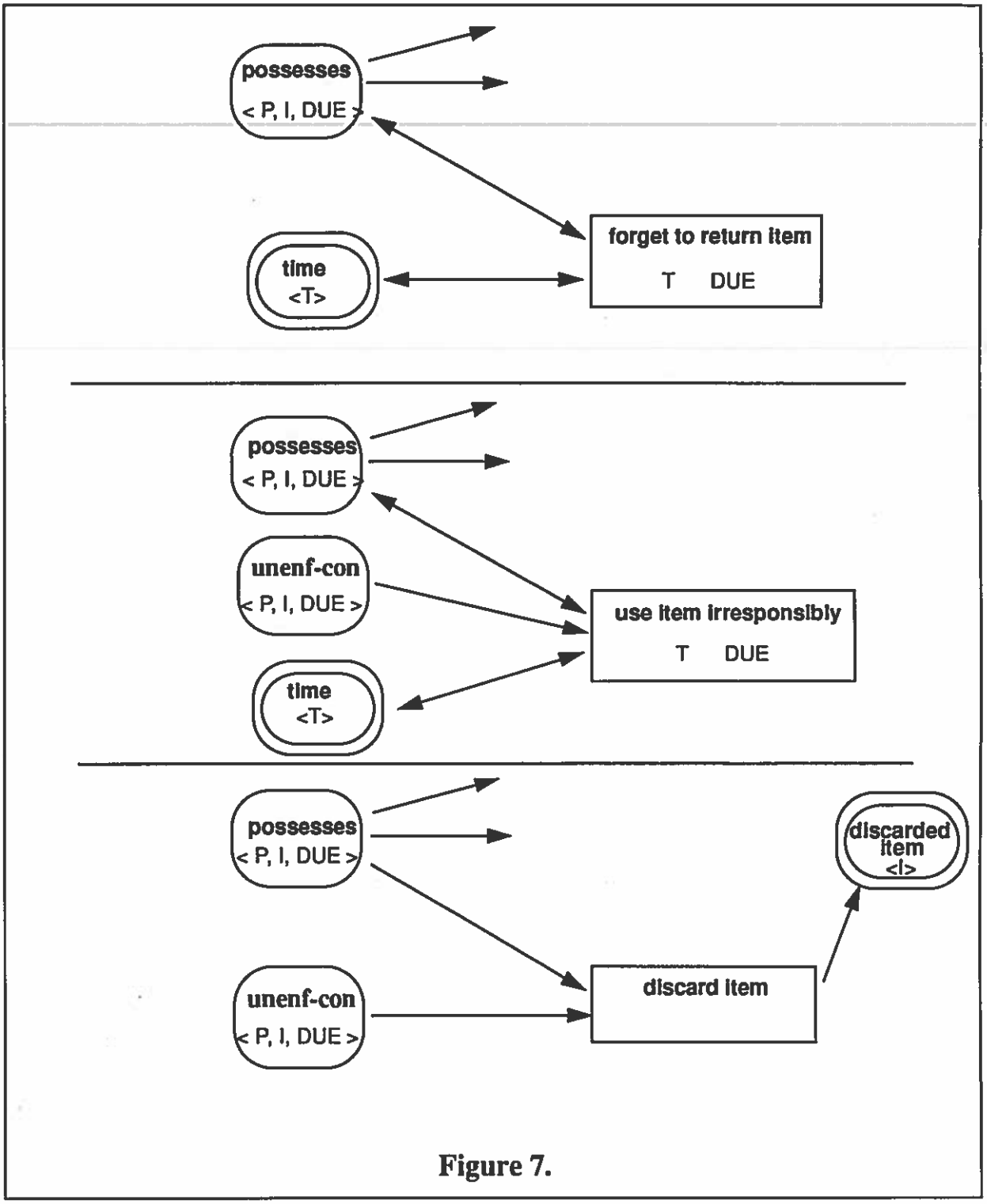


Figure 7.

As shown in figure 7, each scenario adds a piece of environment behavior to make its point:

- Forgetting involves giving a borrower a new option/transition, *forget to return item*, separate from using and returning an item. This transition allows an item to be past due.
- The *use item irresponsibly* transition allows a borrower to continue to use an item past its due date. It is predicated on having an unenforced contract with the repository.
- The third scenario adds a *discard item* transition that allows a book to become monotonically overdue: the item is consumed by the new transition and goes permanently to a *discarded item* place, never returning to the library. As with irresponsible use, discarding is predicated on having an unenforced contract with the library.

The cb-critic presents each of these scenarios by initializing the client's proposed model (i.e., marking the client's net), and then controlling the non-determinism to make the point.

It is interesting to note that the type of validation produced by the cb-critic in this example is not one of goal violation by the model; the overdue goal *is* avoided in the model of figure 6. Instead the cb-critic is criticizing the current model as incomplete. The cb-critic attempts to point out how any one of three plausible extensions will lead to goal violation. This is very much a knowledge-based approach, using knowledge of typical human behavior in borrowing domains to point out holes in what purports to be a closed specification.

5.1.2 Validation by Planning

The major advantage a tool like the cb-critic is reuse of analysis. The cb-critic does not have to *regenerate* scenarios from scratch for each new problem. Instead it uses goals to *index* into its store of analysis knowledge and *retrieve* applicable scenarios. However, in avoiding the regeneration problem, the cb-critic has introduced several new problems, each associated with case-based reasoning in general: 1) populating the case-base with sufficient cases to cover a domain is difficult, and 2) finding the correspondence between (i.e., matching) a past case and a current problem description is resistant to full automation.

We are exploring a tool based on plan-based validation that attempts to keep the advantages of the cb-critic while avoiding its disadvantages [Anderson&Fickas 1989]. The tool, called the pb-critic, generates new scenarios on the fly. However, once a scenario is generated the pb-critic uses an abstraction mechanism to store a form of the scenario for future use. In this way, the matching problem of the cb-critic is replaced by a more tractable specialization problem in the pb-critic.

The pb-critic uses a means-ends style of plan derivation to build scenarios from scratch. The pb-critic stores operators at various levels of abstraction, and builds plans to represent both goal achievement and goal failure. We will use the loan period example to demonstrate this. In particular, consider the three scenarios generated by the cb-critic (see figure 7) as criticism of a proposed model of library borrowing (see figure 6). We will follow the same three scenarios through the pb-critic.

The pb-critic generates one or more behavior threads through a non-deterministic cb-critic type model. However, the pb-critic does not use a Petri net style representation, but instead stores places as relations and transitions as primitive operator components that can be *composed* to build behaviors or scenarios.

Figure 8 presents a scenario/plan generated by the pb-critic to demonstrate that a borrower can irresponsibly cause an item to become overdue. Using means-ends analysis, the pb-critic looks for plans which achieve 'overdue book'. One operator which produces 'overdue book' is 'use irresponsibly'. This operator requires that the book be on loan to the patron, that the contract be unen-

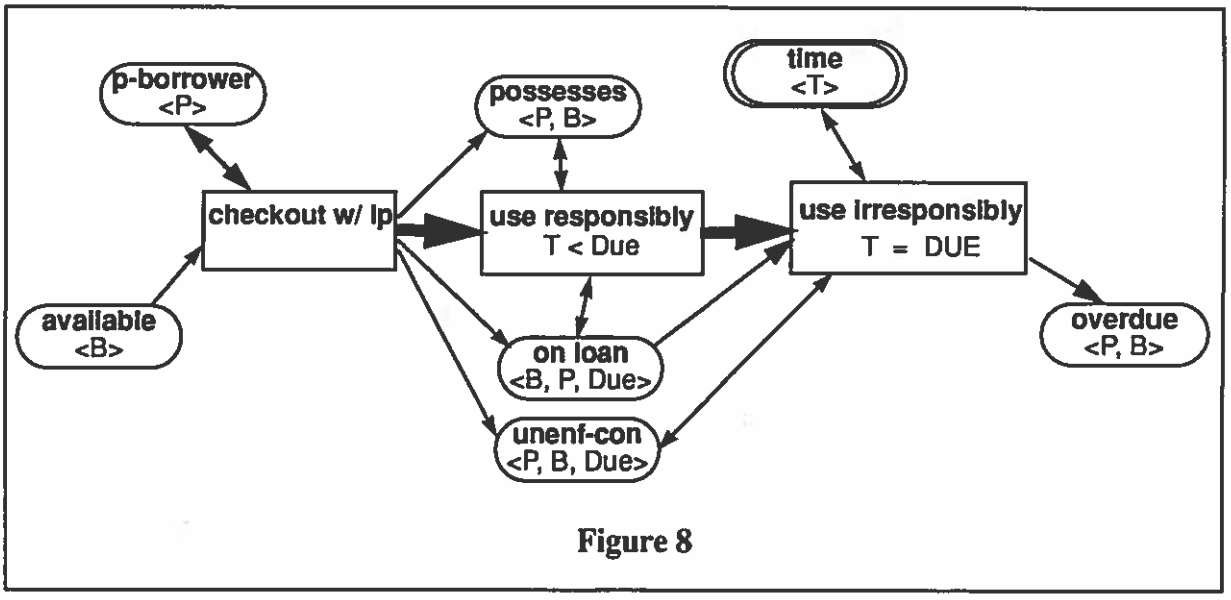


Figure 8

forced, and that the due time matches the current time. Similar scenarios/plans could be generated for the discarding and forgetting scenarios of section 5.1.1.

Rather than discovering each problem individually, it would be useful to find all of the problems at once. To do so, the pb-critic not only stores primitive operations, it stores abstract operators as well. For instance, the three operators used in the cb-critic's scenarios in figure 7 could be stored in the pb-critic as specializations of a single abstract operator:

allow-loan-period-to-expire (Patron, Item, Due, T)

uses: time(T), T = Due

consumes: on-loan (P, I, Due)

produces: overdue (P,I)

Figure 9 shows the pb-critic's taxonomy of operators that produce the relation 'overdue book'. Each of the specializations is a descendant of the abstract 'allow loan period to expire' operator.

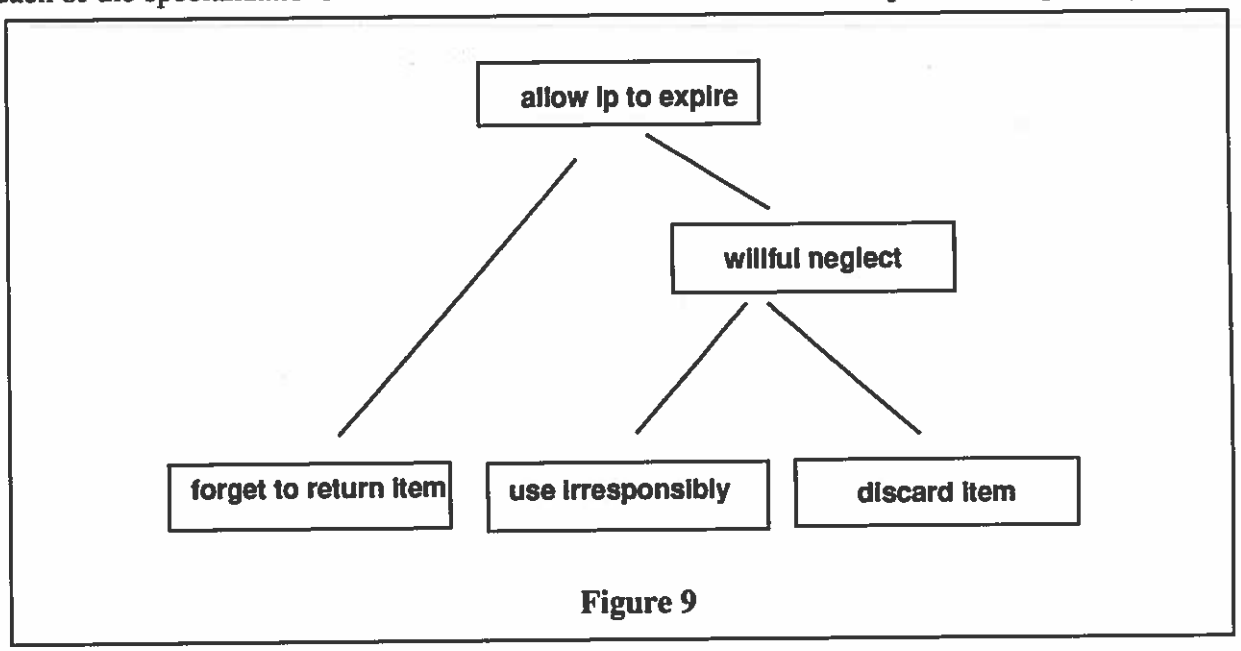


Figure 9

The abstract 'allow loan period to expire' operator can be used in planning as if it were a primitive operator. Thus, all three scenarios (forget, use irresponsibly, discard) are captured in a single abstract plan, as shown in Figure 10. This not only reduces the cost of critiquing, but, as we show later, may make modification easier.

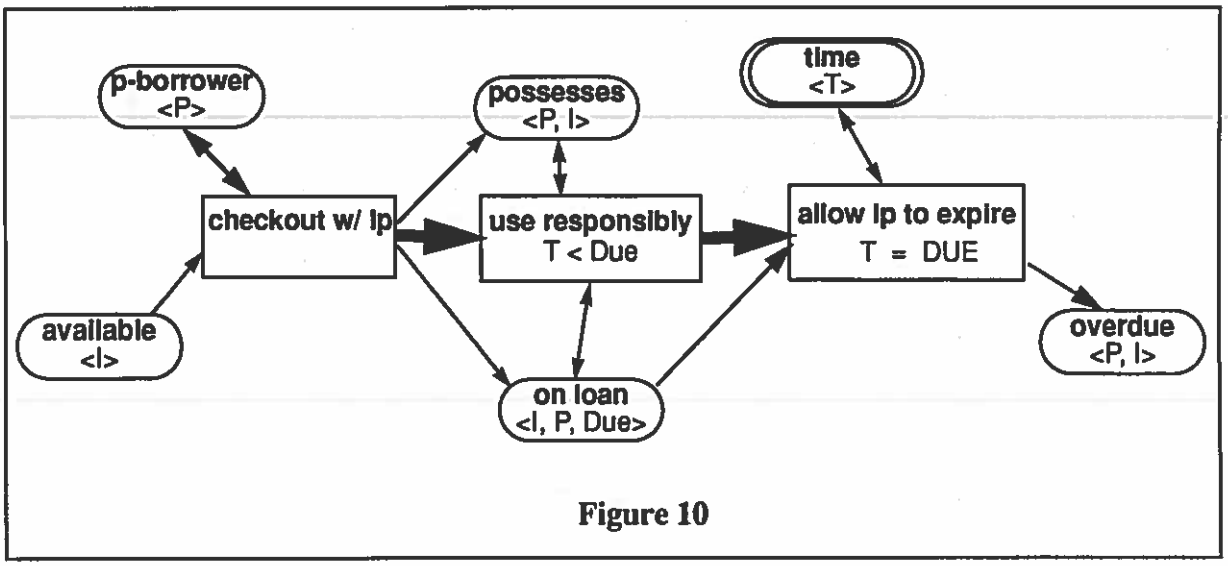


Figure 10

From this abstract scenario, the three concrete scenarios of figure 7 can be produced by specializing allow-item-to-become-overdue into each of its three siblings [Anderson&Farley 90]. In the pb-critic, this abstraction process is completely general: abstract operators may be children of yet more abstract operators. Abstraction is carried out by finding common relations *produced* by a set of operators and constructing a new abstract operator that contains the commonality [Anderson and Farley 88]. When a new concrete or abstract operator is added to the catalog, the abstraction hierarchy is automatically updated to "fit" the new operator in. We will see how the abstraction hierarchy can be put to use when we look at the second process in our methodology, that of modification.

Finally we note that abstract operators begin to give us one of the advantages of the cb-critic: avoiding rederivation of stock scenarios. However, we must add another piece to complete the picture. That piece is the cataloging of abstract scenarios/plans. Rather than rederiving the same abstract plan each time a library specification is being critiqued, the pb-critic is able to store abstract plans in the form of 'abstract macros' [Anderson&Farley 88]. For example, the plan for achieving an overdue item would be stored as shown in Figure 11.[1]

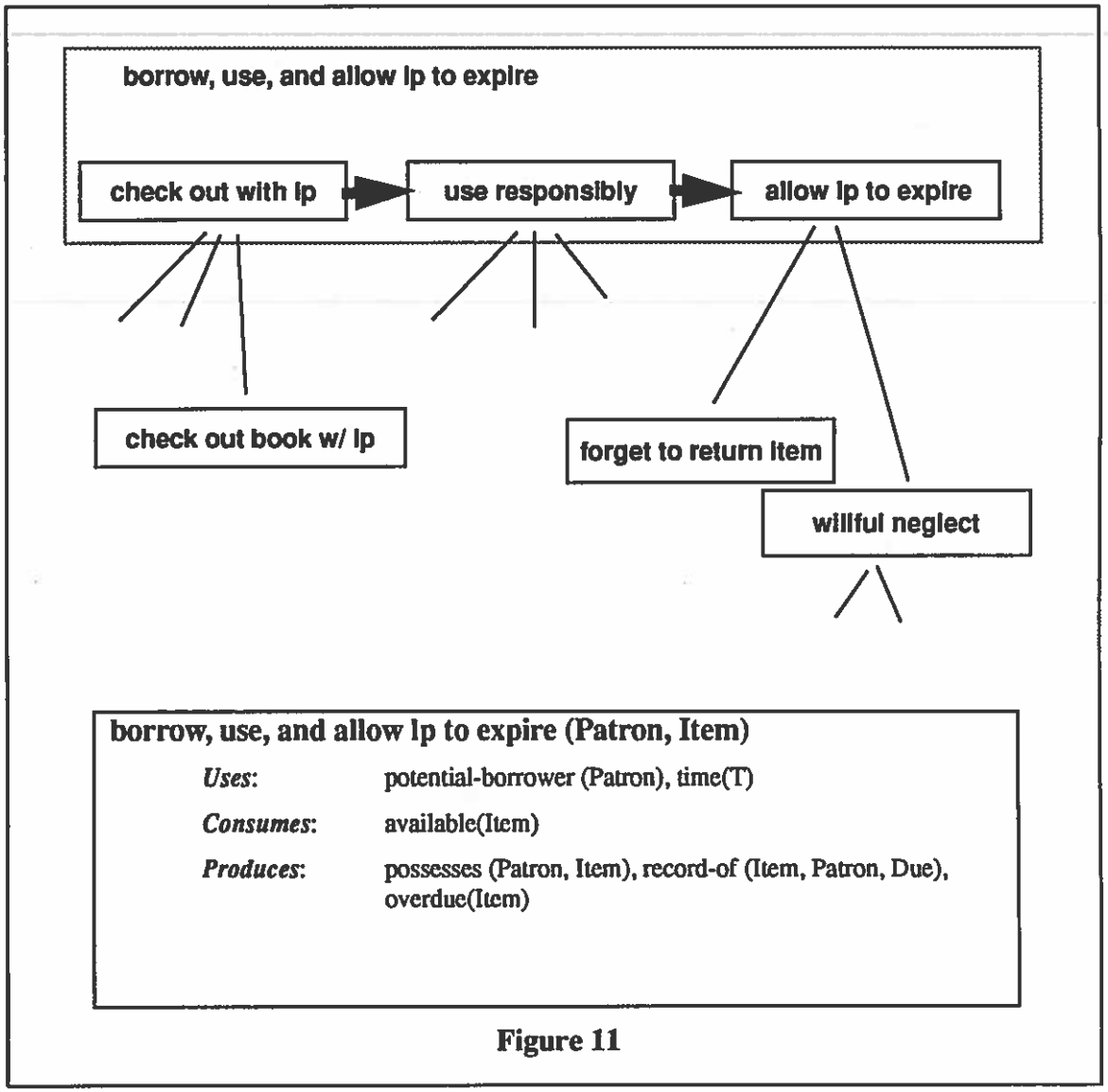


Figure 11

During subsequent critiquing sessions, this abstract operator will be retrieved whenever the pb-critic is asked to critique a resource management system in which avoiding overdue items is an issue. The operator will initially be retrieved as a single unit and then decomposed into its several steps. Each step can then be specialized to match the particular specification being critiqued.

1. In reality, the use-responsibly operator is abstract in the pb-critic as well. Its children are the various operators that represent more concrete uses of library material, e.g., refer to, browse, copy.

5.1.3 Interaction-based validation

The cb-critic and pb-critic validation tools assume a single set of goals and a single specification, or what we call a single perspective. We are exploring a third validation tool, called the interaction-based critic or ib-critic, that broadens its view to encompass multiple perspectives [Robinson 1990]. The ib-critic assumes that multiple agents may exist in the world, each with its own perspective on the final form of the system to be built. The ib-critic allows these agents to state their own goals and develop their own specification under those goals. Its role in validation is to analyze the separate specifications for consistency.

The ib-critic organizes goals into a hierarchy, using a modified version of goal/subgoal relation between a parent and its child. In particular, the notion of *partial* goal satisfaction is introduced; a goal may be satisfied to a degree by having some but not all of its children satisfied. In this view, children represent independent methods for achieving their parent goal. This is illustrated by considering a portion of the goal tree for the library domain, as given in figure 12.

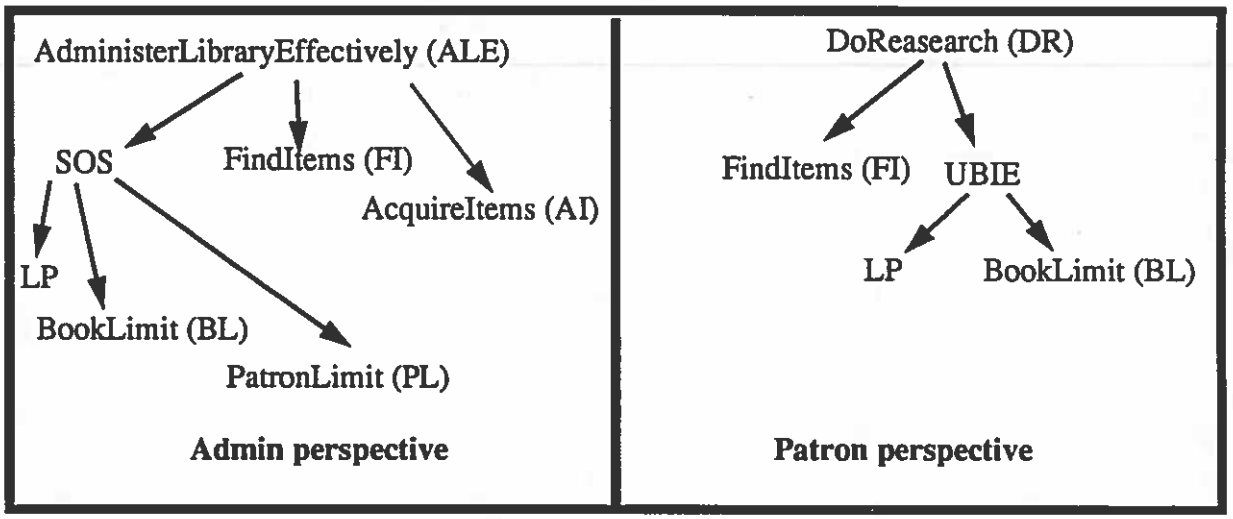


Figure 12

Under each non-leaf goal G in the tree, there exists a set of alternative sub-goals for achieving G. For instance, there are three means of keeping an adequate stock on the shelves (SOS): limit the loan period of an item (LP), limit the number of items the patron can borrow at any one time (BL), limit the total number of patrons that can borrow (PL). These three alternatives represent the *potential* means known to achieve SOS. To include SOS as a goal in the initial development state (i.e., as a member of $GOALS_0$ in figure 2), a client/agent must choose none, some or all of its alternatives as *actual* goals to be achieved, and to state to what degree they will satisfy SOS. In summary, the agent must order all combinations of alternative sub-goals for every non-leaf goal G in the tree. The leaf goals of the tree will be linked directly to the specification components which implement those goals.

An agent supplies ordering information for a goal G by defining a preference function PF_G. The domain of PF_G is an element of the power set of alternatives of G.The range of PF_G is a percentage value from 0 to 100. As an example, suppose the administration agent provides the following preference function for the SOS goal in figure 12:

$PF_{SOS}(\{LP, BL, PL\}) = 100\%$

$PF_{SOS}(\{LP, PL\}) = 50\%$

$PF_{SOS}(\{LP, BL\}) = 10\%$

$PF_{SOS}(\{BL, PL\}) = 10\%$

$PF_{SOS}(\{PL\}) = 5\%$

...

Using a slightly more illustrative notation:

$$\{ \{LP, BL, PL\}_{100\%}, \{LP, PL\}_{50\%}, [\{LP, BL\}, \{BL, PL\}]_{10\%}, \{PL\}_{5\%}, ... \}$$

In this case, the administration agent states that implementing a loan period, a borrowing limit, and a patron limit will satisfy the SOS goal completely. The second preference only satisfies the SOS goal to a 50% degree. Following alternatives quickly drop off from there.

Some goals may have an infinite number of scalar values as alternatives. For these type of goals we introduce ranges as part of the power set of the alternatives. For example, the alternatives of the loan period (LP) goal might be ordered as follows:

$$\{[0 .. 14]_{100\%}, [15 .. 45]_{50\%}, [46 .. \infty_{+}]_{0\%}\}$$

Here, the administration agent has specified that a loan period of 0 to 14 days is first preference and will completely satisfy the LP goal, a loan period of 15 to 45 days is second preference and will partially satisfy the LP goal, and that a loan period of 46 days or beyond is unacceptable in satisfying the LP goal.

To complete the loan period example, assume that the patron agent has defined the following preference for the LP goal from its perspective:

$$\{[180 .. \infty_{+}]_{100\%}, [30 .. 179]_{50\%}, [0 .. 29]_{0\%}\}$$

Assume further that the patron agent has given inclusion of the goal of utilizing borrowed items effectively (UBIE) a high preference. Likewise, the administration agent has given a high preference to SOS.

At this point we have the initial goal sets for both agents and begin to develop a specification from each perspective [Robinson 1987], [Bearman&Fickas 1988], [Robinson 1989]. The result is shown in figure 13.

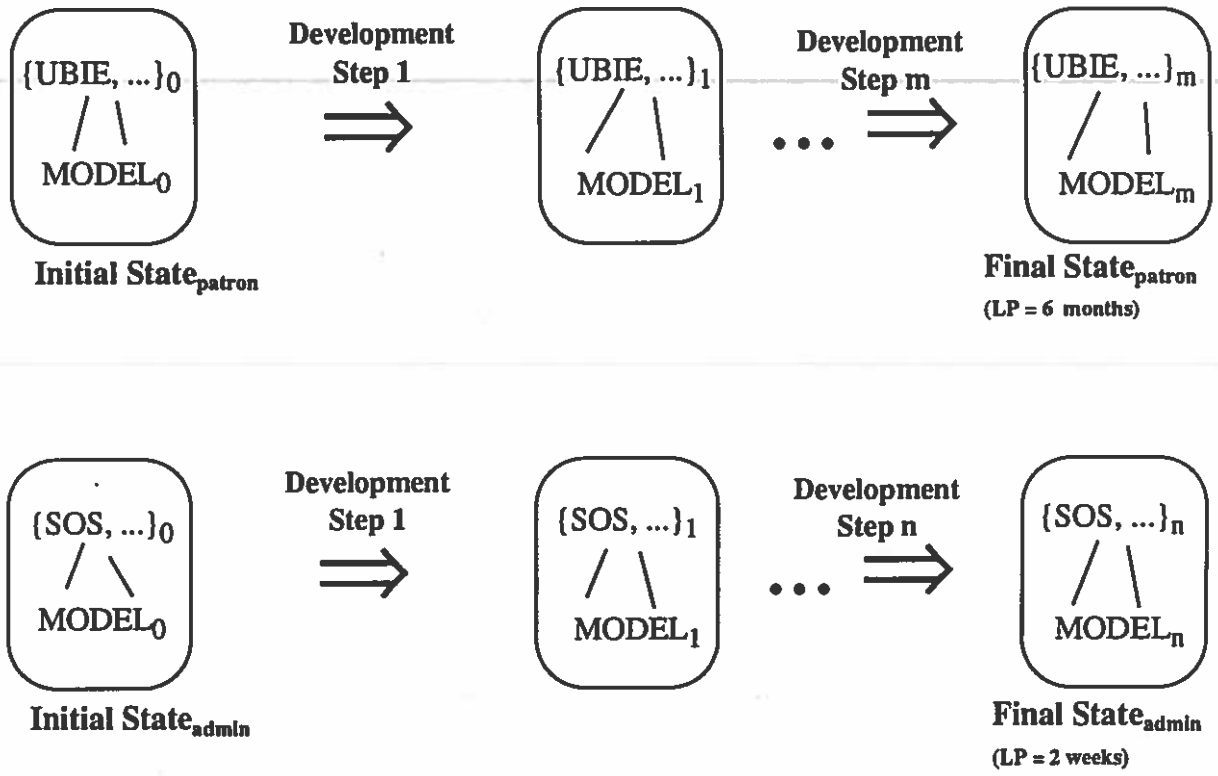


Figure 13

We are now ready to compare the two specifications, $model_m$ and $model_n$ for consistency. The ib-critic carries out analysis in three steps: 1) correspondences among specifications are identified, 2) conflicts between the specifications are detected, and 3) specification conflicts are reformulated into conflicts among goals.

Step 1: Correspondence Identification. Since two specifications have been constructed independently, we can expect that terminology is not consistent. The correspondence identification step determines specification equivalences. In our example, this equivalence mapping must be carried out between tokens, places, and transitions in $model_m$ and $model_n$. This is currently an interactive process: the ib-critic suggests equivalences based on name, type, and I/O equivalences; the human analyst uses this information as a guide to decide which components represent the same concept

Step 2: Conflict Detection. After correspondences are determined, the non-corresponding components are checked for conflicts. In our example, suppose the ib-critic notes a conflict between a loan period value of 14 days in $model_n$ and a loan period value of 180 days in $model_m$. This is a specification or feature conflict.

Step 3: Issue Formation. Next, *issues* are identified by tracing *incorporation links* from conflicting specification components to the leaves of the *ordered* goal hierarchy. In this case, both the 14 day and the 180 day values for loan period are traced directly to the loan period goals shown in figure 12. This ends the ib-critic's role in validation. It has found specification level conflicts among different perspectives, and has connected these conflicts into the leaves of the goal tree in each perspective. This information will be put to use during the modification process discussed in section 5.2.2.

5.2 Modification: adjustments in response to criticism

We have explored two general techniques for overcoming criticism encountered during the validation process: 1) modifying the model, and 2) modifying the goals. The first approach we will discuss addresses model modification, and the second approach both model and goal modification[1].

5.2.1 Plan-based modification

The input to the plan-based modifier (pb-modifier) is a scenario from the cb-critic or a plan from the pb-critic that demonstrates a goal violation, i.e., either an achievement goal is shown to be unachievable, or an avoidance goal is shown to be achievable. The following two model modification options are available in the pb-modifier:

1. *Redesign.* Change the set of actions provided by the model. The assumption here is that the development of a specification is a *design process*: it involves selection of a specific set of actions or functionality from a larger set of alternatives. If we catalog these alternatives in a formal way, we might be able to go back and reconsider them when we run into trouble with the set we have chosen [Alterman 89]. The pb-modifier provides such a catalog.
2. *Acquire.* Add new actions to the catalog (and hence, new alternatives to the model). The assumption here is that the catalog may be incomplete; attempting to choose other alternatives may fail if we do not have all the *plausible* alternatives represented. The key question is what is plausible. The pb-modifier uses a form of analogical reasoning to suggest plausible new operators.

We will look at each of these two techniques in turn for the loan period example. Looking back to figure 10, we will assume that the pb-modifier is given a critique for the avoidance of overdue items as an abstract scenario (figure 14):

Figure 14

We will choose to redesign the model to foil the above plan. To do so, we will have to consider removing or changing one or more of the three actions (henceforth, operators) used to gain a bad result. The pb-modifier divides operators into two classes, artifact and environment. The assump-

1. All of our tools allow *manual* modification of either the model or the goals, i.e., all have built-in model and goal editors. We focus here on tools that support the modification process more actively.

tion is that environment operators may be changed or removed, but only with great effort. In the pb-modifier, both the use-responsibly and allow-lp-to-expire operators are marked as (abstract) environment operators. Hence, we will look at them only as a last resort.

Our first naive attempt might be to *remove* the check-out-with-loan-period operator: if items can't be checked out, they can't become overdue. This foils the above plan, but introduces other goal failures having to do with giving users effective access to (i.e., possession of) material.

Our next alternative is to *replace* check-out-with-loan-period with another type of check-out operator, one that foils this plan, and at the same time does not introduce new goal failures. We can use the enabling conditions of the abstract operator allow-loan-period-to-expire as a focus:

Time (T) where T = Due or unenforced-contract (Item, Patron, Due).

If we can disable one of these, we can eliminate the plan as a counter-example. And by eliminating an *abstract* plan as a counter-example, we eliminate all of its more concrete manifestations!

We set our task as finding an alternative check-out operator that does not enable the allow-lp-to-expire operator. No candidate artifact operators have any effect on time, so we focus on the unenforced-contract relation. For this example assume that the pb-modifier has one alternative check-out operator to choose from, one with no loan period nor any record of borrowing. This operator is found by looking at the operator abstraction hierarchy (see section 5.1.2) for siblings of check-out-with-loan-period (see figure 15).

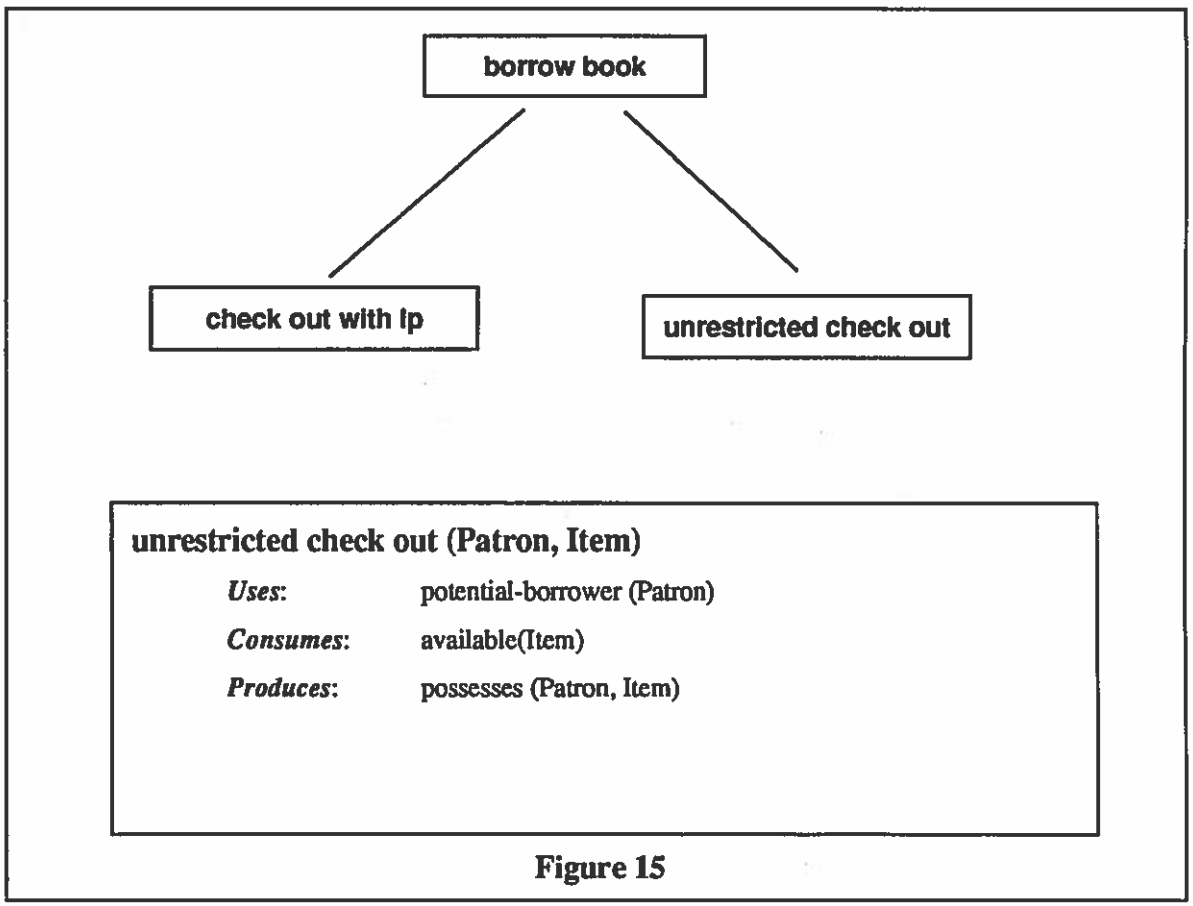


Figure 15

Replacing check-out-with-loan-period with unrestricted-check-out does foil the plan in figure 14. Because there is no due date, there is no possibility of a book becoming overdue. However, it also violates other goals related to keeping adequate stock on the shelves.

It appears that we will not be able to foil the abstract plan successfully. Our next approach will be to look at the more concrete versions of the plan by specializing the allow-item-to-become-overdue operator into each of its children. Using this divide-and-conquer strategy, we may be able to foil each specialized plan individually, i.e., find a means of preventing forgetting, find a means of preventing irresponsible use, find a means of preventing discarding.

Looking first at the allow-lp-to-expire operator (figure 9), we note that the more specific willful-neglect operator has a new pre-condition of unenforced-contract. If we can find a check-out operator that does not produce this relation, i.e., enforces the borrowing contract, then we can foil this abstract plan. For this example no such operator exists in the operator hierarchy for *library* systems. However, this may be because our model of libraries and their potential borrowing policies is incomplete. Assuming for the moment that this is the case, we will look to other borrowing applications for plausible additions to the pb-modifier's library check-out operators. That is, we will attempt to use analogical reasoning to find operators in a base domain to add to the target (library) domain. To do so, we will use the same mechanism we used to find alternatives earlier: we will look up the operator abstraction hierarchy. Figure 16 shows a piece of the operator class hierarchy that includes the borrowing domain of video rental. Looking at video rental check-out operators, we notice rent-video-with-enforced-contract. The effects of this operator are shown in figure 17.

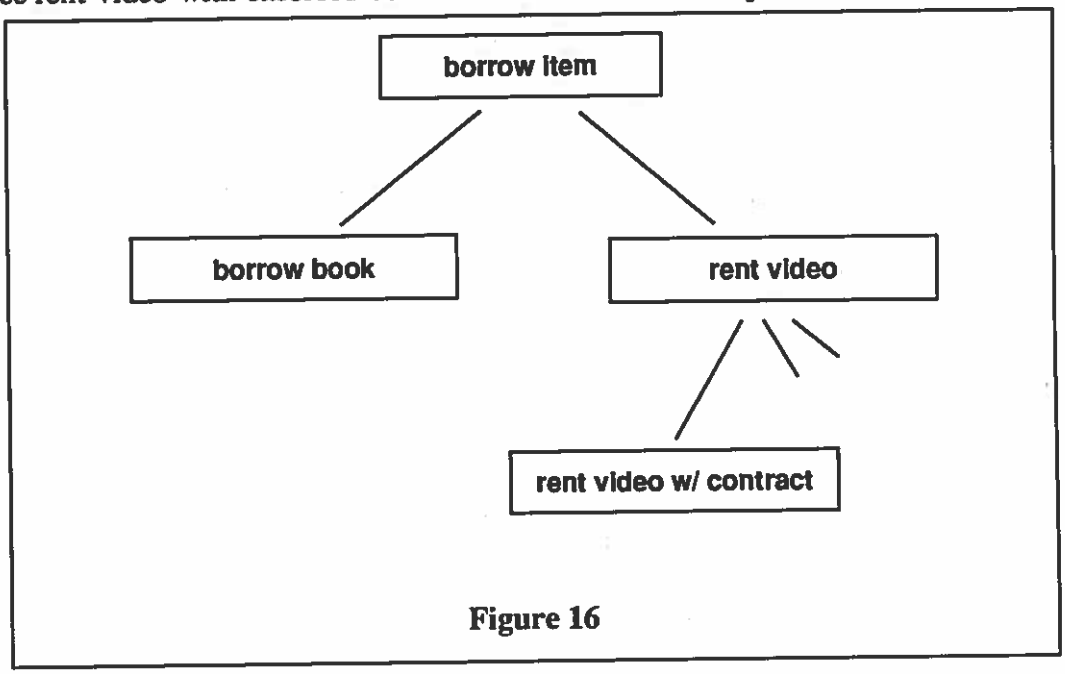


Figure 16

The operator produces a contract between the store and patron for the return of the video by the due date. The contract provides a way for the store to enforce responsible behavior on the part of the patron. According to our simplified model, patrons will not act irresponsibly if there is an enforced contract. This appears to be a plausible operator for the library domain, and hence we begin the process of transferring it.

rent video with enforced contract (Patron, Video)

Uses:	potential-borrower (Patron), time(T)
Consumes:	available(Video), has-money (Patron, $X)
Produces:	possesses (Patron, Video), has-money (Store, $X), on-loan (Video, Patron, Due), enforced-contract (Store, Patron, Video, Due)

Figure 17

Transfer consists of two parts: 1) substituting corresponding objects and 2) adding and removing relations. Currently, the pb-modifier does not automate either of these two steps. However, the first step, substituting objects, is supported by the abstraction hierarchy. For example, in the pb-modifier 'book' and 'video' both belong to the abstract class of borrowable items. Thus, 'book' can be substituted for 'video' wherever it appears in the operator definition.

Adding and removing relations from the produce, consume and use lists cannot be automated. A human user must edit the operator description to produce the new library operator shown in Figure 18. The relations dealing with money changing hands have been removed.

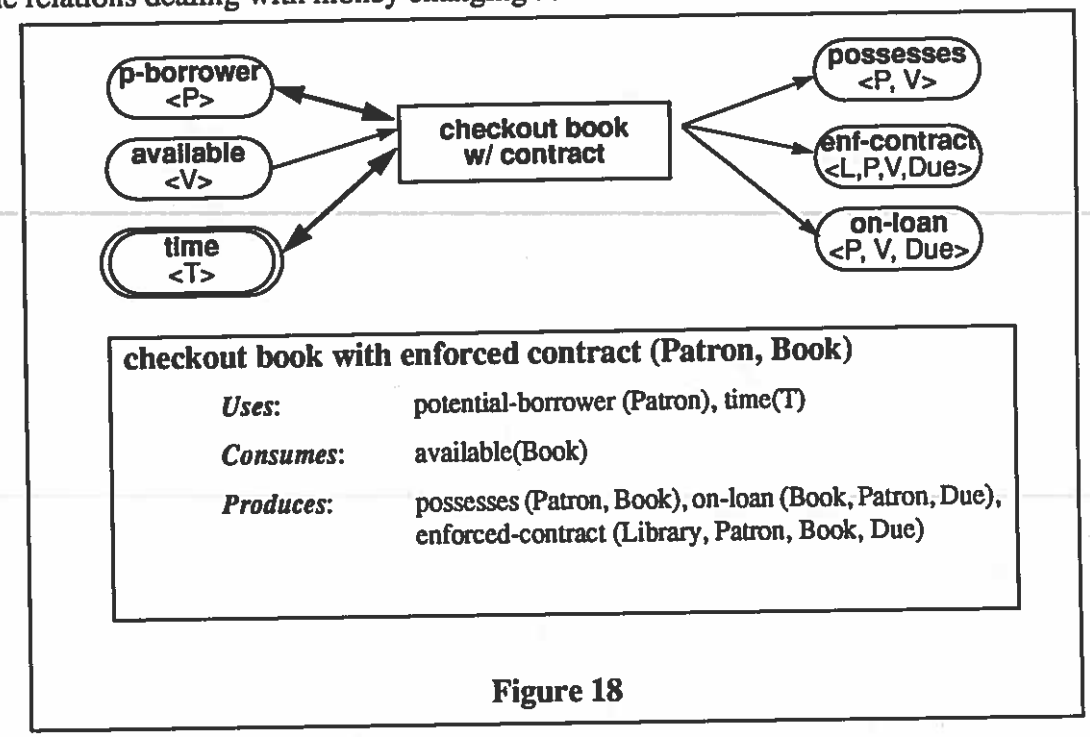


Figure 18

Once the new check-out-book operator has been formed, it is added to the operator hierarchy. Positioning a new item in the hierarchy is done automatically [Anderson&Farley 1988]. The new operator is compared to each operator in the hierarchy with at least one shared effect. For each comparison, the intersection of the two operator descriptions is extracted as the generalization of the pair. If an abstract operator exists which matches the generalization, the new operator becomes a child of the abstract operator. If no such abstract operator exists, a new abstract operator is created and is added to the hierarchy in the same way.

The generalization process is illustrated for the new operator 'checkout-book-with-enforced-contract'. When the new operator is compared to 'rent-video-with-enforced-contract' their generalization is 'checkout-item-with-enforced-contract' shown in figure 19.

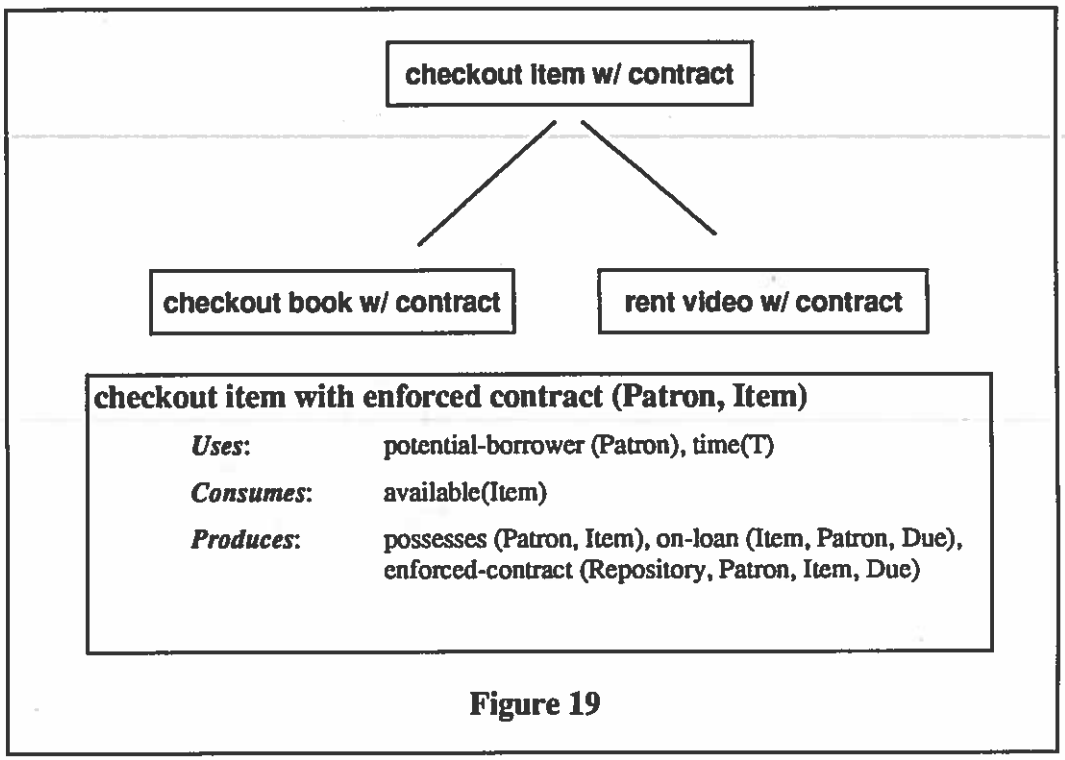


Figure 19

This abstract operator is also added to the hierarchy using recursive application of the pairwise comparison method. The final result is shown in Figure 20.

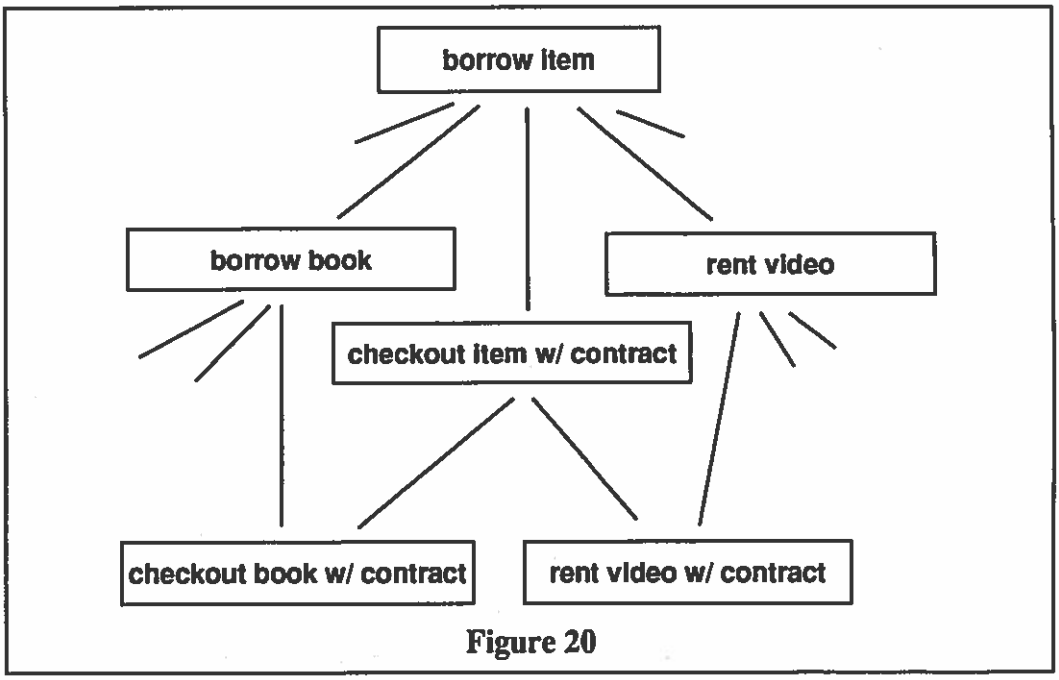


Figure 20

Once the knowledge base has been updated, we return to the problem of modifying the specification. The new checkout-book-with-enforced-contract operator is substituted for the check-out-with-loan-period operator and the new specification is sent back to the critic. Since there is no longer an unenforced contract, this operator foils the critic's plan for allowing the loan period to expire via willful neglect. At the same time, the goal of allowing patrons to use books responsibly is still achieved. Thus, we have solved a general class of problems, covering both irresponsible use and discarding the borrowed item, with a single modification.

Finally, the pb-modifier tackles the problem of allowing the loan period to expire due to forgetting. As before, the pb-modifier first tries to find an operator in the catalog to foil the plan. Since the preconditions of forgetting are the same as for the abstract allow-loan-period-to-expire operator, we know the search will fail. There is also no checkout operator in any other domain that can prevent a patron from forgetting.

In our example, part of the difficulty is that the forget operator has not been represented in sufficient detail. As seen in figure 21, adding detail to the forget operator requires changes to the definitions of other operators as well: the checkout operator now produces the relation 'knows (Patron, Contract)'. This relation is consumed by either the return operator or the forgets operator.

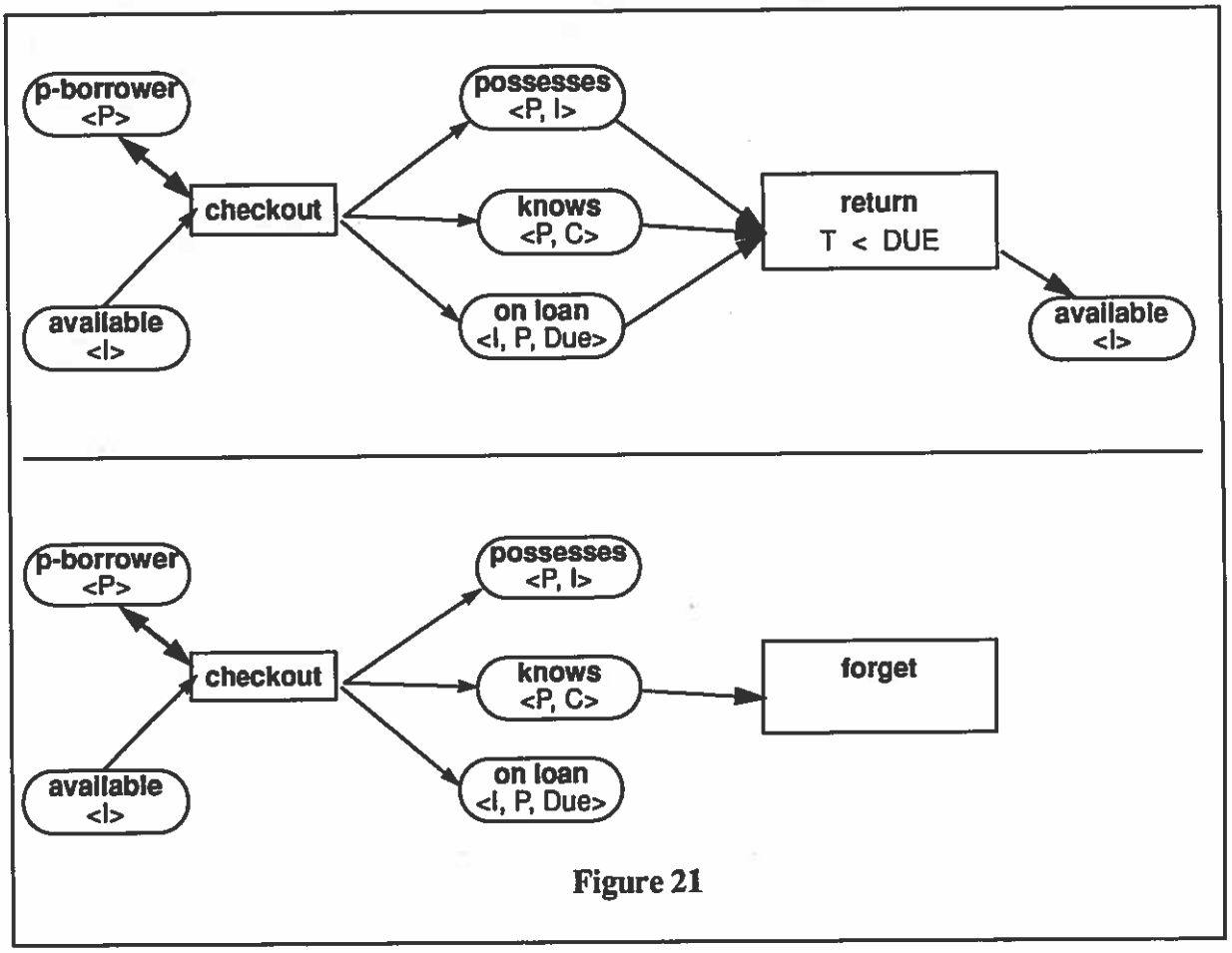


Figure 21

The problem of forgetting cannot be attacked using the previous methods. The only way to prevent forgetting is to keep the patron from knowing about the contract, but the patron must know in order to return the book. Thus, we cannot disable the forget operator. The next best alternative is to coun-

teract the effects of forgetting, through reminding. In more general terms, if an undesirable action cannot be prevented, the next alternative is to try to 1) minimize its occurrence, and 2) restore the desired conditions if it does occur.

The pb-modifier could be used to find a plan for achieving 'knows (Patron, Contract)'. The pb-modifier would suggest some form of notification, which could be adapted to the current problem. Notification might occur before the due date as a precaution or after the due date as a remedy.

Unfortunately, it is still possible for a patron to forget after a reminding, which means that there can never be an absolute guarantee that items will be returned on time. (Similarly, a patron can ignore even an enforced contract.) However, we are better off than we were before, in that the likelihood of overdue books has been reduced. The pb-modifier is not equipped to reason about varying degrees of likelihood, so we must turn to other tools. The use of qualitative modeling is one approach that we are exploring [Downing&Fickas 1990]. The nb-modifier discussed in the next section is another.

5.2.2 Negotiation-based modification

The ib-critic produces a critique that consists of specification conflicts among multiple perspectives and links from specification to goals for each perspective. We have developed a negotiation-based modification tool, the nb-modifier, that uses such a critique to modify either the goals, the model, or both. Negotiation aims to dissolve conflicts, and if that fails, to find "fair" compromises. The nb-modifier employs a two pronged strategy to the negotiation problem: 1) find a "ballpark" resolution using an analytic method based on multicriteria analysis [Zeleny 1982], and 2) improve the resolution by applying heuristic strategies [Robinson&Fickas 1990].

We will illustrate the nb-modifier on the loan period example. Assume that the nb-modifier is given the example conflict seen in section 5.1.3: a loan period of 14 days from the administration's perspective; a loan period of 180 days from the patron's perspective. Assume further that the ib-critic has linked these values to the LP goals as described in section 5.1.3. The nb-modifier has the following negotiation techniques at its disposal.

Analytic Compromise. When the first preferences of two perspectives cannot both be met, consider less preferable alternatives. With this approach, compromises are generated based on goal preferences using a weighted, multicriteria analysis. In the loan period example, using such an analysis would suggest a loan period of 37.5 days. While this is a simple split of the difference, more complex cases lead to a more complex resolution. In particular, the nb-modifier's multicriteria analysis algorithm's complexity turns on the number of specification components, specification conflicts, goals per conflict, and components per conflict. The loan period example is simple because is has few specification components, one conflict, one goal per conflict, and one component per conflict.

Search for Dissolution. When an acceptable analytic resolution can't be found, either because no preferences could be found in common, or because the compromise proposed by the pb-modifier was rejected by one or both of the parties, the pb-modifier will attempt to remove the cause of the conflict.

In the loan period example, one could subdivide the inputs of the loan process and distribute the loan periods over subclasses of items, i.e., text-books, reference-books, journals. The pb-modifier has a *Predicate Input* rule that suggests just this type of resolution. It is not specific to loan periods, nor borrowing systems in general. Instead, it uses means-ends analysis of the specification to construct a proposed refinement to the specification and goals [Robinson&Fickas 1990]. Applying this

rule to loan period, new components must be added to the specification (text books, reference books, journals) and new goals must be considered in the two perspectives (text-book-SOS, reference-book-SOS, journal-SOS, text-book-UBIE, reference-book-UBIE, journal-UBIE).

In summary, dissolution rules such as *Predicate Input* use strategies that uncouple conflict into smaller issues which may be individually resolved or resolved through their combination [Pruitt 1981].

Once the loan period problem has been subdivided, the nb-modifier reexamines the SOS versus UBIE conflict in a more refined light. For instance, the analyst may wish to use the nb-modifier to look anew for an analytic solution to the smaller issues. In the best outcome, all conflict will be dissolved analytically: patrons will be satisfied that keeping some material for longer periods and some material for shorter periods will satisfy the more refined view of UBIE; the administration will come to the same conclusion in regard to the refined view of SOS. More typically, analytic resolution of some sub-pieces will be possible with other sub-pieces remaining in conflict. The analyst may then ask the nb-modifier to reapply dissolution to these remaining sub-pieces, or move on to the following method.

Search for Compensation. When an acceptable means of dissolving remaining conflict cannot be found, compensate the parties whose goals have not been satisfied. This method increases the satisfaction of goals which indirectly contribute to the satisfaction of unsatisfied goals. For example, if the administration agent will not compromise on a value of 14 days, attempt to compensate the "losing" LP goal of the patron by increasing the satisfaction of that agent in more indirect ways, e.g., focusing on BL, a sibling of LP. If this fails, we might attempt to move further up the tree, looking at UBIE's parent and siblings.

The nb-modifier's compensation method views the goal tree in a causal and qualitative fashion: if we decrease our satisfaction of X, and Y is a sibling of X, can an increase in the satisfaction of Y counter-balance it? Along these lines, we have implemented a tool called qm-tool that attempts to answer such questions in qualitative modeling terms [Downing&Fickas 1990]. Like the nb-modifier's search of the goal tree, qm-tool will search a causal model for the most direct action that will compensate for an increase or decrease of satisfaction. However, it goes one step further: if it is unable to find an appropriate compensation action in the causal model it is given, it will locate the places in the model where a new component/goal could be added to bring about compensation. While the exact form the new goal should take is still very much up to the human analyst, pinpointing its location in a complex model is still a valuable service.

Case-based Reasoning. Each of the above three methods uses a form of search to find a satisfactory compromise. Further, this search is carried out anew for each new conflict in each new problem. In well studied domains, such as borrowing systems, conflicts are well known and their resolutions often documented. In these applications, it would appear wasteful to reinvent the same resolutions over and over.

The approach we are exploring in the nb-modifier is to 1) note when a new conflict is similar to one previously considered, and 2) use the old case to construct a resolution. We may wish to consider cases that have resulted in a satisfactory resolution, as well as those which resulted in an impasse: successful cases can be instantiated and modified for the current context; unsuccessful cases can prune analytic compromises, dissolutions, and compensations from consideration.

As an example, suppose the loan period conflict is eventually recast as that between SOS and UBIE. This is a fairly typical problem in borrowing systems: the administration agent needs to keep enough items in a repository for potential borrowers to choose from; the borrower agent needs to remove material, for as long as it needs, to complete a task. Suppose that we find, among others, a recall-on-demand case that is indexed on this type of SOS-UBIE conflict. The resolution suggested by the case is one that uses a mechanism of giving a borrower a large loan period, but with the proviso that material may be recalled after a much shorter loan period if another borrower has requested it. If this compromise is deemed acceptable by both parties, then it is left to the analyst, assisted by the nb-modifier to instantiate the case in the library domain and incorporate it into the final specification.

Use of the case-based approach here saves two types of effort:1) it avoids a complex combination of dissolution and analytic compromise methods needed to produce recall-on-demand from scratch, and 2) it directs the nb-modifier where to place the alternative in the goal trees of the patron and administration, i.e., under the UBIE and SOS goals, respectively.

We expect each of the negotiation methods of the nb-modifier to leave an explicit link to changes made in the specification. Thus, our 37.5 day loan period would be rationalized as an analytic compromise between the two LP goals. The need for sub-types of library material would be rationalized by the dissolution technique used to generate them. A large book limit (BL) would be rationalized as compensation to a patron for a short loan period. A recall-on-demand mechanism would be rationalized as a case-based solution that has worked in other systems. This type of negotiation record is crucial in understanding the final specification or the final delivered system. In applications like library systems, almost every component of the delivered system can be seen as a compromise among competing concerns. Attempting to modify the system (or the specification) without understanding what those compromises were and what they turned on is disastrous.

In the end, we return to the question we posed in section 2: isn't a loan period a simple specification problem? We'd answer by posing another two questions: Can you change it? If so, what are the ramifications of the change you make? Without a representation of the development process that lead to it, one that includes system goals, multiple perspectives, preferences, and compromises, we would argue that either question is impossible to answer reliably.

6. Discussion

We have proposed a two-cycle methodology for constructing closed specifications for complex domains. We further posited three issues that are key factors in any attempt to formalize or automate the methodology:

1. We must find a means to model the unmodelable, i.e., an environment that might include human behavior, behavior of complex hardware/software subsystems, behavior of natural processes.
2. We must recognize that clients specifying a problem should be given certain freedoms, freedoms that might be intolerable in the implementation process but that are necessary in the specification process.
3. We cannot pretend that there is only one perspective or set of goals for a system. Instead, we must represent 1) multiple perspectives, and 2) the process that allows them to come together in a single artifact.

The Kate tools we described in section 5 do not totally solve these problems, either individually or collectively. However, as we discuss below, they do act to further refine the solution space.

6.1 Modeling complex domains

We have presented two approaches to approximating complex domains, one that addresses a model of behavior and one that addresses a representation of goals and preferences. We will take up each in turn.

6.1.1 Approximating system behavior

We may never be able to fully model the myriad ways patrons, administration and staff can interact, either responsibly, irresponsibly, or errorfully, with a library and its material. However, it is not clear that we have to. In well studied applications, known cases or scenarios can replace a first-order theory of the domain. In some sense, we pick out the features of environment behavior that we wish to address in our system, and ignore the rest. The cb-critic embodies this case-based style of modeling.

The cb-critic's case-base approach works well for routine specification problems in well known domains with standard terminology. Outside of these limits, fragileness increases. Thus, if a client proposes an exotic component, it is likely that matching will fail - while each case is represented as an *abstract* sub-net, it is nevertheless meant to be a *typical* abstraction, one that will never cover all of the specifications possible.We have some interest in a more sophisticated matching algorithm that might be able to perform transformations on a case pattern to "jitter" it into matching a specification, thus increasing the number of specifications matchable from a single case. However, we have yet to attempt a transfer of earlier jittering techniques to the critic's matcher problem (a discussion of goal-directed, automated jittering techniques that is the source of our optimism in this approach is presented in [Fickas 1985]).

If a client wishes to apply the critic to an area where little domain analysis has been carried out, then finding agreement on the key goals and cases to represent is problematic. There have been suggestions that a case-base like that of the cb-critic might be acquired by various learning techniques, e.g., example-based generalization. However, such approaches require at least a partial theory of the domain, something currently lacking in the cb-critic in any usable form for learning.

The plan-based approach of the pb-critic and pb-modifier also addresses problems with modeling complex behavior. It provides a type of approximation through abstraction. In essence, it allows us to represent and use abstract operators in exactly the same way that we would represent and use the concrete operators that form the first-order theory of the domain. Lacking such a theory, the abstract operators can be viewed as types of active place-holders that allow both validation and modification processes to continue.

The problem with the plan-based approach is that, at the same time, it is too formal and too weak. It relies on a standard planning representation that views goals as universally quantified state descriptions. While this promotes general, automated tools, it also rules out representation of non-operational goals/requirements, e.g., keep cost below some threshold, maximize reliability. Both the case-based and negotiation-based approaches we have discussed allow non-operational goals to be stated as long as they can eventually be cast in operational terms. However, the reasoning techniques they employ are more ad hoc and difficult to automate and extend.

6.1.2 Approximating system goals

The ib-critic and nb-modifier address the representation of the often conflicting, continually shifting, partially satisfiable goals of multiple interest groups or perspectives. Clearly this is not an area where we can expect a first-order theory soon. Instead, these two tools rely on a combination of user assistance and approximation techniques to navigate in this space. There is no denying that the approximation issues faced in building the two tools have been difficult:

We must represent all known methods. The construction of a goal decomposition hierarchy under this constraint is particularly difficult. A child is one subgoal of the various subgoal decompositions of its parent. We attempt to determine a set of subgoals which both (1) covers the various means of satisfying a goal and (2) contains relatively independent subgoals. Coverage is obtained through experience, but is limited by ever changing technologies. Independence is not easily controlled, as it is a matter of the problem solving methods and technology.

We must reduce preferences to a single, linear scale. To carry out multicriteria analysis, we must compare apples to apples, i.e., all preferences must be mapped to a single scale. Furthermore, to be effectively computable, the algorithm is restricted to a linear scale. One might question the simplifications necessary to carry out analytic resolution, or even if it is a useful technique, in general. Speaking from the decision science field, Pruitt shows that there is real benefit in exchanging numerically weighted preferences [Pruitt 1981]. He notes in the same paper that more sophisticated types of goal representation are "theoretically capable of transmitting rich information from which resolution formulas can be devised. However our data suggest that it has limitations ...".

We must model goal interaction. Goals may interact in a complex fashion, and eventually we will have to reason about this interaction during conflict negotiation (see, for instance, the compensation method in the nb-modifier). There are two approaches to the problem. The first, embodied by the nb-modifier, attempts to form a domain-independent theory of goal interaction and provide general reasoning methods based on the theory. Thus, the nb-modifier represents goal interaction links such as directly-interferes-with and indirectly-supports. Conklin has suggested still other possibilities [Conklin 1989]. An expert human analyst is expected to supply actual links between goals for any particular domain. These links, then, represent a form of compiled knowledge of the human analyst; they are given as expert knowledge of the domain with no further rationalization.

The second alternative is to integrate goals with a causal model of the domain. Using this approach, systems like qm-tool (see section 5.2.2), XPLAIN [Neches et al 1985], and Neomycin [Clancey 1985] back up compiled knowledge with a more principled model of the domain. For qm-tool in particular, goals are linked through a qualitative model (see [Sycara 1988] for a similar approach). While this makes reasoning more difficult than in a tool like nb-modifier that explicitly links goals -- qm-tool requires one to trace goal interaction through a qualitative model -- it has the advantages of reasoning from a non-compiled, principled (albeit abstract) model of the domain.

As a final note on our use of goals, in observing clients interact with an analyst we noticed an interesting phenomenon: without some concrete grounding of their actions, clients navigation of the goal/requirements space was shaky at best. They often had only a vague notion of what stating a goal would buy them, and were quite willing to make changes to the goals (as opposed to the model) under criticism. A remedy to this problem appears to be, in part, the generation of a subset of representative behaviors that reify a goal (another big part would be an estimation of the cost in-

volved in achieving the goal).Without some touch with reality, one can expect a client to move around the goal space with the same abandon shown by politicians concocting line items in a multi-billion dollar budget.

In summary, we must eventually marry a scenario generator like that found in the qm-tool or cb-critic to a tool like the nb-modifier. That is, a client should be able to visualize, directly or indirectly, the ramifications of taking a stand on a goal, stating preferences, and agreeing to compromises. Taking a software engineering view, this might be seen as a type of goal-directed symbolic evaluation or rapid prototyping.

6.2 Requirement freedoms

Our interest is in representing and reasoning about development states that include incomplete, inconsistent, or ambiguous components. The following table illustrates the cross product of the three approaches and freedoms we have presented. We discuss each of the columns below.

	incompleteness	**inconsistency**	**ambiguity**
case-based	assumes all goals known can fill in incomplete model from cases	indirect conflicts point-counterpoint scenarios	not supported
plan-based	assumes all goals known can use analogy to suggest new operators	indirect & direct conflicts point-counterpoint scenarios	can reason with abstract operators & concepts
negotiation-based	assumes all goals & methods known	indirect & direct conflicts resolution methods	preferences & partial satisfaction

Incompleteness. Each of the approaches under the incompleteness column makes an important assumption: any goal that a client may wish to state is known and predefined in the tool; the analyst's job is choosing what subset of goals should be included for any specific problem. In other words, we have focused on reasoning about incomplete models but not incomplete goal sets. Our interest in incomplete models has been twofold:

1. We add new components to a closed specification that has been proposed by the analyst when it is viewed as too naive by the tool. This type of addition is seen in the discussion of the cb-critic in section 5.1.1.
2. We add new components to the known methods of the tool itself to allow reasoning to continue. This type of addition is seen in the discussion of the pb-modifier in section 5.2.1.

In both cases, adding new components has a distinctive deficiency-driven flavor: when a deficiency is found in either model or tool as part of a larger reasoning process (validation, modification), add whatever components are necessary to overcome the problem. This can be contrasted with approaches that center on domain analysis or requirements elicitation as separate, stand-alone processes, things one does to fill out a tool's representation in preparation to beginning specification construction [Dardenne&vanLamsweerde 1990], [Dubisy&vanLamsweerde 1990]. The following rough analogy can be drawn between the two approaches.

- Deficiency-driven acquisition is goal-directed - pieces are filled in only when needed to achieve a validation or modification task.
- Domain elicitation is data-directed - initial pieces are filled in by stock questions, analogical reasoning, or even free form input from a domain expert. From there, implications are followed to derive further pieces of the domain representation.

We can only conjecture that some combination of the two will be necessary; to our knowledge, no tool has been proposed that integrates both techniques.

Inconsistency. Our major effort to date on inconsistency has been on negotiation-based tools. This is a difficult area indeed. On the one hand, we believe that negotiation among interested parties is intrinsic to the specification process. If we really want to capture useful rationales for the systems we build, we will have to represent the negotiation process. On the other hand, successful negotiation often appears to be a decidedly human activity, one based on intuition, discovery of hidden agendas, politics, and human behavior in conflict settings. We have not attempted to represent any of these explicitly. The question is, do we need to? Or can we rely on an expert human analyst/negotiator to carry out the social aspects of negotiation, using tools like the ib-critic and nb-modifier as fancy negotiation-based pocket calculators? If so, is the formal rationale produced by these tools enough? Or will some representation of the social aspect of the final compromise be necessary? All of these questions remain open.

Ambiguity. We have studied two basic approaches to reasoning under ambiguity:

1. We use abstraction as a type of *useful* ambiguity. Both the pb-critic and pb-modifier illustrate this in terms of model abstraction. The goal tree of the negotiation-based tools allows abstract goals to be stated that are, like the abstract operators of the plan-based tools, types of place-holders for less ambiguous goals. While they must eventually be made non-ambiguous by their connection to actual specification components, they can still serve a useful purpose in their abstract state (see section 5.2.2).
2. We use ordered preferences to show a natural fuzziness in goal satisfaction. The representation of such ambiguity is critical to successful negotiation in tools like the nb-modifier.

We also note that the qm-tool can be viewed in much the same light as the pb-critic and nb-modifier - it uses a form of useful ambiguity, called qualitative modeling, to reason about abstract models.

6.3 Multiple perspectives

The ib-critic and nb-modifier tools support multiple perspectives. However, their focus is on the negotiation process. Hence, their representation is directed towards concepts that will aid negotiation: goal hierarchies, goal preferences, incorporation links, compromise techniques.

In a related effort, we have begun to explore the larger issue of representing the means various perspectives (or what we now call agents) are formed, how responsibility is assigned to agents, how agents cooperate to achieve system goals, what mechanisms we must build into a system to counter irresponsible or errorful behavior by agents. After [Feather 1987], we call the problem one of *composite system specification.*

One can view the composite system specification problem from several angles:

- It is a problem of joint problem solving. There are system-wide goals, and the environment and artifact agents must cooperate to achieve them. In this view, techniques from Distributed AI appear applicable.
- It is an interface problem. We must devise an interface for the artifact that allows it to communicate with the environment agents to solve the goals. From this view, techniques from Human-Computer Interaction appear applicable.
- It is a reliability problem. If we make an agent responsible for achieving a goal, we must worry about the reliability of that agent carrying out the responsibility. We may need to add back-up or reliability-enhancement mechanisms. Work in fault-tolerant systems and software safety appear relevant.
- It is a motivation problem. If we make an agent responsible for achieving a goal, we must worry about the motivation of the agent. The agent may have other conflicting goals that will interfere with its responsibility. We may need to add carrot or stick incentives to encourage the agent to choose the good of the whole over its own selfish needs. We see tools like the ib-critic and nb-modifier as applicable here, both in detecting goal conflicts and remedying them.

Other requirements and specification projects have begun to address single pieces of the composite system problem in isolation [Dubois&Hagelstein 1987], [Finkelstein et al 1990]. We have embarked, under the Kate project at Oregon and the KBSA project at Information Sciences Institute, on a broader approach that attempts to integrate the four views above into a single representation. Our efforts to extend the pb-critic and pb-modifier tools to the composite system specification problem is presented in [Doerry et al 1990].

6.4 Tool implementation

We have presented six Kate tools in this paper. The table below summarizes their implementation status.

	automatic	semi-automatic	manual
cb-critic	case retrieval scenario generation	case matching	spec modification
pb-critic	plan generation		
ib-critic	issue formation	spec matching conflict detection	
pb-modifier	goal-directed op retrieval	op specialization op replacement op acquisition	op transfer spec modification
nb-modifier	multicriteria analysis	compensation dissolution	case-based resolution spec modification
qm-tool	scenario generation	op acquisition	spec modification

We have not included entries for control, i.e., the selection of development steps in figure 2. All of our tools are assistant-based, relying on the human analyst to guide the development process. We do not expect this to change in the near future. As an example of the complexity of the control problem, we can turn to section 5.2.2. There, we simply took each of the nb-modifier's four methods in sequential order. However, dissolution, compensation, and case-based resolution cause new issues to be considered. All four methods of resolution can be applied to the new issues; hence, the control of these four methods is necessarily intertwined. While expert human negotiators are able to provide such intertwined control effectively [Raiffa 82], [Pruitt 81], there is no indication that we are close to having a corresponding computer-based model.

Acknowledgments

We thank current and past members of the Kate project -- Eck Doerry, 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 1988] Anderson, J., Farley, A., Plan abstraction based on operator generalization, in *Proceedings of AAAI-88*, St. Paul, 100-104, 1988

[Anderson&Farley 1990] Anderson, J., Farley, A., Partial Commitment in Planning, TR 90-11, Computer Science Dept., University of Oregon, 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

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

[Clancey 1985] Clancey, W., Extensions to rules for explanation and tutoring, *Rule-Based Expert Systems*, Buchanan and Shortliffe (eds), Addison-Wesley, 1985

[Conklin 1989] Conklin, J., Interissue dependencies in gIBIS, Technical Report, MCC Software Technology Program, P.O. Box 200195, Austin, Texas, 78720

[Dardenne&vanLamsweerde 1990] Dardenne A., van Lamsweerde A., Towards Concept Acquisition in Requirements Elicitation, Technical Report N. 12, KAOS Project, Institut d'Informatique, rue Grandgagnage 21, B-5000 Namur, Belgium, 1990.

[Doerry et al 1990] Doerry, E., Fickas, S., Helm, R., Feather, M., Deriving interface requirements through composite system design, Technical Report 90-10, Computer Science Department, University of Oregon, Eugene, OR. 97403

[Downing&Fickas 1990] Downing, K., Fickas, S., Specification criticism via policy-directed envisionment, Technical Report 90-11, Computer Science Department, University of Oregon, Eugene, OR. 97403

[Dubisy&vanLamsweerde 1990] Dubisy F., van Lamsweerde A., Requirements Acquisition by Analogy, Technical Report N. 13, KAOS Project, Institut d'Informatique, rue Grandgagnage 21, B-5000 Namur, Belgium, 1990.

[Dubois&Hagelstein 1987] Dubois E., Hagelstein J., Reasoning on Formal Requirements: A Lift Control System, In *Proceedings of the 4th International Workshop on Software Specification and Design*, pp. 161-167, 1987.

[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

[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

[Finkelstein et al 1990] Finkelstein, A., Goedicke, M., Kramer, J., Viewpoint oriented software development, Technical Report, Computer Science Department, Imperial College of Science and Technology, London, 1990

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

[Meyer 1985] Meyer, B., On formalism in specifications, *IEEE Software*, Volume 2, Number 1, January, 1985

[Neches et al 1985] Neches, R., Swartout, W., Moore, J., Enhanced maintenance and explanation of expert systems through explicit models of their development, In *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, Nov. 1985

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

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

[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

[Rubenstein&Waters 1989] Rubenstein, H., Waters, D., The requirements apprentice, In *Fifth International Workshop on Software Specification and Design*, Pittsburgh, 1989.

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

[Wing 1988] Wing, J., A study of 12 specifications of the library problem, *IEEE Software*, July, 1988

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