Critiquing a Software Specification

Stephen Fickas
P. Nagarajan

CIS-TR-88-01
February 3, 1988

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Critiquing a Software Specification

Stephen Fickas

P. Nagarajan

Computer Science Department
University of Oregon
Eugene, OR. 97403

Abstract

Our interest is in a knowledge-based system that assists in the construction of a formal specification. In this paper, we discuss one component of such a system, a knowledge-based specification critic.

1. Specification as a process

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

This paper is a revised version of one appearing in the Fourth International Workshop on Software Specification and Design, Monterey, 1987 [7].

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

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

CIS Technical Report 88-01 - January, 1988.

bringing more of this type of expert analysis knowledge into the machine. While informal guidelines are useful for many things, we believe that to successfully tackle such traditional problems of specification as ambiguity and incompleteness, we will have to have the machine become involved. This in turn must rest on results in two areas, a formal model of the specification process itself, and a computer-based representation that will allow it to be automated. We believe that one necessary component of such a model will be that of a specification critic.

1.1 A specification critic

We are working on a system that attempts to acquire a problem specification that includes a description of the objects, actions, and constraints of the intended system. In our view, the production of such a specification is not so much a translation process as an interactive *problem solving* process with both client and analyst involved in supplying parts to the final product. This requires our system/analyst to have a thorough understanding of the application domain, and the ability to both critique user descriptions, and suggest components of its own. Part of this process is the type of syntactic analysis that is concerned with dataflow errors, e.g., unused data, disconnected actions. An equally important part is validation of intent. That is, we might expect to take a perfectly valid (syntactically correct, consistent, unambiguous) specification, and attempt to poke holes in it. To do so, we will argue that the following points must be considered:

- *The specification process often involves conflicting goals.* Almost any complex specification is the result of compromises and trade-offs between competing concerns, i.e., there is no perfect specification. Conflicts may arise between different user groups, users and administrators, performance versus budget, and a host of others.
- *The process involves collaboration between client and analyst.* Stated more strongly, clients have only a vague notion of what they want, and a narrow view of what is possible. A good analyst has expertise in pulling out the key points of a client's problem, presenting convincing arguments for and against the inclusion of certain components, and finally filling in the necessary details. All of this occurs within a rich, interactive, problem solving context that includes processes for problem acquisition, disambiguation, and critiquing.
- *The process is inherently knowledge-based.* This point is actually a summary of the two above. To design a specification through refinement, to identify the special cases, to find compromises for conflicting goals, and to argue effectively with a client about what has been wrought, requires knowledge of the application domain, knowledge of implementation possibilities, and knowledge of the production environment, *among others.*

In the remaining sections of this paper we will discuss, in chronological order, our attempts to address these points by construction of a set of computer-based critics for analyzing software specifications.

1.2 A note on terminology

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

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

2. What makes a good analyst?

One could attempt to answer this question by observation. For example, we have seen various forms of the following bugs and techniques by informally watching novice and expert software analysts at work:

Analysis Bug: *the naive physical model*. In domains that require an information management system (MIS) to control a physical system (PS), the MIS must represent the PS in an adequate level of detail. If the level of detail is to too idealized or naive, the delivered MIS will not model certain events and associated states in the PS, and will have a propensity to become wedged.

Useful Analysis Technique: *usage scenarios*. To combat the naive physical model, we have observed that the more scenarios that can be generated showing the system in use, the better the chance of finding the events and states that must be represented. The best analysts appear to be the ones able to find an adequate level of representation by setting up hypothetical situations for a client to work through. **Corollary**: the more knowledge an analyst has about the client's domain, the better he or she is at generating scenarios.

Non-Useful Analysis Technique: *the customer is always right*. Analysts who rely on the client knowing what is possible and what is required generally produce only marginally accepted systems. While inexperienced analysts often hold spirited discussions on what can be delivered in a given time frame, they rarely question the client's statements of need, acting more as Santa jotting down a wish list. As we will discuss in the next section, this seems contrary to the behavior of experienced analysts, and in our observation, is a critical factor in eventual client dissatisfaction with delivered code.

To provide more formal support for the above speculations, we ran a set or protocol experiments. Our interest was in the techniques used by an experienced analyst (as opposed, for instance, to the mistakes made by novices). The details of these experiments are reported in [9]. Because they have had an impact on our later work on a specification critic, we will summarize them here. Looking at three different analysts (with an average of 10 years experience) and four different clients over four different sessions, we found the following:

> **Finding 1**: experienced analysts use hypothetical examples to 1) explain a concept to the client, 2) argue for the inclusion of a requirement proposed by the analyst, 3) argue against the inclusion of a requirement proposed by the client, and 4) further refine their understanding of the client's problem.
>
> **Finding 2**: related to 1, experienced analysts are aware of the higher level policy issues in a domain, and are able to use this knowledge to show a client the benefits and drawbacks of including certain components in their specification. In particular, two sessions were largely devoted to discussing the resources available (e.g., development budget, training budget, maintenance budget) and politics involved in installing an MIS in an existing bureaucracy.
>
> **Finding 3**: experience analysts use summarization to 1) verify their understanding of a concept (''Let me make sure I have this right, a file contains X,Y,Z, right?''), 2) verify that they have covered all of the necessary aspects of a concept (''So, we have covered how a file is created, and how a file is updated. Oh, that reminds me, I haven't considered ...''), and 3) as an understanding check between client and analyst (''Let's go over this one more time to make sure we understand each other. You want the following actions: ... [client: ''Right'']. And these special cases hold: ... [client: ''Right'']). In particular, it was rarely the case that a summarization had any finality to it, but instead was a conscious or unconscious effort at verification and acquisition.

Each of these findings has lead to new lines of research in our project as a whole. In this paper, we focus on findings 1 and 2 above. Specifically, we are attempting to capture the flavor of our analysts' constructive criticism of a client's proposed requirements.

3. Critics 1 through 5

Our interest in a specification critic system is just one component of a larger project whose goal is to provide assistance to a software analyst in producing a formal specification. This project, called Kate [10], focuses on the following 3 components:

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

Component 2. A specification construction component that controls the design of the client's emerging specification.

Component 3. A critic that attempts to poke holes in the client's problem description.

Our focus in this paper is on the first and third components, the domain model and the specification critic. Our attempt to build a critic has taken us through several iterations. We will organize our discussion around the 5 major versions that have come from our efforts.

3.1 Critic-1

As an initial effort, we constructed a tool made up of two pieces, a specification language and a rule-based critic. The framework of the tool consisted of a **model** part, an **example** part, and correspondence links between components in **model** and **example**[3]. The use of **model** was as a representation of a particular application domain D. The use of **example** was as the representation of a particular problem specification in D. Our specification language, used by both **model** and **example**, can be viewed as an augmented Petri-net representation that allows computable predicates on both arcs and transitions [20]. The language also supports abstraction through a class hierarchy in a manner similar to Greenspan's RML language [12]. Initial versions used NIKL [13] as the basis for implementation; more recent versions use KEE/SIMKIT [14].

To use the tool, several manual steps had to be employed. First a representation of a particular domain had to be coded in **model**. As our initial domain, we chose that of resource management. We justify this choice elsewhere [6], but note here that the domain covers a wide range of applications including management of human, physical and information resources, and forces one to deal with users, staff personnel, resource overflow and underflow, security, and a host of other "interesting" problems. An outline of the components of critic-1's **model** is given below.

Resources, e.g., physical resources, borrowable resources, information resources, humans, office space, furniture, hardware, courses, sessions, seats, books.

Resource depositories, e.g., physical plant, buildings, class rooms, banquet rooms, conference rooms, borrowing houses, libraries.

Resource managers, e.g., staff, editors, custodians.

Resource users, e.g., attendees, students, computer users, borrowers, patrons.

Resource operations, e.g., add resource, remove resource, gain access to resource, return resource, wait for resource.

---

[3] Because an example-model link is often hypothetical, each such link is made in a separate context. In this way, alternative mappings from example to model can be maintained, reflecting the frequent ambiguity in a client's description. For example, note the ambiguous nature of the constraint on transaction 1 on line L8 in Appendix A.

Queries, e.g., resources available, who-has-what

Security operations, e.g., give authorization, remove authorization, check authorization.

Resource constraints, e.g., maximum size, minimum size, borrowing limits, time limits.

Resource constraint management, e.g., waiting lists, dunning notices, fines.

The second step involves translating the specification to be critiqued into **example** format, i.e., into a concrete form of our Petri-net representation. The specification/**example** we will discuss in this section is that of an automated library system, a standard one in discussing specification research [18]. The particular incarnation we will use comes from the problem set handed out prior to the Fourth International Workshop on Software Specification and Design [11]; it is reproduced, with line numbers for reference, in Appendix A.

For the third and final step, correspondence links must be forged between components in **model** and those of **example**.

To reiterate, all three of these steps were carried out manually, i.e., by members of our group. Once **model**, **example**, and the correspondence links were in place, a rule-based critic, implemented in the ORBS language [8], was used to find components of **model** that were not linked to components in **example**. A typical rule takes the following form:

(*defrule* <name>
match: <**model** component> <correspondence link> <**example** component>
. . .
<**model** component> <correspondence link> <**example** component>
test: <predicate 1> . . . <predicate n>
action: <warning action> . . . <warning action>)

The left hand side of the rule matches on mappings from **example** to **model**. Further test predicates (in the form of Lisp functions) can be applied to the matching components. The right hand side takes actions to warn of possible problems. Note that virtually no domain-specific knowledge resides in rule form. Instead, the rules contain representation-specific knowledge, and hence, can be viewed as a type of syntax checker for the language and links we have defined.

Running the tool on the problem description in Appendix A (after it was translated into **example** form), we were able to produce warnings such as the following:

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

Each of the missing components above was viewed as ones *typically* found in a physical resource borrowing system, i.e., components of **model**.

3.2 Critic-2

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

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

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

These new warnings capture, respectively, the notion that certain types of queries compromise user privacy, and that placing constraints on borrowing may adversely affect a user's ability to perform some task.

3.3 Critic-3

It became clear by talking with expert analysts that there is really no such thing as an inherently "bad specification", only one that does not conform to policies in force. In particular, the goodness or badness of a component in **example** can only be judged *relative* to the goals, resources and policies of the client[4]. Thus, the two-part partition of **model** in critic-2 was overly rigid. What we really need is 1) a broad representation of the basic components of the domain, 2) a representation of the policies of the domain, and 3) a way to show the relationship between domain components and policies. It is only after we have this third component that we can point to problems in the specification in a convincing fashion..

We built a tool that captured all three components listed above. This involved combining the two parts of **model** in critic-2 into a more general, single model. Further, a simple representation of policies was defined based on discussions with domain experts and a study of the Library Science literature [3]. Seven broad policy classes were defined for resource borrowing systems[5]:

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 privacy of users.
4. Account for human foibles, e.g., forgetting, losing items, stealing.
5. Account for development resource limitations, e.g., money, staff, and time available to develop system.
6. Account for production environment limitations, e.g., money, staff, and time available to run and maintain the delivered system.
7. Recognize the human dynamics of group (patron, staff, administration) membership.

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 should note here that Mostow's group at Rutgers is studying the implicit design decisions that go into implementing a system [15]. Their approach has been to produce a rationalized design that would lead to an existing piece of code, given design recollections from the original implementors of the code, and a theory of interacting design goals and methods for combining them into a coherent design. Although

---

[4]We will henceforth use *goal* and *policy* synonymously. Further, we will include resource constraints, both on the development of an implementation and on the operational environment, as representable as policy decisions, e.g., "minimize operational staffing costs". See section 3.4 for further discussion.

[5]We make no claim that this is a either a necessary or sufficient list of policies, but simply one that has allowed us to handle the set of resource management problems we have studied. Further, certain items clearly extend beyond this domain into interactive systems in general.

they address two separate software processes, we expect that results from their work and ours will eventually show a common notion of goals/policies. Along this same line, we see much in common with Wilensky's representation of meta-goals [21].

We allow each of our seven policies 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.

As a final step, **model** components were linked to policies. Each such link can take on one of two values: *positive* - the component supports the policy; *negative* - the component thwarts the policy.

Figure 1 shows a small portion of **model** in critic-3 with policies denoted by square boxes, domain components (objects, actions, constraints) denoted by rounded boxes, and policy-to-component links as highlighted arcs: negative arcs end with a black circle, positive arcs are drawn normally. Arcs between domain components are taxonomic.

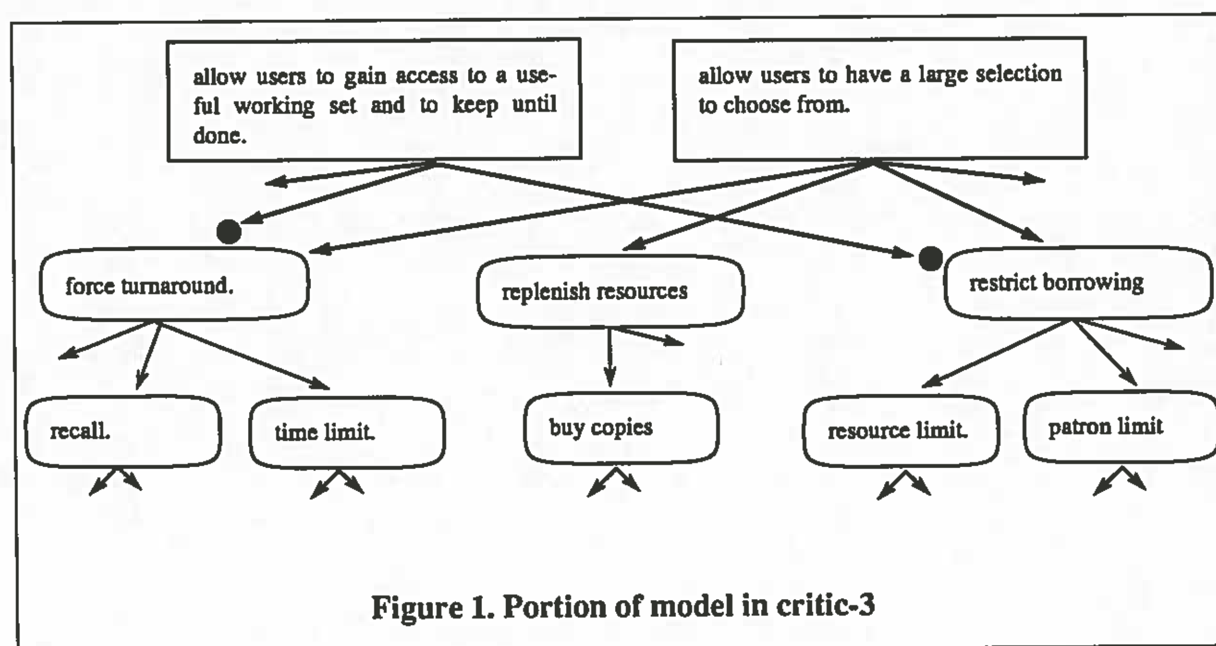

Figure 1. Portion of model in critic-3

Figure 1 represents a static view of **model**. When used in a critique, correspondence links would exist between domain components (the leaves of the component taxonomy) and example components. Further, each policy would be marked with a value important, unimportant, or unknown (the default).

Critic-3's rules can now be more tightly classified into three types.

Type 1: A policy marked as important (or unknown) is linked *positively* to a **model** component. No corresponding component can be found in **example.**

Type 2: A policy marked as important (or unknown) is linked *negatively* to a **model** component. A corresponding component can be found in **example.**

Type 3: A policy marked as unimportant is linked *positively* to a **model** component. A corresponding component can be found in **example.**

As can be seen in figure 1, policies can be linked to abstract components. Critic-3 takes the following conservative view of this situation:

Type 1: *all* leaf nodes that share the abstract node are checked. Warnings are given for *any* that cannot be found in correspondence with **example** components.

Type 2: *all* leaf nodes that share the abstract node are checked. Warnings are given for *any* that can be found in correspondence with an **example** component.

Type 3: this is a weaker type of critique than the first two. It 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. Like type 2, *all* leaf nodes that share the abstract node are checked. Warnings are given for *any* that can be found in correspondence with an **example** component.

It is critical to note that it is possible, and in fact typical, for the same **model** component to have both positive and negative links to some set of policies. That is, it may positively support policy A and negatively affect policy B (In figure 1 "A" is *give users a useful working set*, and "B" is *give them good stock on hand*). While we would like to think that A and B are never both simultaneously marked as important, it is not untypical for a client to describe a conflicting set of goals or policies. One key process in specification design is then coming to grips with conflicting policies through various forms of trade-off and compromise. Critic-3 does *not* note these conflicts, but simply presents the three types of warnings given above.

In conjunction with discussing critic-3's performance on Appendix A, we will present a university library analyst's assessment of the same problem description. In particular, we asked the analyst to critique the text description outloud, and recorded the session in both audio and video form [9]. There were three things worth noting about the results. First, much of the session was taken up with establishing "the ground rules", i.e., the policies in effect. In critic-3's critique, we left all but one of the policy nodes marked as of unknown importance to reflect the meagre information given in the text description. In our session with the library analyst, we felt compelled to fill in at least some of the policy details of the problem under direct questioning by her. The negative side of this is that we are no longer comparing exactly the same problem. However, there is a positive side as well: the policy questions raised by the analyst were at least superficially equivalent to ones in our **model**

representation. Thus, we find support for 1) the general notion that a critique of a specification must take into account the overall goals and available resources, and 2) the specific use of policies to capture this information in critic-3.

The second thing to note is the actual critique given by the analyst. First we will summarize the major findings of critic-3's critique of the problem description in Appendix A (henceforth, WS), and then compare it with that of the library analyst. We remind the reader that critic-3's criticism is given in light of only a single component-policy link; other links that might be used to counter the criticism are not used to soften the critique. Further, the criticisms of the problem description rest on viewing a policy of unknown importance as important, a fact built in to the rule based critic.

1. Actions 4 (line L5) and 5 (line L6) are problematic; 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 linked negatively to the policy of maintaining user privacy.
2. The WS constraint that the check out action must be monitored by a staff person (line L8)[6] is problematic; it goes against the policy of minimizing operational staff. This policy was marked as important by at least one interpretation of *small* in the first line of the description.
3. There are missing actions (and associated states); the actions of adding and removing a book (see L3 in WS) can be viewed in finer grain, e.g., remove-lost, remove-stolen, remove-damaged, replace-lost, replace-stolen, replace-damaged. These type of actions are linked positively in **model** to the policy of accounting for human foibles.
4. A borrowing limit (L13 in WS) is a problematic constraint; it is linked negatively to the policy of giving users a sufficient working set.
5. A borrowing time limit is missing; it is linked positively to the policy of maintaining an adequate stock on hand.
6. The division of users into groups (staff and ordinary borrowers in WS, see L7) is without corresponding actions to add and remove members from a group. These actions in **model** are linked positively to a policy of recognizing the human dynamics of group membership.

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

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

- (2) She assumed that no library could be effective without some type of control on check out and check in. Thus, she was assuming that if we needed to minimize cost, it would not be through removing staffing from circulation. The message for us was that we would need some type of prioritization ordering on policies.
- (5) Her assumption was that you must give users both an adequate set of resources (prompting her criticism akin to critic-3's in 4), and an adequate time to keep them out. Again, there seemed to be an ordering of policies that placed this above concerns of keeping adequate stock on hand.
- (6) These seemed so obvious that she assumed that they were omitted just as a matter of detail, and that any competent designer would include them (!).

In summary, the comparison of critic-3's analysis with that of the human analyst reassures us that the representation of policies in a specification critic will be a key component. Our findings also point to the need for a more refined view of policies, their interaction, and their connection to domain components.

3.4 Critic-4

While the representation of policies in critic-3 seemed to capture the type of goal or context information employed by human analysts, it lacked a notion of policy interaction. For instance, many components in critic-3's **model** were linked to *conflicting* policies. It is clear from our protocol studies [9] that experienced analysts have at least a partial ordering on the policies of the domain, an ordering that can be used to choose among various components needed in the final specification. In critic-4 we defined a partial ordering on policies. Thus, the client may note things like "user privacy overrides timely access to resources", where user privacy and timely access to resources are both policies represented in **model**. Now a query such as that in item 4 in Appendix A can be more confidently analyzed: while it has support from one policy, that policy is overridden by another that notes its negative aspects.

Besides concern with the interaction between policies, we saw the need for a more detailed representation of the policies themselves. Specifically, in pursuit of generality, we believed we might be overloading the policy notion in critic-3, where we use such policies to represent a) the various goals of the users and administrators of the system, b) the resource constraints of the system, and c) a simple model of human behavior in resource management applications. For the latter in particular, concepts such as borrowers forgetting, borrowers stealing, borrowers gaining and using information illicitly, tend to interact with specification components in complex ways, sometimes requiring something more akin to a script-like representation. In critic-4 we include the notion of a *usage scenario* to capture this knowledge. A usage scenario consists of 1) a set of Petri-net components that represent a particular type of user transaction with the system (TAXIS scripts [1] use a similar approach), and 2) an optional list of policies that are positively or negatively reinforced by the scenario. We employ usage scenarios to capture the way users and staff typically use and misuse a system in a particular domain. An example may be useful here.

Suppose that a user Boris wishes to know the reading material of a user Joe. The key to this scenario is the condition that Boris be in a state of knowing Joe's identification. Reaching such a state can be trivial if personal names are used as id; passwords would require a more

complex impersonation scheme. The objects of the scenario, then, are two users Ub and Uj, and a resource R. The scenario is linked negatively to the policy of maintaining user privacy, and can be paraphrased as follows:

> "A user Uj (Joe) has checked out a resource R; Ub (Boris) is in a state in which he can identify himself as Uj; Ub queries the system for the resources checked out by Uj; Ub moves to a state in which he is aware of the identity of R."

As can be seen, there are two parts to a scenario such as this: 1) system components that represent the objects, actions and constraints that we have control over in the specification (e.g., the check out and query actions), and 2) the environment components that describe how we might expect the outside world to behave, at least in the cases where it impinges on the system. To use the above scenario in a critique, Critic-4 must match the system components with components in **example**; the environment components come along as necessary and sometimes unwanted baggage.

We can look at another scenario that matches the same query as the scenario above, but has a positive link to the policy of accounting for human foibles, specifically, human forgetfulness. In particular, suppose that a user U has checked out a resource R, but has now lost track of its identity.

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

Our claim is that scenarios such as those above model the ways in which we expect the environment to interact with the system being specified, thus providing a kind of extended interface-model or closed specification [5]. By linking environment behavior to policies, we gain further definition of policies. At the same time, scenarios supply a type of rationale for specification components that might be otherwise missing. We can use the second scenario above to illustrate this. In particular, in this scenario the forgetful behavior of a user ties together the check out, check in, and query actions; if we remove the check out action from the specification, we would expect the query to go as well; it makes no sense to worry about forgetful users when users have nothing to forget.

One might argue that the same type of analysis can be carried out by looking at the preconditions and postconditions of the actions involved. Thus, the query action may have a precondition that a borrowing database must exist; when we remove the check out action, this database should go with it, leaving the query precondition unsatisfiable (an arc with no source/place in our Petri-net representation). Such an argument is certainly valid in the example we have presented. However, there are three further points worth making. First, a dataflow type of error message gives a rather syntactic view of the problem. In our scenario we have linked the three actions by the expected behavior of the environment (i.e., the users). Thus, we can *explain* the connection between the two actions in domain terms (see paragraph above).

Second, if someone adds a check out action to a specification, we can *show the need* for adding a corresponding query on the resources checked out by a user.

Finally, we can also *comment on the effect of removing* the query action while leaving the check out and check in actions intact: the check-out/check-in cycle may be broken for some set of resources, leading to books not returning, leading to dwindling stock on the shelves, ...

The second and third cases, in particular, cannot rely on system-syntax analysis alone to produce their results.

We are now at a version of our critic that incorporates policies and usage scenarios to analyze a specification. However, it is still a batch system. The next version of the critic attempts to incorporate our results into an interactive tool.

3.5 Critic-5

Our goal here was to integrate critic-4 into an interactive editor for developing our Petri-net style of specifications, i.e., a system for building **examples**. To achieve this, we modified critic-4 in the following ways:

1. We made the policies in **model** editable by the user. Thus, he or she can now mark none, some or all of the policies as important or unimportant. Further, he or she can change a policy's importance at anytime during the specification process, or simply leave it unmarked (i.e., of unknown importance).
2. It is now up to the user to ask for a critique of the evolving specification. Again, this can occur at anytime during the construction of the specification.
3. We added a simulation component to the critic that allows it to "run" a usage scenario with an animated graphical display.

The new system, which we will call critic-5, allows a user to incrementally build a specification, call for a critique when ready, edit the specification, reinvoke the critic, et cetera. The system supplies some help in establishing correspondence links, e.g., checks on name matches, but in general still relies on the user to designate **model/example** links[7].

The presentation of a critique by critic-5 is of some interest. In critics 1 through 4, an explanation consisted of a piece of canned text with appropriate references to **model** and **example** components. An effort was made to expand the explanation facility in critic-5 on the basis of a set of protocol experiments run by our group. In particular, a library analyst that we studied was able to give a detailed analysis of a problem, and produce a seemingly unlimited number of examples on demand. In our view, she had 1) a deep understanding of the "first principles" of library science, 2) a knowledge of specific libraries and thus specific examples, and 3) the ability to generate hypothetical examples to illustrate some abstract principle or policy. In critic-5 we have attempted to tackle the last issue, that of presenting a critique by way of examples. Our approach is to attach to each usage scenario a description for generat-

---

[7]This component matching problem is clearly problematic; we expect its solution to lie finally with results from the concept learning area (see [4], for example). Currently, the system uses simple name correspondence when possible, and otherwise, queries the user.

ing an example that demonstrates the scenario to the client, i.e., to make the scenarios dynamic versus static. To critique a particular state S (the **example**), critic-5, as with critic-4, first matches components of S against cataloged scenarios. If a scenario C matches, an example is generated in S that can include either concrete or abstract data (Petri-net tokens in our case) for execution purposes. Figure 2 shows an example. This scenario of "a run on a resource depository is presented by the system as 1) a Petri-net *place* that holds a limited number of resource *tokens* that are checked in, 2) a *place* that holds one or more borrower *tokens*, 3) a *place* that holds zero or more resource *tokens* that are checked out, and 4) a *transition/process* that takes as input *tokens* from checked in resources and borrowers, and moves the resource *token* to the checked out *place*.

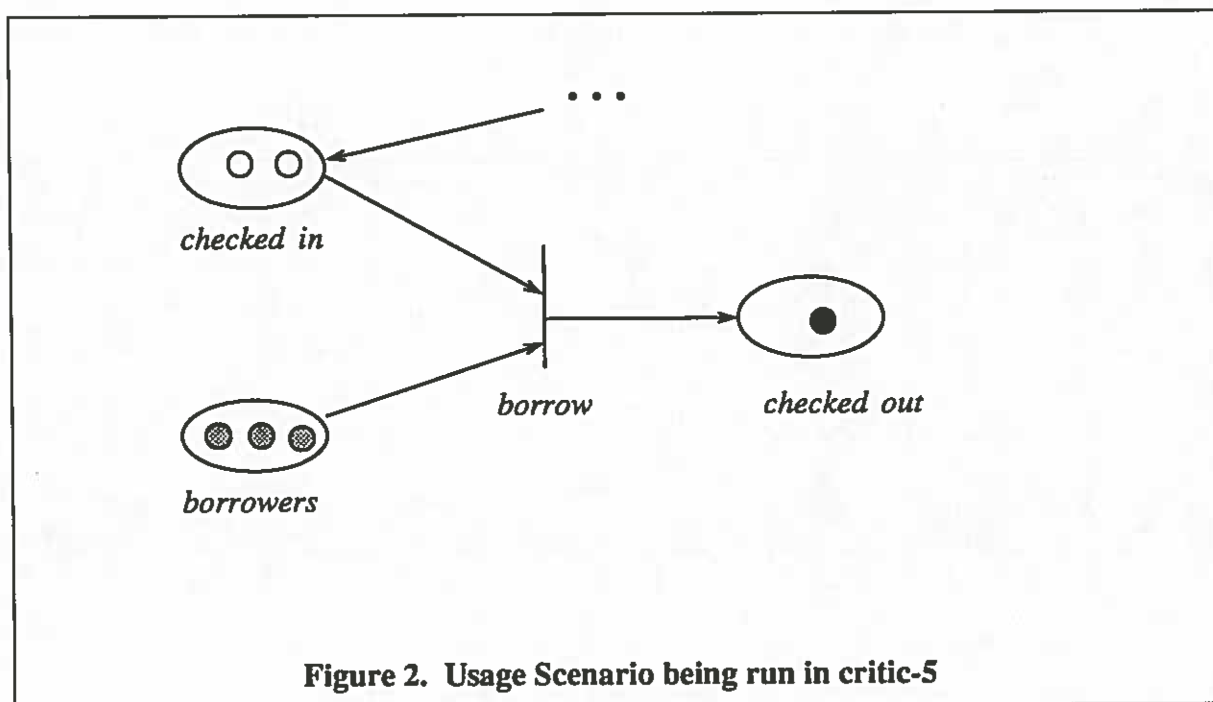

Figure 2. Usage Scenario being run in critic-5

Also included in the case is the initial placement of tokens to start the scenario, and a directive to the non-deterministic control mechanism of the net to fire the borrow process whenever possible, i.e., constantly until no more resource tokens are available. Along with the execution trace of the scenario (using SIMKIT's animation tools [14]), the system also generates a canned text summary of the underflow problem for the user in a form similar to that of critic 4.

There are several things to note about critic-5's simple attempt to generate and run examples. First, there is some question about the level of detail needed to make a point with such animated examples. For instance, in figure 2 we do not show the real consequences of a depository running out of resources: loss of confidence, by borrowers turned away, that the depository is a reliable source. While we could have extended the scenario in figure 2 to mod-

el at least some aspects of this more detailed information, we instead chose to rely on the user's ability to infer the implicit consequent. It seems clear that choices on the level of detail presented must ultimately be linked to the sophistication of the client in the domain.

Second, note that the system is attempting to demonstrate an extreme case in figure 2. For instance, other processes might exist in S for replenishing resources either by checking them back in or buying new ones (as represented by the elision of an input arc to *checked in* in Figure 2), or even removing borrowers from the pool. However, at least in terms of non-deterministic control, an unrestricted borrowing process *could possibly* lead to the case shown. It is the analyst's and client's decision whether it is worth building in further restrictions (e.g., by attaching predicates to the borrow process) to prevent the case.

Finally we note that it looks worthwhile exploring the representation of best-case and average-case scenarios as well, thus extending the type of examples that can be generated, and coming closer to Rissland's notion of a continuum of cases to consider for any specific issue [16].

The next section discusses critic-5's simulation ability in terms of the more general notion of operational specifications.

3.6 Specification evaluation in general

It seems clear that some form of "operationalization" is necessary to find intention bugs in specifications. The rapid prototyping approach provides this by building a source code program from the specification, or using the specification itself if it is executable, and running it against intent. This latter approach often rests on what is commonly known as symbolic evaluation. Cohen [2] and Swartout [19] have built a symbolic evaluator and behavior explainer for Gist that serves as a good reference point. In their system, the user designates a particular Gist action to test. The system then runs the action on symbolic data, and outputs an execution trace. Another system sifts through the trace to summarize the highlights. While this is a useful approach to the problem of handling the often massive amounts of data produced by a run, Swartout notes the following [19]:

> "The current [Gist] symbolic evaluator is not goal driven. Rather than having a model of what might be interesting to look for in a specification, the evaluator basically does forward-chaining reasoning until it reaches some heuristic cutoffs."

Swartout goes on to argue that if the Gist symbolic evaluator had some notion of what was interesting, it could avoid lengthy and unproductive paths. Our arguments in this paper 1) support Swartout's conjecture, and 2) extend it along several lines. First, interestingness for a domain-independent language such as Gist must center on features of the language rather than features of the domain. What we have proposed is to explicitly represent what is interesting about the domain, again part of the knowledge we see an expert analyst having. Second, the goal of the Gist symbolic evaluator is to allow *the user* to test the specification; all the machinery is set up to explain the results of the test. We foresee a need for an active critic of the specification. Such a critic would generate its own test cases to try to poke holes in the current problem description. This distinction between the "attitude" of the two approaches is important. The work on symbolic evaluation to date is based on a view that

the user knows what he or she wants; the problem is making sure the machine has represented it properly. In our view, a user has a sketchy idea of what her or she wants, and has rarely thought out all of the consequences. One of the roles of an expert analyst is to recognize and show the user the ramifications of his or her actions, e.g., adding a sub-branch to a library, extending borrowing limits of certain library users, allowing staff to borrow without restriction, each of which is likely to have difficult to foresee interactions with the existing description. We believe all of this must happen in an interactive environment *tied to the development process*. Thus, symbolic evaluation is not something you invoke after the fact as you would a compiler, but instead should be part of the construction process itself, e.g., integrated with the editor or elaboration system. Given this, we see the following roles for operationalization in a specification system:

1. The role of *validator*, as used by Swartout [19] and Cohen [2]. Here, *the user* selects test data and an action or actions to be confirmed; the system executes the action, and presents the results.
2. The role of *validator*, as used in section critic-5. Here, *the system* selects test data and an action or actions to be confirmed by the user.
3. The role of *refinement driver* that checks for missing details, e.g., "Let's suppose that x has occurred; how do you want to handle it?". This type of testing is initiated by the system to fill in holes in the current problem description (the DESIGNER system plays a similar role in algorithm design [17]).

Critic-5 attempts to take on the second role. For the first role, while the user can set up and run a specification test in our system, we currently do not provide the analogue of Swartout's explainer; the user must interpret the results manually. To address the third role, we will need to more tightly link our critic with the development system itself, possibly allowing the critic to suggest high-level editing commands for overcoming some confirmed specification bug. While each of these roles come into play during a complex development, we reiterate that it is solely the second role that we have addressed to date.

4. Conclusions

We have followed the evolution of a computer-based critic of software specifications. We have shown, at least informally, that such a critic is able to produce the same type of critique as an experienced analyst that we have studied. In constructing our critic, we have come to conclude the following:

- A critique of a specification must eventually rest on the goals, the constraints, and the limitations in force. There are no bad specifications, only ones that do not support a given set of policies.
- Effective critiques must be presented in domain terms. In particular, while representation-specific but domain-independent critiques are useful, we found that human analysts that are expert in a domain concern themselves more with the interaction of the system with its environment, e.g., the typical and atypical ways the system will be used. This requires a domain-specific (A.K.A. expert system) approach to critiquing.

- If the specification language is operational, then so to should be the critique. A major advantage of operational specification languages is that the user can run his or her own test data to validate intent; clearly a critic will want to take advantage of the same ability.

We can also list a set of open issues that we have either finessed or ignored in our critic:

- The matching problem. An expert system approach says that we have a knowledge-base of interesting cases to look for. How will we recognize these cases in the specification being constructed?
- The generation problem. To what level must a critique be given, i.e., how sophisticated is the client with the problems that can crop up in the domain of the system and environment?
- The control problem. When are critiques most effectively delivered during the *incremental* construction of a specification? As a related question, can critiquing be used to drive specification construction?

Frankly, we see the matching problem as one that will continue to be problematic into the foreseeable future, and hence one that must rely on finesse, e.g., restricted languages, manual intervention. The generation and control problems are also knotty in their full generality. However, even our small study of expert analysts [9] pointed towards a crude model of critiquing through examples; we would expect that a more focused investigation would allow a partial but useful theory to be developed.

Acknowledgments

We would like to thank Sarah Douglas and Barbara Korando for their assistance in carrying out our protocol experiments. Comments by John Anderson, Bill Robinson, Martin Feather, Sol Greenspan and Jack Mostow on an earlier draft of this paper are appreciated. We would also like to acknowledge the support of the National Science Foundation under grant DCR-8603893.

References

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

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

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

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

[5] 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

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

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

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

[9] 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

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

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

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

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

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

[15] Mostow, J., Voigt, K., Explicit incorporation and integration of multiple design goals in a transformational derivation of the MYCIN therapy algorithm, AI/VLSI Working Paper 43, Department of Computer Science, Rutgers University, 1987

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

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

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

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

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

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

Appendix A

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.