Being Suspicious: Critiquing Problem Specifications

Stephen Fickas and P. Nagarajan

Computer Science Department
University of Oregon
Eugene, OR. 97403

Abstract

One should look closely at problem specifications before attempting solutions: we may find that the specifier has only a vague or even erroneous notion of what is required, that the solution of a more general or more specific problem may be of more use, or simply that the problem as given is misstated. Using software development as an example, we present a knowledge-based system for critiquing one form of problem specification, that of a formal software specification.

1 Introduction

Suppose one were given a problem P to solve. Suppose further that it was known that generating a solution S for P will require a large effort. The question we ask in this paper is a pragmatic one: should we take P on blind faith and forge ahead, or should we scrutinize P carefully before committing resources to its solution? We will argue for the latter approach. More specifically, we propose that three types of specification critiques can account for a useful and interesting set of specification errors:

Unsupported policy: a domain goal or policy that we wish the system to obey is not supported by any specification component.

Obstructed policy: a domain goal or policy that we wish the system to obey is actively obstructed by a specification component.

Superfluous component: a specification component can be seen to support no domain goal or policy of importance.

To investigate this hypothesis, we have built a computer-based system that, given a problem specification in a specific domain, will view the specification as suspect until it can be rationalized, using the three criteria above, against a set of domain policies. In this paper we describe this system, and report on our efforts to evaluate it on a standard specification problem.

2 Towards a specification critic

The characterization of P, for our project, is that of a formal software specification. In earlier work, we also studied formal and automated means of mapping specifications to implementations, i.e., the solution space S [Fickas, 1985]. While our project is concerned with *software* specification techniques, we suggest that the approach we propose here might find application in any domain where problem specification is difficult (because of complexity, ambiguity, ignorance) and solution techniques are costly.

Our interest in a specification critic (henceforth, we will use *problem specification* and *specification* synonymously) is one part of a larger project whose goal is to provide assistance to a software analyst in producing a formal specification. This project, called Kate [Fickas, 1987], rests on the following 3 components:

1. A model of the domain of interest. This includes the common objects, operations, and constraints of the domain, as well as information on how they meet the types of goals or policies one encounters in the domain.

2. A specification construction component that controls the design of the emerging specification.

3. A critic that attempts to poke holes in the client's[1] problem specification.

Our focus in this paper is on the first and third components, the domain model and the specification critic (see [Swartout 1983] and [Yue 1988] for a complimentary, domain independent approach to specification analysis).

2.1 Basic critic components

The critic consists of a **model** part, an **example** part, and correspondence links between components in **model** and **example**. The use of **example** is as the representation of a specification under review. Our problem description language, used by both **model** and **example**, can be viewed as equivalent to a Petri-net in its support of places, tokens, transitions, and non-deterministic control. However, it also extends the basic Petri-net model in the following ways: it supports token objects, token types and token abstraction through a class hierarchy similar to Greenspan's RML language [Greenspan, 1984]; it introduces the notion of a place type with capacity [Wilbur-

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

[1] We will use the singular form of client as a useful simplification in this paper. In reality, there are often many "clients" to satisfy.

Model cases are linked to policies. Each such link can take on one of two values: *positive* - the case supports the policy; *negative* - the case obstructs the policy. Figure 1 shows a small portion of the resource management **model** with policies denoted by square boxes, domain cases denoted by rounded boxes, and policy-to-case links as highlighted arcs: negative arcs end with a black circle, positive arcs are drawn normally. Arcs between domain cases are taxonomic.

Note that figure 1 presents a static view of **model**; when used in a critique, correspondence links would exist between cases and **example** components. Further, each policy would be marked with a value *important*, *unimportant*, or *unknown*. Thus, the final step the user must carry out to run the critic is to give some or all of the policies a value; unmarked policies are given a value *unknown*, which in turn is conservatively viewed as *important* by the system.

By choosing various policy values, we can "take the view" of various system users. For instance, if we take the (selfish) view of a user of the library, we might mark a large selection set and unlimited borrowing as important, and all other policies as unimportant. We can later change policy values to reflect the good of the whole (i.e., take the library administration's view by marking privacy and minimization of cost as important) and rerun the critic.

2.2 Critic execution

There are three types of problems that are of interest to our critic:

> **Non-support**: A policy marked as *important* (or *unknown*) is linked *positively* to a **model** case. A case match is not found in **example**.
>
> **Obstruction**: A policy marked as *important* (or *unknown*) is linked *negatively* to a **model** case. A case match is found in **example**.
>
> **Superfluousness**: A policy marked as *unimportant* is linked *positively* to a **model** case. A corresponding case match is found in **example**.

The third critique is based on the notion that components added to a specification in support of an unimportant policy will tend to add unnecessarily both to the complexity of the system, and to the cost of its operation and maintenance.

We note that it is possible, and in fact typical, for the same case to have both positive and negative links to some set of policies. That is, the case may positively support policy P1 and negatively affect policy P2. Taking an example from figure 1, P1 is *give user's a good stock on hand*, P2 is *give users a useful working set*, and the case is *force turnaround* (or *restrict borrowing*). While we would like to think that P1 and P2 are never both simultaneously marked as important, it is not untypical for a client to describe a conflicting set of goals or policies. In fact, most borrowing systems can be viewed almost solely as compromises between conflicting concerns. One key process in specification design is then coming to grips with conflicting policies through various forms of trade-off and compromise. While some of our other work on Kate has begun to explore this area [Fickas, 1987], this version of the critic does not attempt to temper its criticism by looking beyond single links. On the other hand, it does allow policy values to be changed and a critique to be rerun.

There are three forms of output from the critic. The first is a parameterized version of canned text, e.g., "policy P1 is marked as important and is not supported in example", "policy P2 is marked as important and is obstructed by component C in example".

Second, and more importantly, we have begun to explore the use of simulation to back up a critique. Our long term goal is to be able to provide the rich and seemingly inexhaustible example generation (in the form of verbal simulations) seen in protocols of human analysts attempting to back up a point [Fickas *et al*, 1987]. As a start, we have built a simulation component that 1) allows each of the critic's cases to include one or more scenarios for demonstrating that case dynamically, and 2) uses the scenarios to animate the corresponding portions of **example**, in the form of transitions firing and tokens being moved around the net. Each scenario includes an initial marking of the relevant sub-net in **example**, and constraints on the non-deterministic control to force exemplary paths to be taken. The initial marking data may be either a) abstract -- if the case pattern uses abstract objects, we can use instances of the same abstract objects in setting up an example run -- or b) concrete -- we may decide to use a refinement of the case objects, e.g., Mary Smith checking out *The Life of a Gene* and keeping it 3 years beyond its due date. In this way, the scenario set for any case C may contain a mixture of abstract and concrete examples of C. The system runs a case's scenarios, one by one and in ordered fashion, under user direction.

Figure 2 shows a snapshot of a run-on-a-depository scenario being simulated using its matching **example** components (shown in parentheses). The corresponding case, that of a resource underflow condition, matches on any unconstrained check out action (unpredicated transitions are represented by vertical lines) where human borrowers and physical resources are involved. As an initial marking for this scenario, we set up a small number of resources and a larger number of resource users. The simulation will be abstract in the sense that tokens represent any physical resource (any type of library resource in this case) and any human borrower (staff, faculty, students in this case). It is concrete in that each token represents a single, physical resource or single human user (as opposed to, say, an information resource, a mechanical resource-consumer, or aggregates of each).

The other key piece of information necessary to make this scenario work is a constraint on non-deterministic control, one that will run the check out action continuously until no resources are left (but demand still exists). In other words, to give the worst case view we will ignore other processes that may exist to replenish the resource stock (e.g., buy more, force check in) or lessen demand

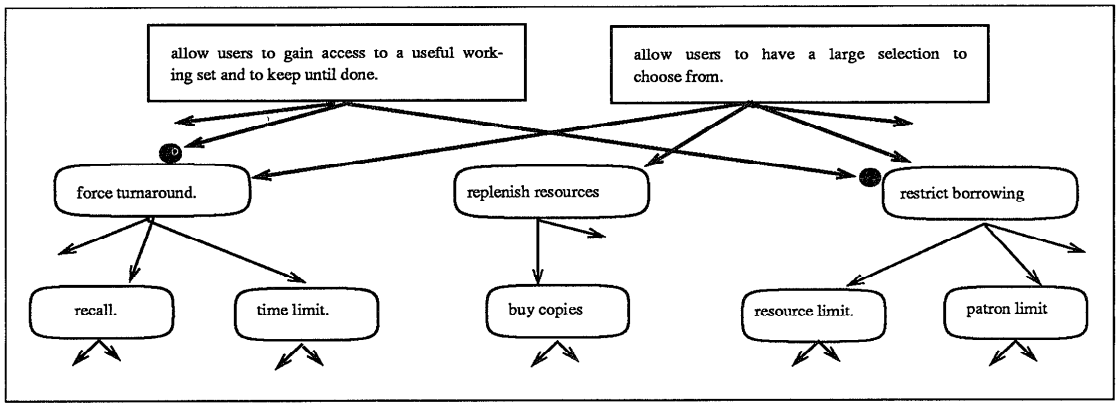

Figure 1: Portion of model

(e.g., remove borrowers) as long as the check out transition is enabled.

The third and final type of output the critic can supply is in a more positive form. In particular, the user can ask to see all **example** components *supporting* a particular policy, or given a specific component in **example**, he or she can ask to see the cases that have been matched using the component, and the policies those cases support. When integrated with a specification editor, this has shown to be a useful tool in determining the ramifications of specification changes in terms of the overall goals of the system.

3 Evaluation

We have run the critic on the problem description in the Appendix from various points of view, e.g., with policy values reflecting selfish users, with policy values reflecting the good of the whole. In this section, we will discuss a critique of our best reconstruction of the implicit policies of the problem after looking at the origins of the library example and talking to the authors of the version used in [FIWSSD 1987]. We set the policies 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 various 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 policy information. With this in mind, we give several representative findings of our critic:

1. The query actions in L5 and L6 are found to be *obstructive*; they may be used to give out user-confidential information. In general, any action that gives out information about a user's borrowing record, whether now or in the past and whether to the same user, or to someone else, is part of a case that is linked negatively to the policy of maintaining user privacy; this policy is marked as important here.

It is worth discussing one supporting scenario for this case in more detail, that of a devious-borrower. It consists of the following actions (transitions): borrower B checks out resource R; borrower C gains access to B's identity; C queries the system, as B, to find what resources B has checked out; C learns that B has checked out R. The point to note is the need to represent behavior for both the system *and* its environment. In this case, the scenario uses existing components of **example** (check out, query), but also supplies environment components of its own (illicit gain of one borrower's id by another) to run the simulation. In summary, the scenario extends **example** with new components (objects, places, transitions) to make a point.

2. The constraint that the check out action must be carried out by a staff person (L8) is *obstructive*; it matches a monitored-withdrawal case, which in turn is linked negatively to a (sub) policy of minimizing circulation staff.

3. Certain actions are *not supported*. In particular, the actions (and associated cases) of adding and removing a book (see L3) can be viewed in finer grain, e.g., remove-lost, remove-stolen, remove-damaged, replace-lost, replace-stolen, replace-damaged. These type of actions are captured in cases linked positively to various sub-policies of accounting for human foibles. Since the general human foibles policy is viewed as important, these cases look for a match. None is found.

Also, the division of users into groups (staff and ordinary borrowers in L7) is without corresponding actions to add and remove members from a group. There are corresponding cases in **model** that are linked positively to the policy (and sub-policies) of recognizing the human dynamics of group membership, and these cases expect to match if that policy is important. It is and no matches are found.

While no superfluous components were found on this run, it is not hard to change policy values to generate such a critique. For instance, by marking the (sub) policy of accounting for human forgetfulness as unimportant, the query in L5 becomes not only obstructive (see 1 above), but superfluous as well, a bad combination in general.

To further evaluate these results, we asked an experienced library analyst to critique the text description of the Appendix outloud, and recorded the session in both audio and video form. We will summarize the four major points to come from this work; in [Fickas, 1987] and [Fickas *et*

Ham, 1985]; it allows computable predicates on both arcs and transitions, each of which can reference token/object slot values and token/object types. Initial versions used NIKL [Kaczmarek *et al*, 1986] as the basis for implementation; more recent versions are built on the SIMKIT package of KEE.

The **model** is used to represent a set of "interesting" problem specification cases to consider for a particular domain. The domain we have chosen initially is that of resource management systems. Our cases, to date, are hand-coded transcriptions taken from 1) written texts and articles on analyzing problems in the resource management domain, and 2) protocols of human analysts, familiar with the domain, constructing and critiquing specifications. A case consists of the following fields:

- *A description of a particular pattern to look for in* **example**. As discussed above, the representation used in **example** is that of an augmented state-transition net; the pattern here takes the form of a sub-net.
- *A link to a policy*. This is used to index the case to higher level concerns within the domain. This will be discussed shortly.
- *An ordered set of simulation scenarios*. These are used to demonstrate various aspects of the case. Each scenario contains operational instructions for 1) setting up initial data, 2) constraining non-deterministic control to exemplary paths, and 3) running the sub-net in **example** linked to the case.
- *Canned text description*. As the name implies.

The correspondence links tie model cases to actual constructs within the specification, i.e., they bind components of a case's sub-net to components in example. As an illustration, figure 2 depicts a sub-net pattern in a resource underflow case. Correspondence links have been built to bind the sub-net components to a particular specification of a library under critique; the specific library/example components that are bound are shown in parentheses.

To use the critic, we must first translate the specification to be critiqued into **example** format, i.e., into our augmented Petri-net representation. The specification/**example** we will discuss in this paper is that of an automated library system, a standard in discussing specification research. The particular incarnation we will use comes from the problem set handed out prior to the Fourth International Workshop on Software Specification and Design [FIWSSD, 1987]; it is reproduced, with line numbers for reference, in the Appendix.

After translation, correspondence links must be forged between case components in **model** and specification components in **example**. The system supplies some matching help here by looking in **example** for token, place, transition, and predicate names that are commonly used in resource management domains, and hence in **model** cases. Components unrecognized by the system must be manually linked by the user.

Finally, we must deal with the overall goals of the client. In particular, we have come to believe that there is no such thing as an inherently good or bad specification, only one that does not conform to the resource limitations and users' goals in force. Thus, the goodness or badness of a component in **example** can only be judged *relative* to the user's goals and the resources available. We will use the term *policy* to denote both organizational goals and resource constraints. For the latter, we will include resource limits on both the development of an implementation and on the operational environment, e.g., "minimize operational staffing costs". Based on discussions with domain experts and a study of the domain literature, seven broad policy classes were defined for resource borrowing systems[2]:

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.
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.

Each of these seven can be further refined, e.g., maintain privacy of users' borrowing record, maintain privacy of users' queries, etc. We can also further specify each policy in terms of more specific domains, e.g., maintain an adequate stock of books on the shelves, maintain an adequate stock of video tapes available for rental.

We allow each policy to be in one of three states: *important* - the client has explicitly noted that the policy should be enforced; *unimportant* - the client has explicitly noted that the policy should be ignored; *unknown* - no explicit statement has been made about the policy. A value given to policy P is inherited by all ancestors of P. Thus, marking the policy of accounting for human foibles as important will in turn mark all refinements of that policy -- forgetting books, stealing video tapes -- as important. Conversely, we can mark policies in a finer grain if necessary: prevention of stolen items is unimportant, but reminding forgetful users of borrowed items is important.

[2]We make no claim that this is either a necessary or sufficient list of policies, but simply one that has allowed us to handle the set of resource management problems that we have studied to date. Also, it is clear that certain policies in this list extend beyond this domain.

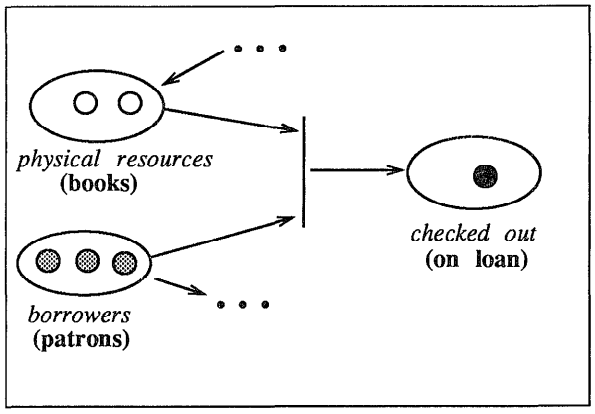

Figure 2. Resource underflow case (with bindings)

al, 1987] we describe our protocol analysis and results more fully.

1. We found general support for our representation of policies. The analyst spent the first part of the session establishing the "goals" of the library. These goals were all covered by our current set of policies.

2. The analyst's critique registered well with the critic's. With few exceptions (see below), the same type of case was given by analyst and critic, if not in the same style.

3. The major discrepancy between analyst and critic was in the analyst's ability to deftly juggle competing concerns. In particular, she was able to weigh the importance of various policies, 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, the query in L5 along with the constraint in L9 could be viewed as a livable compromise.

4. The analyst's ability to generate, at will, both abstract and concrete examples to back up a point was impressive. The critic's contrasting lack of sophistication in this area is discussed in the next section.

In summary, the comparison of the critic's analysis with that of the human analyst points to the representation of policies in a specification critic as a key component. Our findings also point to the need for a more refined view of policies, their interaction, and their connection to domain cases, and a more powerful means of backing a case with a range of scenarios. The next section discusses these issues and others raised by our experience with this critic.

4 Current work

Below we list the issues that we feel must be of immediate concern for the next version of the critic.

Policy interaction and utility. Wilensky describes different types of goal interaction, and plans for handling each of the types [Wilensky, 1980]. It seems clear that we will need something similar for our policies, e.g., "user privacy always overrides timely access to resources". Along the same line, a notion of policy utility beyond the simple values important and unimportant will be necessary. This has been brought home, in particular, as we have begun to look at compromise strategies that allow two or more conflicting policies to each be partially met simultaneously [Fickas, 1987].

We also note the correspondence of *features* in [Chapman 1982] and *goals* in [Mostow and Voigt 1987] to our work on policies in general.

Simulation. We have shown how simulation can be an effective critiquing tool. However, its full potential would seem to rest on better models of explanation in general, e.g., when should we use abstract or concrete data, how much of the context must be provided, how far do we have to follow the results. As an example, we have shown a case of "a run on a depository". Is this enough to convince a client that a potential problem exists in his or her specification? In particular, we do not show a real consequence of a depository running out of resources, i.e., loss of confidence by borrowers turned away from the depository that it is a reliable source. In other words, we expect the client to infer this type of knowledge, and to decide if it is worth worrying about. Whether this is warranted or not is clearly dependent on the sophistication of the client in the domain.

Along the same lines, what is the right mix of scenarios to attach to a case? For example, in the scenario in figure 2, we jump right to the unlikely event of everyone wanting resources at once. Examples such as this are sometimes easy to dismiss as too extreme (however, see "Bank Runs", the formation of the FDIC to prevent them, etc.). A more convincing scenario might show a gradual depletion of resources under average (or even favorable) borrowing conditions. In the end, we might like a progression of best case to worst case scenarios. We can simulate this crudely in our current critic by attaching an ordered set of scenarios to a case. In operation, we expect a best case criticism to be presented first. If the user decides to address the criticism by editing the specification, the critic moves on to a slightly worse case. This cycle of system critiques and user fixes continues until either the system has thrown its toughest critiques at the specification (e.g., the extreme scenario in figure 2 is reached) and the user has addressed them all, or the user has decided to live with some scenarios not being handled (because of limitations in space, time, money or any number of other reasons).

We note the similarity of the above argument style with that of Rissland's work in the area of case-based legal reasoning [Rissland, 1986]. We also note that in her system a single case or scenario is represented in addition to one or more dimensions along which the case can be stretched (for instance, "resource supply and demand"). A separate example generator can instantiate the base case by moving along one or more dimensions. In general, this dimension/generator approach is clearly more powerful than the explicit scenario list we now employ, and is one that we believe will move us closer to that seen in our human analyst.

An interactive critic. The goal here is an interactive editor for developing specifications, i.e., a system that provides

tools for both construction and criticism in an interleaved fashion. This is in much the same spirit as that of deficiency-driven algorithm design in Steier and Kant's DESIGNER system [Steier and Kant, 1985]. Our current system supports both a specification/**example** editor and the critic we have discussed in this paper. Thus, a user can edit a specification, run the critic, respond to criticism through further editing changes, etc. The problem is a lack of automation in matching, and rematching after changes. We are exploring two approaches to the matching and rematching problem. First, we have given the editor a component catalog for the resource management domain. These components are ones found in our cases. If the user selects components from the catalog in the construction of his or her specification, we can automatically match them against cases (actually, we just follow component-to-case links, avoiding matching altogether). If the user supplies non-catalog components, then we must fall back on common names, and finally, user intervention.

Interleaving of editing and critiquing brings up the rematching problem: given a local editing change, we would like to avoid rematching the entire specification to the entire case-base. Our approach has been to isolate changes to a small subset of specification components, rematching these while retaining past matches outside of the local context. While we have had some preliminary success in localizing changes in the specification language we are using [Fickas, 1987], the problem remains an open and difficult one.

References

[Chapman, 1982] Chapman, D., A program testing assistant, *Communications of the ACM*, September, 1982

[Fickas, 1985] Fickas, S., Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, November, 1985

[Fickas, 1987] Fickas, S., Automating the Software Specification Process, Technical Report 87-05, December, 1987, Computer Science Department, University of Oregon, Eugene, OR. 97403

[Fickas *et al*, 1987] Fickas, S., Collins, S., Olivier, S., Problem Acquisition in Software Analysis: A Preliminary Study, Technical Report 87-04, August, 1987, Computer Science Department, University of Oregon, Eugene, OR. 97403

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

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

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

[Mostow and Voigt, 1987] Mostow, J., Voigt, K., Explicit integration of multiple goals in heuristic algorithm design, In *Proceedings of IJCAI-87*, Milan, 1987

[Rissland, 1986] Rissland, E., Dimension-based analysis of Hypotheticals from Supreme Court Oral Argument, COINS, University of Massachusetts

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

[Swartout, 1983] Swartout, W., The GIST Behavior Explainer, In *Proceedings of AAAI-83*, Washington, DC, 1983

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

[Wilensky, 1980] Wilensky, R., Meta-planning, In *Proceedings of 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

Appendix

L1. Consider a small library database with the following transactions:

L2. **1.** Check out a copy of a book / Return a copy of a book;

L3. **2.** Add a copy of a book to / Remove a copy of a book from the library;

L4. **3.** Get a list of books by a particular author or in a particular subject area;

L5. **4.** Find out the list of books currently checked out by a particular borrower;

L6. **5.** Find out what borrower last checked out a particular copy of a book.

L7. There are two types of users: staff and ordinary borrowers.

L8. Transactions 1, 2, 4 and 5 are restricted to staff users,

L9. except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves.

L10. The data base must also satisfy the following constraints:

L11. - All copies in the library must be available for checkout or be checked out.

L12. - No copy of the book may be both available and checked out at the same time.

L13. - A borrower may not have more than a predefined number of books checked out at one time.