Specification Reformulation During Specification Validation

Kevin M. Benner
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
(310) 822-1511
Benner@isi.edu

Abstract

The goal of the ARIES Simulation Component (ASC) is to uncover behavioral errors by "running" a specification at the earliest possible points during the specification development process. The problems to be overcome are the obvious ones – the specification may be large, incomplete, underconstrained, and/or uncompilable. This paper describes how specification reformulation is used to mitigate these problems. ASC begins by decomposing validation into specific validation questions. Next, the specification is reformulated to abstract out all those features unrelated to the identified validation question thus creating a new specialized specification. ASC relies on a precise statement of the validation question and a careful application of transformations so as to preserve the essential specification semantics in the resulting specialized specification. This technique is a win if the resulting specialized specification is small enough so the user may easily handle any remaining obstacles to execution. This paper will (1) describe what a validation question is, (2) outline analysis techniques for identifying what concepts are and are not relevant to a validation question, and (3) identify and apply transformations which remove these less relevant concepts while preserving those which are relevant.

Introduction

Validation at the requirements level is often characterized as validation with respect to the client's or stakeholder's intent. The goal of specification validation is to identify those aspects of the specification which do not conform to the client's intent and then to make appropriate changes. More practically, this boils down to uncovering bugs in the specification and fixing them. The goal of this work is to address a specific subclass of specification errors that have not previously been satisfactorily addressed. In particular, this work addresses identifying errors in the dynamic behavior of a high-level specification. This work will distinguish itself from related works by being able to handle large, very-high-level specifications. This is done by making explicit specific validation questions which focus validation activities sufficiently enough so that traditional validation techniques, like simulation and direct execution are tractable.

The problem of identifying errors in the specification and the cost of finding these later during the development process is well documented [Boehm, 1981]. Among these errors, the most difficult to identify early on are those which concern behavior. In general these include: (1) inconsistency between specification components, (2) incompleteness with regard to known scenarios, and (3) inconsistency between requirements and their realization in the specification.

The work describe herein will uncover behavioral errors by "running" a specification at the earliest possible points during the specification development process. The problems to be addressed are the obvious ones – the specification may be large, incomplete, underconstrained, and uncompilable. These problems are addressed via a four step process. First, the validation activity is decomposed into specific validation questions. Second, the specification is reformulated to abstract out all those features unrelated to the identified validation question thus creating a new specialized specification. Third, the specialized specification is executed with the purpose of proving or disproving the validation question. And finally fourth, since the specialized specification was constructed in a disciplined manner, one may now infer the result of the validation question about the original specification.

The general feel of the interaction is more like a debugging sessions, particularly early in the development. The goal is to get something running quickly and easily so as to reveal behaviors implied by the specification and make them accessible to end users and stake holders for early validation (or more likely, early error identification). During a typical validation session, the specialized specification and its validation question are executed. The simulation system using the validation question will guide the execution toward satisfaction of the validation question. When this is

not possible the simulator will point out how the validation question has been violated. The stake holder and analyst will observe the execution. When a validation question is not satisfiable, the analyst will be able to explore the behavior space to understand why this is the case. Appropriate changes may then be made to the specification and the specialized specification construction process replayed. This is then followed by re-execution of the specialized specification. Naturally this process may be repeated.

The abstraction or reformulation process employed during specialized specification construction is the heart of the ARIES Simulation Component(ASC, pronounced "ask"). It relies on a precise statement of the validation question and a careful application of transformations so as to preserve the essential specification semantics in the resulting specialized specification. This technique is a win if the resulting specialized specification is small enough so that the user may easily handle any remaining obstacles to execution. This paper will (1) describe what a validation question is, (2) outline analysis techniques for identifying what concepts are and are not relevant to a validation question, and (3) identify and apply transformations which remove these less relevant concepts while preserving those which are relevant.

The work described in this paper is a component of a larger effort called ARIES [Johnson *et al.*, 1991] which is concerned with the overall task of requirements acquisition and specification development and validation. Requirements may be stated informally and then gradually formalized and elaborated. Validation is facilitated via a variety of graphical and textual presentations. Elaboration and refinement are supported via evolution transformations. Additionally, mechanisms for reuse and concept encapsulation have been provided.

The example used throughout this paper is drawn from the air traffic control domain, specifically behaviors concerning *handoff* — passing control of an aircraft from one air traffic controller to another. Some of the concepts included in the full specification are: control, physical location, sensors, tracks, maintaining tracks, flight plans, aircraft movement, agents within the air traffic control domain, etcetera.

Validation Questions

At the beginning of the requirements acquisition process, many requirements are not easily expressed as abstract, concise, declarative statements of stake-holder needs. Rather at this point, requirements are often more easily expressed informally as a mix of situations and experiences which the stake-holder wishes to have handled by the system to be specified.

Informally, a validation question is any question a user or stake-holder may have about the specified system. It could encompass anything he/she believes to be pertinent. The goal of a validation question is to provide a means to ask these questions. Fundamentally, validation questions are statements of user's or stake-holder's requirements. They are stated in a manner as similar as possible to the way they are manifest in the user's or stake-holder's real world. And they hopefully have little dependence on how these requirements may be realized in the specification. This section will show how these goals are attained by allowing the stake-holder to express his/her requirements via the following constructs: scenarios, to describe partial orderings of states and events using both abstract and concrete concepts, and assumptions, to support the implicit assumptions common in natural language and often used when stake-holders describe their needs.

Consider the following natural language questions. They are the intuitive basis from which we will evolve formal validation questions.

- VQ-1: Does handoff occur before the aircraft moves from its current airspace to an adjacent airspace?
- VQ-2: Once in-route to a particular location, is control maintained throughout?
- VQ-3: Will the system recognize when an aircraft is out of conformance with its flight plan?

Figure 1: Some Informal Validation Questions

The above questions illustrate several important characteristics of validation questions. First, validation questions often implicitly rely on scenarios and assumptions to provide a narrowed context. Second, validation questions often use concrete and/or qualitative instances to focus on specific, relevant attributes of the specified system. And finally third, validation questions often embody some expected interaction that the analyst is trying to stress. The remainder of this section will describe how validation questions are described in terms of scenarios and assumptions.

Scenarios

Scenarios are a partial ordering of events and/or states. They allow one to describe a complex sequence of activities at an arbitrary level of detail without necessarily making commitments regarding their causal relationships.

This section will define the semantics of a scenario with respect to state transition diagrams[1]. The specific semantics is determined by the scenario mode. Alternatives include comparative, restrictive, or prescriptive. The mode is selected by the analyst during formulation of the validation question. Each mode constrains the behavior space of the specification in a progressively more restrictive manner.

[1]Other notations for scenarios are also available and are often used. They are isomorphic with STDs.

- **Comparative** has no effect on the behavior space of the simulated specification, but acts as a watch dog informing the user as to the satisfaction or partial satisfaction[2] of a scenario during simulation. In this case satisfaction of a node determines the current node. The current node is neither necessary nor sufficient to advance to the next node.
- **Restrictive** means that nondeterminism within the behavior space is pruned so that if the scenario may be satisfied it will be. Basically, satisfaction of each node is necessary but not sufficient to advance to a following node. Informally this mode is best described as a procedural invariant.
- **Prescriptive** means that the simulator advances the scenario from one node to the next irrespective of the state of the simulation. More formally, the simulator treats the satisfaction of a node as necessary and sufficient for advancing to the next node. Operationally, advancing to a transition node implies that the corresponding event is invoked. When the event is completed the following state node is made true.

Main validation question The validation question labeled as *VQ-1* in figure 1 is formalized as the state transition diagram shown in figure 2. States *within-s1* and *within-s2* are the qualitative values for the aircraft being in sectors *s1* and *s2* respectively. Transitions *handoff* and *alert-controller* represent the events of the same name and the following states represent completion of these same events. Transitions *m-1* and *m-2* represent any event that would result in the final state *within-s2*. During simulation, ASC will drive this state transition diagram to reflect the state of the simulation. If an illegal transition occurs the analyst will be informed.

The goal here has been to express typical, critical situations that the user wants to be sure are handled in a specific way. This could have been done in terms of concept at any level of abstraction. Typically as the specification moves closer to completion, the validation question may become more complex and may be expressed in terms of lower-level concepts.

Driving scenarios Driving scenarios are prescriptive scenarios, typically used to model actions outside the scope of the current specialized specification. Figure 3 shows two driving scenarios that are used within this example. The driving scenario commits to manage specific concepts. In this case those concepts are *track-position* and the qualitative values derived from *track-position* (e.g., *top-of-block-altitude* and *within-accept-handoff-computed-point-distance*). When a driving scenario manages a concept it supersedes all other specification concepts which attempt to influence the same concept. This information will be used later during specification reformulation.

[2]Satisfaction of a state means the predicate associated with the state is true. Satisfaction of a transition means the event associated with the transition has been invoked. Satisfaction of a scenario means that the states and transitions of the scenario have been satisfied in the order specified by the scenario.

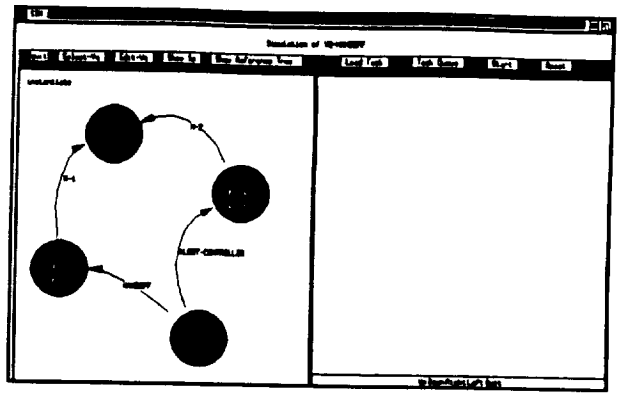

Figure 2: Validation Question – *VQ-Handoff*

```
procedure DS-LEVEL-3()
   := steps(
         insert within-s1 ac1;
         insert top-of-block-altitude ac1;
         delay();
         insert enter-new-airspace ac1)

procedure DS-LEVEL-4()
   := steps(
         insert within-s1 ac1;
         insert top-of-block-altitude ac1;
         delay();
         insert within-accept-handoff-computed-point-distance ac1;
         insert within-s2 ac1)
```

Figure 3: Driving Scenarios for *VQ-Handoff*

The level of abstraction at which driving scenarios operate is one of the primary influences on the level at which simulation will be done. Instead of driving *track-position*, the analyst could decide to drive *sensor-reports*. This would result in a larger, more detailed specialized specification which would include processing of *sensor-reports* into *tracks*.

Scenarios to constrain nondeterminism Specifications are highly underconstrained, particularly early in their development. ASC allows one to execute a specification in spite of this by providing various mechanism to constrain the nondeterminism within the context of a validation question. One of these mechanisms is s restrictive scenario which acts as procedural constraint on the behavior space.

One example of this is the *Handoff* transition in figure 2. *Handoff* is not actually an event but rather a *restrictive* scenario which is constraining the simulation to only consider handoffs consisting of an initiate and accept phase (see figure 4). This scenario precludes handoffs from being canceled or rejected. More

complex scenarios would deal with this after first performing validation on this simpler case.

```
scenario handoff(ac:track)
    := steps[
          automatic-init-handoff(ac);
          accept-handoff(ac, any controller, any controller)]
```

Figure 4: Restrictive Scenario

Assumptions

Assumptions allow the analyst to codify what are often implicit assumptions made by developers as they build rapid prototypes. The advantage of this approach is that it documents said assumptions. Once recorded, the analyst can now separately validated the assumptions with stake-holders within the context of the current validation question. This way the analyst can be sure that assumptions do not trivialize the basic intent of the validation question. A later section of this paper will show how assumptions are used during specification reformulation.

```
invariant FIXED-SET-OF-TRACKS
    for-all (t1:track) element-of(t1, {ac1, ac2})
```

Figure 5: Assumption for *VQ-Handoff*

Assumptions are expressed as invariants. Figure 5 contains one of the assumptions used in the current example. Since we are not validating *track-processing*, we can relax some of the constraints on tracks and for now constrain the number of tracks in the simulation model. *Ac1* and *ac2* are defined as tracks in an unshown initialization scenario.

Influence Analysis

The previous section described how a validation question is formalized. This section will show how the validation question is used to reformulate the current specification into a specialized specification which is simulatable. We begin with influence analysis.

Brooks in [Brooks, 1986] warns that descriptions of software that abstract away its complexity often abstract away its essence. Influence analysis is a means of allowing the analyst to see through this complexity to distinguish between concepts which are most relevant to the validation question and those which are not. Once identified, ASC provides reformulation transformations which remove those concepts which are not relevant.

In general, formally showing whether or not one concept effects another is undecidable. Even at an informal level, causality is a vary hard problem. Influences finesse this issue by relying on rules which are easily computable and which generate all potential influences rather than making claims about actual influences. As such, the resulting influence graph should be considered a conservative representation of concept influences – that is they may indicate influences which are not actually possible, but are safe, in that they will not fail to indicate the presence of an influence that does exist.

Once the initial influence graph is generated, more knowledge intensive approaches are applied to remove many of those potential influences which are not actual influences.

Another problem in extracting the influence graph is that influence paths could be through arbitrarily many intermediaries and in an incomplete specification would be inherently suspect. Rather than deal with this problem, ASC allows the analyst to limit the path length it will look at during analysis. Granted this has the horizon effect, but this can be minimized. (see section Horizon Effect Addressed)

Influence Definition

An influence identifies the conduits through which one concept effects another during execution. ASC divides these conduits up into three classes: information, control, and miscellaneous. This section will informally characterize each of these classes and then illustrate them via an example. ASC has operationally formalized these concepts based on the specification language Reusable Gist [Johnson and Feather, 1991].

- **Information influences** are concerned with the flow of information between concepts. Stated another way, when information changes, how does it percolate through the system? Some examples of these types of influences are: database updates, assignment statements, and definitional use of data declarations (i.e., relations, types, and instances) by other data declarations.
- **Control influences** are concerned with *if* and *when* behaviors may occur. Some examples of this class of influences are preconditions on an event, invocation of an event, invariants, and conditional statements (Note, granularity of influences is at the level of declarations. Thus influences on or by statements are reflected as influences on or by the event which contains the statement).
- **Miscellaneous influences** are concerned with influences on and by the the validation question. Most influences are *onto* validation questions with driving scenarios being the exception.

Figure 6 is the Reusable Gist definition of the event *accept-handoff*. Figure 7 shows a paraphrase of this event. The resulting primitive influence graph is shown in figure 8.

```
Demon ACCEPT-HANDOFF(track,
                     current-controller:controller,
                     receiving-controller:controller)
    precondition controlled(track, current-controller) and
                 handoff-in-progress(track,
                                     current-controller,
                                     receiving-controller)
    postcondition controlled(track, receiving-controller)
    := steps(track.controlled ← receiving-controller;
             remove handoff-in-progress(track,
                                        current-controller,
                                        receiving-controller);
             track.track-status ← 'normal)
```

Figure 6: Reusable Gist definition of the event *Accept-Handoff*

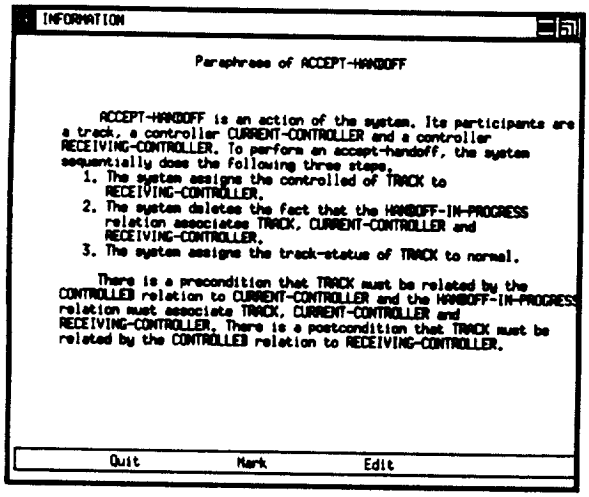

Figure 7: Paraphrase of the event *Accept-Handoff*

The influence graph of figure 8 is most similar to de Kleer's mechanism graph from his work in qualitative reasoning [de Kleer, 1986; de Kleer and Brown, 1986]. The mechanism graph shows the causal influences between concepts. A vertex contains an information value which represents a specific circuit component attribute (e.g. the voltage or current at a given component). Edges represent how a change in a vertex value is propagated to adjacent vertices. Edges are derived from either component models or domain specific heuristics. In influence graphs, a vertex represents a specification concept declaration (or fragment). Edges represent how a concept influences either the behavior or value of another concept.

The influence graph (figure 8) shows most of the immediate influences on and by *accept-handoff*. *Receiving-controller*, *current-controller*, and *track* are parameters of the event. *Enabling-pred-of-accept-handoff* is a composite node representing the precondition of the event. *Controlled*, *and*, and *handoff-in-progress* are relation referenced by the event. Edges within the graph represent the direction in which influences are propagated. Note that *how* each influence effects a given node is not represented. This is in fact outside the capability of influence analysis in ASC. None the less, this still provides a great deal of information to the analyst when creating a specialized specification as we will see later.

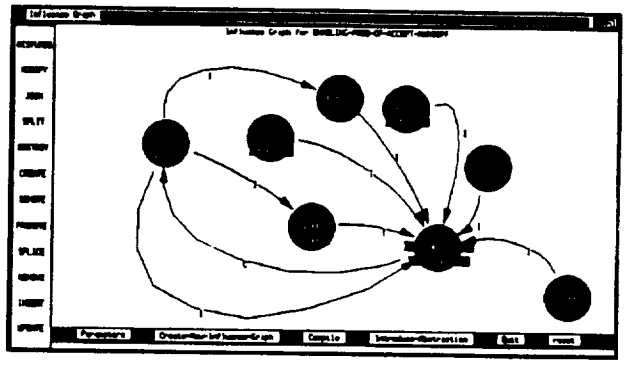

Figure 8: Primitive Influence Graph of the event *accept-handoff*

Automated Graph Abstraction

Though the graph in figure 8 could be used as is, it shows many influences which really do not drive the dynamic behavior of the specification. This section will describe some of the influence abstraction rules which are applied automatically by ASC. Figure 9 shows the resulting influence graph. It is this graph, not the previous one, which the analyst is first shown after influence analysis.

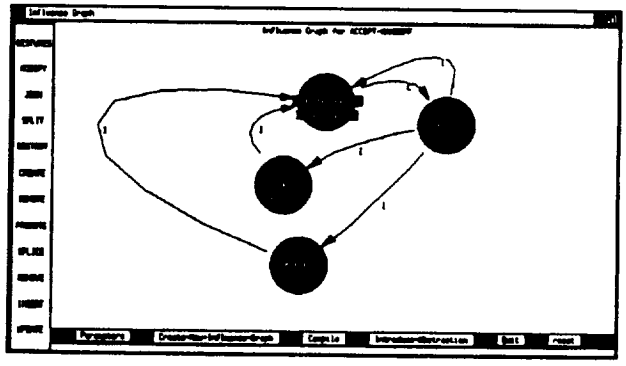

Figure 9: Influence Graph of the event *accept-handoff*

Below are some of the abstractions which are automatically applied during influence analysis.

- Remove influences on self.
- Remove influences from concepts in the Predefined folder (i.e., commonly used relations, e.g., *and*).
- Remove *static* concepts which have no influences on them[3].
- Remove variables and parameters which are not explicitly influenced.[4]

Interactive Graph Abstraction

Not all abstractions can be done in an automatic fashion. Typically, the presence of certain influences indicate either an error in the specification or an opportunity to apply an abstraction. The analyst must make these decisions. ASC identifies these cases during automatic influence abstraction and then posts notifications via an agenda mechanism. When the analyst is ready, he/she may view the agenda and the alternative actions recommended by ASC. Recommendations typically include suggested transformations which can cause the desired effect in either the evolving specialized specification or the underlying specification. Some of these interactive suggestions are:

- When there are no influences on a type or relation declaration, suggest that the concept should be declared static.
- When there is an influence on a type or relation declaration, suggest that the concept should be changed to dynamic (e.g., explicit or derived relation).
- When an influence node has only a single input information influence and only a single output information influence, suggest that the intermediate node be abstracted out and the input and output nodes be modified to be a direct influence.

For validation question *VQ-Handoff* of figure 2, influence analysis results in 224 influence nodes. After automatic abstraction this count is reduced to 97 influence nodes with 51 posted suggestions (most of which concern suggestions on declaring concepts as dynamic or static). After the analyst handles the most obvious suggestions, the influence node count is reduced to 77. Though an improvement over the starting point of 224 concepts, there are still a lot of concepts to compile for simulation.

Specification Reformulation

The previous section's analysis and reformulation were basically independent of the validation question. This section will suggest more drastic reformulations which take advantage of the knowledge implicit in the validation question.

Modification, whether motivated by errors discovered in the specification or by simplifying assumptions, are accomplished via the application of transformations. Since ASC is a component of ARIES, it is able to take advantage of an extensive library of evolution transformations [Johnson and Feather, 1990]. These transformations formally evolve a specification based on specific desired effects.

An important feature of the reformulation process is that not all the effort to build the specialized specification need be thrown away after doing validation. Many of the applied transformations are equally valid in both the specialized specification and the original specification. ASC allows the analyst to declare during reformulation if a transformation should be recorded and later applied to the original specification. This selective record of transformations provides an opportunity for these transformations to be replayed on the original specification. (This selective replay capability has not yet been implemented in ASC.)

Reformulations Motivated by the Validation Question

Reformulation based on a validation question is analogous to how partial evaluation (mixed computation) [Ershov, 1985] is able to generate an efficient residual program based on a more general program and a subset of its input parameters. This technique is potentially more powerful because a validation question is a richer source of knowledge than just a list of input parameters.

Reformulation based on assumptions We begin with *VQ-Handoff*'s assumption (see figure 5) that there will be a fixed number of tracks which already exist. Figure 10 shows the influence graph of *fixed-set-of-tracks* and all of the concepts in the specification it directly influences, i.e., *track*, *initiate-tracking*, and *extract-track-info*.

The analyst begins with the event *initiate-tracking*. Influence analysis shows there are no other influences on it. Visual inspection reveals to the analyst that it creates tracks. Since the assumption says there will be no new tracks, this event is superfluous and thus should be abstracted out.[5]

The analyst next handles the type *track*. The implication of the assumption is obvious. The type should be declared static. Note that this is not generally true within the air traffic control domain, but illustrates a common result of validation question assumptions.

The third influenced node is the event *extract-track-info*. The analyst attempts to handle it as he/she

[3] If there is an influence on a static concept post it as an error.

[4] Explicit information influence are various forms of assignment. Event parameters are almost always removed since the dominating influence is the control influence on the event (e.g., who the caller is).

[5] A theorem prover would be very useful here. Given the narrowed context, it might be tractable to prove the above conclusion automatically. This is outside the scope of ASC.

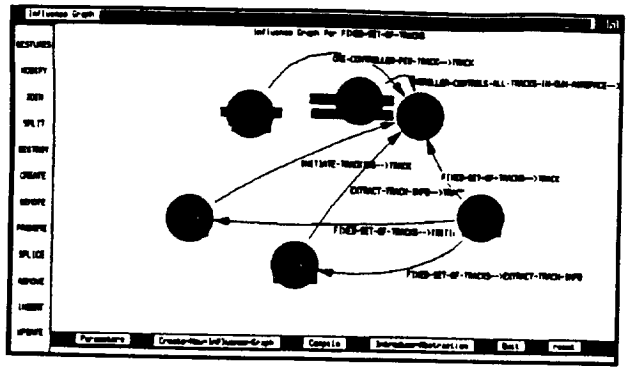

Figure 10: Influence Graph of the assumption *fixed-set-of-tracks*

handled *initiate-tracking*. In this case, visual inspection of the event shows that this event both creates *tracks* and assigns them a *track-position*. At this point the analyst needs to determine if he/she will specialize *extract-track-info* into a new event which only deals with *track-position* or if the event may be abstracted away completely. The analyst then displays a new influence graph as shown in figure 11. This influence graph shows that *extract-track-info* influences both *track* and *track-position*. Additionally, *track-position* is influenced by the driving scenario *ds-level-3*. At this point, remembering that driving scenarios have taken responsibility to be the sole maintainer of the concepts they influence, the analyst now removes *extract-track-info* from the specialized specification.

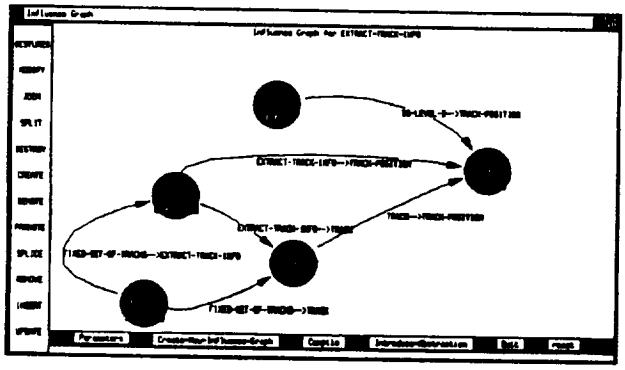

Figure 11: Influence Graph of the event *extract-track-info*

Reformulation based on driving Scenarios In the current example, influence analysis shows that the *next-controller* relation used by *automatic-init-handoff* relies on *paired* and *flight-plan* neither of which is fully defined. Additional analysis shows that *flight-plan* influences only a few other concepts including *conformance*. The analyst decides to abstract out *flight-plan* and *paired* and then model *next-controller* and *conformance* without them.

Two approaches are possible. One is to redefine *next-controller* such that it can be derived from concepts already within the specialize specification or to directly maintain the relation. In the spirit of picking the approach which is quick (and hopefully not too dirty), the latter is chosen. This is easy to do within the context of the validation question. The analyst simply includes an assertion as part of the driving scenario that *next-controller(ac-1, c1)*.

More reformulation based on assumptions *Conformance* is handled slightly differently. Since *VQ-Handoff* deals with handoffs which are initiated automatically, the analyst can take advantage of this specialized case knowledge and assume *conformance* is always true. To assume otherwise would imply one is not in an automatic handoff situation and thus would be outside the scope of *VQ-Handoff*.

Note that it might be tempting to abstract away *conformance* by assuming *inhibited-handoff* is false, but this would fail because *inhibited-handoff* also influences other events which are directly involved in the validation question (see figure 12). Such assumptions which influence system behavior with respect to the validation question are not acceptable.

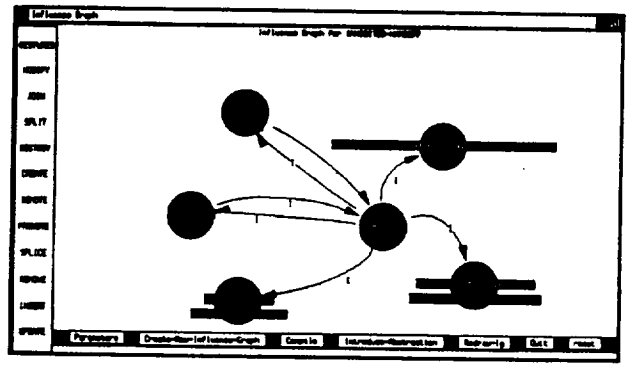

Figure 12: Influence Graph of the relation *Inhibited-Handoff*

Another use for assumptions is to decompose validation questions into smaller more manageable pieces similar to how assumptions are used in proofs to break them up into multiple, hopefully more manageable, pieces. In the case of automatic verses manual initiation of handoff, a well chosen assumption creates two distinct scenarios which are then handled separately.

Reformulation to introduce qualitative abstractions An earlier version of the validation question *VQ-Handoff* was expressed in terms of *track-position*. Likewise, many event preconditions were expressed in terms of *track-position*. Rather than describe scenar-

ios in terms of *track-positions*, the analyst decided to reformulate the specification to introduce qualitative abstractions. ASC facilitated this by showing what concepts where influenced by *track-positions*. The analyst was then able to use ARIES transformations to replace complex predicates about *track-position* with new predicates expressed in terms of newly defined relations. These new relations were *top-of-block-altitude*, *within-handoff-computed-point*, *within-accept-handoff-computed-point-distance*, and *within-accept-handoff-computed-point-time*. Each is a specializations of *track-position*. This now allowed the analyst to easily describe validation questions and scenarios in terms of these qualitative states rather than in terms of *track-position* which has many more states but which fall into one of these five qualitative states.

Reformulating scenarios as run-time constraints Not all optimizations are realizable during specialized specification construction. This problem is pointed out by Meyer in [Meyer, 1991] when applying partial evaluation to imperative languages. The problem is that compile time execution can result in side-effects which are not noticed at the appropriate time. This is because the side-effect could happen during specialized specification construction and not during simulation. The problem with this is that other parts of the specification which trigger on the side-effects of the partially evaluated scenario will now not have those side-effects to react to at run-time. As a result partial evaluation at specialized specification construction time must be constrained not do anything that causes triggering states to disappear.

In *VQ-Handoff*, the handoff scenario includes only *automatic-init-handoff* and *accept-handoff*. It excludes several other events that could be a part of a typical handoff scenario (e.g., *manual-init-handoff*, *reject-handoff*, and *cancel-handoff*). ASC translates these scenarios into a procedural invariant which ensures the appropriate behavior at run-time. The advantage of such a constraint is that one is not forced to deal with control issues regard the full set of events until the pairwise (in this case *automatic-init-handoff* and *accept-handoff*) interaction has first been resolved.

Horizon Effect Addressed

ASC mitigates the horizon effect by trying to create a specialized specification which defines a closed simulation model. In such a model there are no outside influences and all influence paths within the closed model are known, thus there is no horizon effect.

With respect to the current example, reformulation continues until a closed model is achieved. The final specialized specification contains 44 influence nodes.

Summary

The success of the ARIES Simulation Component may be measured with respect to the following criteria.

- ability to execute a specification where previously it could not be done.
- ability to execute a specification with less effort than was required before.
- ability to document requirements satisfaction.
- ability to make validation comprehensible to stakeholders.
- ability to provide a flexible approach for system validation.

At this point its too early to address most of these issues. I have only applied ASC to a few validation questions, all within the domain of ATC handoffs. The most quantifiable results to date concern the number of concepts involved in the specification of *VQ-Handoff*, figure 13.

	number of concepts
ATC Knowledge Base	1,400
Primitive Influence Graph	224
Initial Influence Graph	97
Final Specialized Specification	44

Figure 13: Number of Concept Declarations for Validation Question *vq-handoff*

This illustrates that only a fraction of the total number of possible concepts were actually needed to achieve executability. Additionally, the focus provided by the validation question provided direction on which concepts to flesh out next in order to achieve closure and executability. Granted these techniques could and have to some degree been applied by hand when one builds a rapid prototype. The difference is that ASC generates both a rapid prototype and a formal characterization of how it relates to the original specification.

As a sidebar, in the process of constructing the specialized specification I discovered several errors. Many of these were in fact errors in the specification which I believed was essentially correct. This seems good, in that error discovery is an important precursor to validation.

References

Boehm, B. 1981. *Software Engineering Economics.* Prentice Hall.

Johnson, W.L.; Feather, M.S.; and Harris, D.R. 1991. The KBSA requirements/specifications facet: ARIES. In *Proceedings of the 6th Knowledge-Based Software Engineering Conference.* to appear in *IEEE Expert.*

Brooks, F. P. 1986. No silver bullet: Essence and accidents of software engineering. *Computer* 20(4):10–19.

Johnson, W.L. and Feather, M.S. 1991. Reusable gist language description. Available from USC / ISI.

Kleer, J.de 1986. How circuits work. In *Qualitative Reasoning About Physical Systems*. MIT Press. 205–280.

Kleer, J.de and Brown, J. S. 1986. Qualitative physics based on confluences. In *Qualitative Reasoning About Physical Systems*. MIT Press. 7–83.

Johnson, W.L. and Feather, M.S. 1990. Using evolution transformations to construct specifications. In *Automating Software Design*. AAAI Press.

Ershov, A. P. 1985. On mixed computation: Informal account of the strict and polyvariant computation shemes. In *NATO ASI Series, Vol. F14 - Control Flow and Data Flow: Concepts of Distributed Programming*. Springer-Verlag. 107–120.

Meyer, U. 1991. Techniques for partial evaluation of imperative languages. In *Proceedings of Sympsium on Partial Evaluation and Semantic-Based Program Manipulation*, Yale Univ., New Haven, CT. 94–105.