A Broad Spectrum Toolset for Upstream Testing, Verification, and Analysis

Susan L. Gerhart
MCC Software Technology Program
9390 Research Blvd., Austin TX 78759
512-338-3492 gerhart@mcc.com

1 Introduction

The science of testing, verification, and analysis relies on formally defined objects to which mathematically-based techniques are applied. In the upstream of software development where objects are ill-defined, partially defined, or otherwise fuzzy, the corresponding notions of testing, verification, and analysis are harder to envision. We use the term "upstream" to encompass all the problem-solving activities involved in determining requirements, evaluating alternative design approaches, assessing evolutionary possibilities and other project risks, and producing the artifacts (the design) which control the downstream where, in principle, implementation concerns are dominant. Even where formal requirements languages are used, the uncertainty of what is being expressed, plus whether it is what is really wanted, require different analyses.

Section 2 briefly describes the MCC LEONARDO project, which is proposing and experimenting with new ways to assist upstream software development. Section 3 rationalizes case analysis on sequences of events as a broad spectrum upstream activity. Section 4 goes through an example, emphasizing the process model for use of the toolset, SEQIFY, which is discussed in Section 5.

2 The LEONARDO Context

The essence of the LEONARDO vision is (1) that all software development is a mixture of informal and formal activities, but particularly so in the upstream, and (2) that computer-based cooperative work modes provide new and more effective ways of organizing technical workers (architects, physicians, and engineers, as well as software developers). LEONARDO is envisioned as a primary environment for supporting upstream technical work in the 1990's.

How can inherently fuzzy thinking on yet unformalized objects be made more formal? One answer lies in the proper combination of structured and unstructured objects. For example, hypertext webs can be highly structured at the level of links and attributes of objects, yet nodes can contain unstructured material. The Software Technology Program (STP) has adopted a model of upstream design conversations, [1,2], based on the fundamental units: issues, positions, and arguments. Issues are questions to be addressed, positions are answers, and arguments are reasons. Another semi-structured mode is scenarios, [3], which are realistic but fictional accounts of how an envisioned system will be used, with all the glorious context of characters, motives, mishaps, plots, and morals. Yet another semi-formal approach, [4], defines role-activity models with sets of tools and interactions so that computer-supported cooperative work becomes thinkable. See also [5,6,7].

We expect the output of upstream development to be some kind of formal specification from which the downstream will proceed. However, formal specifications entail too much detail to be tackled too early in the upstream where issues still abound and the system vision is unclear. This paper presents a conceptualization of upstream systematic testing, verification, and analysis without compromising the servicing of less well-defined objects and while stimulating the creativity whereby new systems are conceptualized.

3 Rationale for the Approach

3.1 A Sequence View of the World

The common thread for crossing the pontoon of upstream development activities is our old friend, the sequence. Sequences are widely used as abstract data types, e.g. the Affirm system [8,9] had a library of sequence operations and lemmas. The theory of sequences is used in communication protocol modeling [10], in ghost variables describing histories of events of programs, and in avoiding the nasty array properties that lead to explosive case analysis in verification conditions. Using the standard constructor-modifier-selector model of abstract data types [11], we can alternatively view the data objects as the result of a sequence of events that add to or modify the object while selectors (observers) look back through the history of events constructing the object until they find the combination of events that provide the needed information.

The generalization of axiomatic methods from abstract data types to transition systems has been explored in [10,12]. For example, in a library [13] with events to check.in and check.out and add and remove books, we consider the sequence of these events rather than distinguish some events as constructors and others.
arious human calculations must be avoided, in either generating good cases rapidly and in analysing the cases.

Our proposed system is called SEQuIFY to connote both verifying (in the general sense of confirming or substantiating) properties of sequences and the world view of seq-ifying everything.

3.2 Case Analysis

Several tenets underlie the SEQuIFY methodology.

Intuition-based testing works. We (=the author) believe in testing the old-fashioned way: primarily through intuition. Though there is scientific progress in identifying the fundamental difficulties of testing and suggesting as remedies various combinations that yield more reliable results, these approaches do not fit well with the upstream. As observed earlier, the necessary formal objects are not present, leading us to suggest sequences as the fundamental formal object upon which to base upstream analysis. We also believe that excellent testing results can be achieved from a pragmatic approach: good error guessing, focusing on the most important properties, and recognizing that the results of testing must be efficiently scrutinized to identify errors. In particular, we believe that the essence of good testing is going after difficult combinations of conditions, where reasoning breaks down at both abstract and concrete levels.

Case analysis has many uses. Testing is just one way of using the more general approach of case analysis. First, a good case demonstrates many kinds of behavior that a user/designer will need to think about sooner or later. Second, in the upstream, we see testing primarily as a way of raising issues, not of dividing correct from incorrect. Third, deciding on the testing criteria and devising the test cases is another form of requirements analysis: what are the properties of the system and how will they be accounted for? Fourth, case analysis is a complementary form of reasoning to induction where proving methods excel.

Fast TurnAround is essential. In the upstream, there's a need to see a wide spectrum of behavior. It is important to generate good cases rapidly and in such a way that analysis of them will also rapidly lead to significant and understandable issues. Laborious human calculations must be avoided, in either generating or analysing the cases.

Following from the above reasoning, we focus on case analysis as the upstream analog of testing, verification, and analysis. The objective of upstream case analysis is to provide a range of specialized instances for scrutiny by people with knowledge of, plus a stake in, the product being developed.

4 Example: A Synchronization Package

4.1 Context

The Threads Package [14] applies a formal specification method (the Larch two-tiered approach) to an existing package of synchronization primitives. The primitives are specified in an extended pre-post-condition format using simple notions of sets of processes (threads). The primitives include requests to acquire and release resources, wait for and signal conditions, and generate exceptions to alert other processes of relevant behavior. The novelty of the package is that several restrictions on primitives have been loosened for efficiency sake, leading to conventions in practice, which are explained in text accompanying the pre-post specification. Figure 1 gives a fragment of the specification dealing with the acquisition of resources; omitted are the exception handling parts of the package specification. A mutex (short for "mutual exclusion") allows threads to cooperate on access to shared variables.

Our objective in this paper is to make this traditional, "static" specification into a more executable form, generate instances of calls of the primitives, and investigate the properties of sequences of those calls. This provides testing of the specification by deriving an alternative executable form, as well as the opportunity for further using the specification in an instructional way.

begin(cavaats) Note that we are NOT proposing a tool for debugging concurrent systems, but rather an approach to illustrating and analysing a specification for which a multi-processing example has been chosen. NOR is this example typical of all upstream analysis in that a specification already exists — the process and toolset are the main focus of the example. Deviations of our model from the intentions of the specifiers [14] are yet to be resolved, but are inherent in the process.

Briefly, we will derive an executable specification from the pre-post one by defining a notion of legal sequences and observable properties. A legal sequence of events observed on the Threads Package consists of a possible subsequence of atomic procedure calls from routines using the primitives, augmented with the actions of a system scheduler (to grant a resource to a thread awaiting it or to awake sufficient processes when a signal is generated). For example, it is legal for a release event to occur for a thread and a mutex only if the thread actually holds the mutex.

holds.mutes(T:thread, M:mutes) is an example of an observable property. One specification in axiomatic terms has the form of the fragment in Figure 2. Note that holds.mutes is well-defined only for legal sequences and that it optimizes a bit, e.g. it uses the fact that legal guarantees holds.mutes when a wait occurs and that it should be invariant that only one thread hold a mutex. Now, what kinds of questions can we ask about a particular event in a possible schedule of primitives:

- what makes that event legal? is it that some preceding event occurred or that some invariant can be proved?
- checking legality, is there anything in the sequence that would make the event NOT legal?
- what other events would have been legal at that point?
- which threads hold which mutexes (the resources)?
- what sequence of further events would lead all current threads toward their desired end states?
The sequence specification is subject to invariants, e.g.

\begin{enumerate}
\item \caption{Pre-Post Specification Fragment}
\begin{align*}
\text{hold}_\text{mutex}(T,M) &= \text{false}.
\text{hold}_\text{mutex}(T:\text{request}(M)|S,T1,M1) &=
\text{hold}_\text{mutex}(S,T1,M1).
\text{hold}_\text{mutex}(T:\text{acquire}(M)|S,T1,M1) &=
\text{not}(M=M1) \text{ or } \text{hold}_\text{mutex}(S,T1,M1).
\text{hold}_\text{mutex}(T:\text{release}(M)|S,T1,M1) &=
\text{not}(M=M1) \text{ or } \text{hold}_\text{mutex}(S,T1,M1).
\text{hold}_\text{mutex}(T:\text{wait}(M,C)|S,T1,M1) &=
\text{not}(M=M1) \text{ or } \text{hold}_\text{mutex}(S,T1,M1).
\text{hold}_\text{mutex}(T:\text{signal}(M,C)|S,T1,M1) &=
\text{not}(M=M1) \text{ or } \text{hold}_\text{mutex}(S,T1,M1).
\end{align*}
\end{enumerate}

\begin{enumerate}
\item \caption{Sequence Specification Fragment}
\begin{align*}
\text{ATOMIC PROCEDURE Acquire}(\text{VAR } m : \text{Mutex})
\text{ MODIFIES AT MOST } [m]
\text{ WHEN } m = \text{NIL } \text{ENSURES } m_{\text{post}} = \text{SELF} \\
\text{ATOMIC PROCEDURE Release}(\text{VAR } m : \text{Mutex})
\text{ REQUIRES } m = \text{SELF} \\
\text{ MODIFIES AT MOST } [m]
\text{ ENSURES } m_{\text{post}} = \text{NIL}
\end{align*}
\end{enumerate}

Figure 1: Pre-Post Specification Fragment

Figure 2: Sequence Specification Fragment

Such proofs verify that the terms used in the specification are well-defined but they also extract unsuspected invariants which often later turn out to be unexpressed requirements. Specifications alone rarely tell the full story; their derived properties - the theories of the specifications - are often valuable.

Supposing we could generate many sequences imitating threads calling these synchronization primitives under the coordination of a scheduler which yields legal histories, what could we do with them? Well, the first question is: what makes a sequence interesting? e.g. is it important to understand those histories of the form: (1) a resource is requested, (2) acquired, (3) held while the thread waits for some condition to be satisfied, (4) released, (5) used by another thread, and (6) then re-acquired? An approach is to generate many variations of such a scenario. For example, to make this sequence of events happen according to the specification, one thread might have to block another from getting the resource before it, thus producing a much longer chain of events than expected. It would be especially useful if we could just describe some generally interesting patterns of events and then force them to have observable properties. To even see such a long scenario unfolding, we might want to display, as in Figure 5, the scenario broken down by thread and animate the effects of a mutex being passed back and forth among threads.

We'd like \textit{SEQIFY} to aid in the generation of interesting sequences from sequence specifications, in the portrayal of their dynamic characteristics, and in the capturing of issues about all the dependent artifacts - the original specification, user manuals, code, designs, code test cases, etc. Many kinds of research questions will arise:

- how is legality defined?
- what kinds of observable properties yield information about both the sequence and related specifications?
- how can threads and resources be visualized so that future users of this package will rapidly glean information?
- can invariants be portrayed in the sequence of events?
- can sequences of troublesome calls be extracted from programs using the primitives and studied using the tools?
- can useful sequences of synchronization primitives be generated to test tricky parts of the implementation?

4.2 The SEQIFY Process

Let's use a LEONARDO method to provide a high-level description of the SEQIFY process. Very simply, we write a "job description" for each specialist on the Case Analysis Team (SEQIFY may be only one of several tools used). Remember that a specialist is a person with a specific set of skills, a tailored set of tools, and a particular mindset, all of which improve their ability to do specific tasks. If the tasks fit well together, the overall objectives will be achieved by a team of such specialists (as opposed to a team of undifferentiated software engineers). Figure 3 describes a set of technical roles appropriate for a team of analysts and Figure 4 shows the interaction of the team (oval=role, box=SEQIFY component). Of course, the roles on the Case Analysis Team need not be filled by separate individuals, nor are they envisioned as full-time jobs. For example, the Coordinator may be another specialist, a liaison from another design team, or a QA-type person. The process model serves as a set of requirements from which more detailed specifications are written for what each specialist needs and does.

Suppose that the context is an upcoming review of the Threads specification by several parties who have a stake in it, e.g. implementors and eventual users. The objective of the review is to raise issues primarily about the specification's form and content and secondarily about implementation and usage. The specification as it appears in [14] is given to the Case Analysis Team. Here's what each role does:

\textbf{Modeler}

The objective is to define useful test sequences of primitive calls which are consistent with the pre-post form of specification and likely to raise important issues about it. The following \textit{observables} characterize the behavior of the Threads Package, where types are \(M: \text{mutex}, T: \text{thread}, C: \text{condition}\):

\begin{align*}
\text{holds}_\text{mutex}(T,M) \text{ tells whether } M \text{ is held by } T \\
\text{wants}_\text{mutex}(T,M) \text{ tells whether } T \text{ has requested but not acquired by } M \\
\text{waiting for condition}(M,C) \text{ tells whether } T \text{ has suspended, awaiting } C \text{ associated with } M \\
\text{active}(T) \text{ tells whether } T \text{ is able to request resources, i.e. not waiting for a condition or a mutex}
\end{align*}
Sequences are calls by processes with effects of a scheduler,
1:   tl requests ml
2:   tl acquires ml
3:   t2 requests ml
4:   tl requests ml
5:   t3 requests ml
6:   tl acquires ml
7:   tl waits for condition c
8:   t3 acquires ml
9:   t3 signals condition c
10:  t3 awakes
11:  t3 releases ml
12:  t2 acquires ml

One interesting behavior is threads waking up and getting a mutex but finding that the condition waited on has been invalidated by another thread. (Several interesting behaviors have been taken directly from the article [11].) Another is the important distinction that Broadcast awakens all waiting threads but Signal awakes, usually, only one and the various assumptions about what the waiting condition achieves.

A legal sequence is essentially a feasible schedule for some sequence of calls by threads. Thus a scheduler must be modeled, although the model will not constitute its specification. As with most models, some simplifying assumptions will be made that hopefully will not mask any problems of the pre-post specification or confuse the reviewers with too much detail. The scheduler will grant some number of Acquires after every primitive call to threads that want some free mutex and it will schedule an appropriate number of Awakes after every Signal or Broadcast.

Examples of conditions identified for testing are: one thread holding a mutex when another one wants it, a thread holding more than one mutex, more than one threads waiting on a condition,... These conditions are expressed in terms of the observable properties and equalities and bounds constraints. The testing strategy is to generate sequences for all 2^N combinations of these conditions. That schedulers have been known to fail when there is no work to do or when only one thread is doing anything is justification enough for considering seemingly trivial cases. This combinatorial process will also check out the sequence model and its implementation. Only a few really interesting sequences will be chosen for use in the review, but this is one way of generating possibilities with less bias.

Model Coder

Each observable property and condition for testing will be coded in a standard representation which doesn't reflect whether the intended use is to check whether a sequence has a given property or to generate a sequence that has that property. The form is similar to the axioms in Figure 2. This representation is the input to a Property Interpreter (along with some search control parameters) which manages the generation process. Some additional simplification rules are required. The code itself is tested by generating all possible ways of making each condition come true, including legality, on sequences of limited length (although this is a long overnight run). Each schedule generated is checked by a simpler form of the model which is only good for checking. This still leaves open the possibility that some feasible sequences were not generated.

The Coder (perhaps cooperating with a Prove role) also checks out the original model by inductively proving some simple invariants; e.g. ThreadsAccountedFor. The model also should stipulate that no mutex is held by more than one thread and that no thread both waits and holds a mutex. But the latter really is more of an issue than a rule: the definition of legal sequence doesn’t allow it but what does the specification say and what should the implementation do if a thread requests a mutex it already holds? Does it make sense?

The Coder sets up a generation process driven by two kinds of requests: (1) given a sequence of calls on primitives and a set of properties, generate a schedule for the calls satisfying the properties and (2) given a sequence and a set of properties, treat it as a schedule and generate the call sequence of which it is a schedule, provided the properties hold. Usually the Generator role will just input an uninstantiated sequence of a certain length and let the Property Interpreter find the events which satisfy the properties.

Sequence Visualizer

The Visualizer’s job is to lay out windows so that the Case Generator and End Analyst can get good feedback from running the tests. See Figure 5. The Visualizer uses a standard layout and writes some simple programs to decompose a sequence into subsequences of events for each thread. The visualization is based on the mutexes moving around the sequences whenever an Acquire or Release or Wait occurs. The thread events are laid out in a line with a header containing the name of the thread. Each event and mutex is shown by an icon with its identification attached and icons are symbolic, e.g. request = key, acquire = closed lock, release = open lock, mutex = safe, wait = hourglass, awake = sunrise. The observable properties, e.g. wants_mutex, are shown by links between the thread header node and the moving mutex. Thus at any point in the sequence, the status of observables is obvious from the picture. A dual view is prepared where the threads move through the mutexes.

Case Generator

This role produces a table shown in the left window of Figure 6, for combinations of conditions. The table is represented as a tree (top branch true, bottom branch false). The End Analyst and Generator can view the tables as trees or in tabular form, popup the cases and move them to the Animator, access explanations prepared by the Coder for model components, build many tables to test various combinations of conditions and generally learn the characteristics of the problem while producing cases.

End Analyst

The End Analyst role interfaces the sequence model with the original specification, using a panoramic design setting as in Figure 6. The left window contains cases produced by the Case Generator and three central windows contain animations prepared by the Visualizer. On the right is a window into the Analysis Issue Base in which to record issues about the specification and/or the model. Some issues have already been added by prior Case Analysis Team members, e.g. about requests for a mutex already held or about how significant a simplification it would be to consider only one condition per mutex or why the term “enqueue” is used when there isn’t really any queue in sight.

The End Analyst selects some interesting cases that will be studied for a while and some that might be walked through at the review. Sequences are picked up from the case table, moved to the sequence window and then prepared for visualization. A VCR-like controller as well as a slider control allows the End Analyst to play forward and backward through the sequence to observe its effects. Each sequence has been created (that’s why it’s in the Case table) to have some specific combination of conditions that the Modeler thought was interesting and these guide the End Analyst in sampling interesting sequences.

The End Analyst finds some suspicious aspects of legality, e.g. threads getting blocked after a Signal when they seemingly should be active again. A specific sequence is concocted that seems to contradict the blocking and the property Active is given to the Property Interpreter for checking. It rejects the sequence as either illegal or not satisfying the property, but the End Analyst can’t see why the sequence is not legal. When the review
meeting comes around, this issue will be considered by the specifiers and the animation will be revived in the context of a meeting held in the panoramic design setting.

Case Team Coordinator

The Coordinator investigates each issue (or delegates the investigation), tracks them through to other issues, and assures a resolution (at the appropriate time) of each issue. For example, the Coordinator is responsible for alerting the Coder and Modeler to the possibility that the End Analyst’s blocked waiting issue might be their error. Before the review, they address the issue (as expressed in the Analysis Issue Base) and record their positions and arguments. The Coordinator also sits with the End Analyst for a while and acts as a surrogate reviewer to assure that the right visualization effects, e.g., speeds of playing through and layout, are set up to avoid wasted time in the review meeting. The Coordinator is responsible by merging the Analysis Issue Base with the Design Information Base by connecting issues and making sure issues are expressed in terms intelligible to other designers.

5 Description and Status of the Toolset

Architecture

SEQIFY is a collection of Prolog modules written in Sun Quintus Prolog, making extensive use of the Quintus library. In the SEQIFY architecture, each component has a general part (case table manager, property interpreter, and sequence controller) and specific parts (conditions and generators, problem-specific properties, and special effects windows). We are still experimenting with the separation of general and specific represented as either interpreters or schemas. Interpreters are fixed general components, although their control patterns may be modified by the specific parts, whereas schemas are expected to be instantiated with the specific relations and then transformed somewhat for efficiency. This is a common situation in Prolog since a slight difference in ordering of procedures may lead to massive differences in the search characteristics of the program and, indeed, the procedures may not be finitely interpretable if variables are insufficiently instantiated.

Process specifications for tools are expected to be derived from the role descriptions of the Case Analysis Team Process Model of SEQIFY, which incorporates also the mindsets of the specialists which are expected to be supported. Experiments are being performed by the author playing each role of the Case Analysis Team. SEQIFY is expected to be used with other evolving LEONARDO technology, e.g., an issue management tool implemented over a general hypertext substrate [15]. The verification technology will be ProAffirm, as described in [9].

Interfaces

The window manager is a new (full release in 1988) product called ProWindows, licensed by Quintus from ICL after development by Anjo Anjewierden at University of Amsterdam under Esprit project 108. ProWindows imposes object-oriented structure on the SunView window manager, providing a myriad of graphical and dialog classes that interact with each other and with Prolog through sending messages. The primary interface for controlling the “execution” of sequences is a VCR-like controller1 with an alternative slider-driven interface (the two can be used together). The VCR allows the obvious forward and backward movement with change of delay (period between events which the user may need to absorb the effects of a transition) and stopping and starting. The slider controller allows setting numbers of steps to move and between reporting of results, as well as delay. Switches also allow resetting the origins, e.g., going back to the start. A panel of buttons allow invoking an issue entry tool with issues specific to sequences and in the context of the sequence being analysed, a scenario entry text window with a template of good questions for discussion, and a short dialog for printing the entire Sun screen.

Both controllers operate off a script generated by the Visualizer who lays out the events and objects, prescribes when objects should move and where, and defines other effects for each step of the sequence. Usually these are programmed, but special effects may be created for specific interesting sequences since the script interpreter is extensible.

The VCR works smoothly for short sequences, however the slider interface may prove more useful for longer sequences or for boring stretches. As always, screen space is a problem since the panoramic design view of Figure 6 is not the same as overlapping windows. Script generation to date has only used a simple layout - subsequences pertinent to a single object laid out in lines with other objects moving to the relative positions of events in each subsequence. Additional ways of demonstrating relationships among objects and events should be explored. The ability of the animation to retain user interest and provoke attention toward important issues remains to be seen. However, we are confident that the animation will prove useful for “walking through” customers and for interactions among various design specialists.

Property Interpreter

A Property Interpreter is the key component. Logic programming allows more flexible programming modes than procedural languages, although achieving its full capability can still be quite challenging in Prolog. In our threads example, we can input a sequence of primitives and get out a legal schedule and then test whether (or generate alternative sequences for) observable properties hold or we can input the desired properties and generate a schedule which satisfies it and retrieve from that the sequence of primitives for which that schedule holds. Input and Output are reversible, in principle, in logic programming.

In general, the Interpreter is given a specification of legality and of each property, a set of properties, and a fixed length sequence (to control the backtracking) and it produces a set of sequence-constraints pairs. The correctness criterion is that, if allowed to run to completion, the Property Interpreter will have produced all possible sequences such that if their associated constraints are satisfied the properties hold. Directly expressing property definitions as individual Prolog rules was found to be too error-prone (it’s hard to avoid variables getting bound prematurely), so a representation of standard forms of these definitions was devised. This representation is easy for encoding specifications, but it has not been determined which sequence specifications can be written without contortions.

A version of constraint programming [16,17] may be better than pure logic programming, but the key opportunities in using its generative power can be realized in today’s Prolog. Our current version of the Property Interpreter handles constraints of limited forms: equalities, inequalities, bounds, and unsatisfied properties before the start of the sequence. A simplifier removes many unsatisfiable constraint combinations and makes the constraints readable, but some constraint combinations are simplifiable only through invariants on the specification.

1I'm indebted to Don Petersen, former DEC assignee at STP, for showing me that a VCR-like controller connotes exactly what I wanted — user control over the movement of an action-oriented sequence of events that the user wants to see in more or less detail.
Case Manager

The Case Manager component maintains a table of cases, which are combinations of properties and the data which satisfy them, if any. The SEQ,IFY Case Manager interface is not complete, awaiting better understanding of the Property Interpreter. Previous experiments with this component used a straightforward (in Prolog terms) representation of cases as clauses which paired a truth vector with a result that satisfied it, e.g. `case(tftl,[e1,e2,e3])`. Experiments described in [18] showed that logical interdependence of the conditions was best handled by explicitly defining those combinations which should be excluded.

The Case Generator role will select from a menu of defined properties those to be combined into a table. The table contains all $2^N$ truth combinations of properties, although many may be marked as excluded. Often there will be a set of sequence-constraint pairs to select from and the constraints will not be totally simplified. The Generator will be responsible for the selection and for the proofs that constraints are satisfiable.

Applications

The Threads example as well as an employment agency data base [8] and the VCR specification have each been driven through part of the SEQ,IFY process, e.g. the Visualizer and Coder for the Threads Package. However, most of the scenario in Section 4.2 is still more guide than reality. An internal STP project — part of a coordination system — is a live application getting underway now. We believe that we have a sufficient set of motivating examples to validate the SEQ,IFY concept and to evolve its components to sufficient generality.

6 Related Work

The Interrogator [19] is one of the inspirations for SEQ,IFY. It shows that carefully constructed models of protocols can be engineered in Prolog to uncover otherwise probably undetectable situations, e.g. penetrations of an executing protocol. The trick is to properly describe a desired (or undesired) final state to a Prolog expression of the model so that little user intervention is required to construct a scenario of events leading to that final state. Prolog has been proposed for many testing situations, e.g. [16] shows how specifications can serve as the basis for test cases. The technical problem here is that logic programming is probably not the best paradigm for these situations. [17,20,21] provide more general contexts.

Kemmerer [13] founded a school for studying properties of specifications from both informal and formal perspectives. The library example's state transition and constraint model of specifications, based on INA JO and other formal specification methods, illustrates a fundamental pattern of specifications. Many papers in the specification field have objectives related to ours: [22] uses scenarios to identify clusters of requirements which lead to sub-system partitioning for design purposes; [23] describes a problem-solving approach to general domain analysis for resource-centered systems such as the library; the MIT Requirements Assistant [24] applies an object-oriented approach to the same domain. We envision systems such as SEQ,IFY complementing the object focus of these approaches with dynamic information.

The NRL approach [25] shows the kinds of issues we want to see forced out by formal scenarios. Other kinds of issues arise from the formalisation effort itself, [26]. [27] comes at specification from the approach of database theory, attempting to move such relationships as integrity constraints to proofs at the trans-

action level. SXL [12] is a cleanly worked out executable specification language using Prolog-ish formalisms, but it faces the general questions: what should be executed and how should the results be used? Earlier work on path expressions is possibly relevant, but may be more detailed than our scenarios.

Comprehensive specification approaches seemingly compatible with ours are suggested in [28,29,30], for example generating traces and views of distributed systems. We expect to use these as significant theory upon which to develop further versions of SEQ,IFY.

While our animation capabilities are still primitive, they are inspired by the sophistication of algorithm animation [31,32,33]. The architectures, e.g. Balsa-II, may carry over to SEQ,IFY but the problems being attacked are different.

7 Summary

This work has been motivated by the desire to (1) bridge the informal-formal gap that the LEONARDO project is directly attacking; (2) embed new testing, verification, and analysis activities in an issue-based context; (3) demonstrate the increasing applicability of logic programming paradigms and technology; and (4) suggest the compromise of sequences, where we believe that many useful models can be built to investigate still fuzzy systems under design.

We've spent more time describing the context than the tools, partially because the tools are still evolving rapidly. But we also believe that one barrier to acceptance of formal methods in industry has been the excessive emphasis on formality as an esoteric activity and for its own sake. We take the position that verification, testing, and analysis are natural activities when pursued in combination and in moderation. No attempt to push everything into a "prove it or else" frame of mind will succeed in the upstream (though it is exactly right for many downstream problems). Selective testing isn't ever very convincing, even in the downstream, so we advocate systematic, extensive case analysis as the fundamental activity, especially in counter-point to proving.

We've tried to show how several common sense notions apply. Issues can be the driving force, instead of evaluations of correctness or incorrectness, and issue management can become a systematic activity. We've used "job descriptions" as the top-level process model for the system we are designing. Scenarios (yes, even as in the movie business) can be written about specified objects to put them in the proper context, to stimulate the viewer's imagination about the object, and to bring out the important features to be investigated. As video technology enters the workstation world over the next few years and as the distinction between the real system and the envisioned system becomes blurred, there will be many opportunities for applying and supporting the old ideas of testing, verification, and analysis.
8 Conclusions

Throughout this paper, we've deviated from old paradigms by suggesting that seeking the "golden" test case, the "iron-tight" proof, or the "most powerful" analysis should not be the objectives in the upstream, though they grow increasingly important as formality increases. We believe that issues arising from fuzziness should be resolved in the upstream and that the downstream should be totally a matter of formal analysis and of investigation of trade-offs. The carry-over from the older world is simple: whatever is done must be done systematically and it must be recordable. So, the challenge of testing, verification, and analysis in the upstream is to find ways of making the processes more systematic and to make recording them more compelling. We propose (1) the issue-based communication model to address both challenges and (2) case analysis applied to sequences as the analog upon which to base further research into upstream testing, verification, and analysis.

References


**Case Generator**
Uses specialized tools (operating with the code produced by the Coder) to produce tables of data satisfying cases with an eye toward how they will be used. Determines combinations of conditions that can't be satisfied, interacting with Coder and Modeler where condition meaning is unclear. Records rationale for choosing generators for use by future Case Generators.

**End Analyst**
Selects or combines cases for intended use. Uses visualization methods to identify important issues in cases and to check for obvious errors in the model. May write scenarios describing important cases for other designers to use. Contributes substantially to the Analysis Issue Base as cases are examined. Works closely with Coordinator and Modeler to assure that case analysis objectives are achieved.

**Case Team Coordinator**
As specifications and requirements change, maintains order among versions of case tables and model and animation code, plus alerting other Case Specialists when their services are again needed. Checks interfaces among specialists to assure the right kinds of material are being passed. Monitors issue bases for important early issues for other designers and to assure objectives are on track. Also trains new members of the Case Analysis Team.

Figure 3: Case Analysis Process Model: Roles

**Modeler**
Establishes objectives of the case analysis exercise. Extracts interesting conditions from requirements material, designs, etc. and formulates them in English or some domain notation. Outlines the sequence model: definition of legal, conditions to be combined in cases, invariants to be proved, ways that sequences might be visualised. Decides and records rationale for important and interesting conditions. Raises issues about what is important in specification/design, what terms mean, why certain events are occurring.

**Model Coder**
Formulates the specification for input to the Property Interpreter, probably using special domain library notations. Tunes generators for those kinds of conditions. Proves invariants of the model. Figures out exclusions, i.e. inconsistent combinations of conditions. Feeds back questions or issues to the Condition Modeler to help flesh out the sequence model.

**Sequence Visualizer**
Writes interfaces and view extractors to pick out important features of sequences occurring in cases. Uses standard toolkit for building animations, laying out decompositions of sequences (e.g. matrices, trees, etc.), and producing paper as well as screen recordings of cases. Usually acts as a user interface design specialist for several projects. Tries to find ways to surprise the other team members about properties of their specified sequence model.

Figure 4: Case Analysis Process Model: Interactions