N93-17512

Interactive Specification Acquisition via Scenarios: A Proposal

513-61
136887

Robert J. Hall
AT&T Bell Laboratories
600 Mountain Ave.
Murray Hill, New Jersey 07974-0636
hall@allegra.att.com

P4

Abstract

Some reactive systems are most naturally specified by giving large collections of behavior scenarios. These collections not only specify the behavior of the system, but also provide good test suites for validating the implemented system. Due to the complexity of the systems and the number of scenarios, however, it appears that automated assistance is necessary to make this software development process workable. ISAT is a proposed interactive system for supporting the acquisition and maintenance of a formal system specification from scenarios, as well as automatic synthesis of control code and automated test generation. This paper discusses the background, motivation, proposed functions, and implementation status of ISAT.

Note: This work is still in its early stages; comments, criticisms, and literature pointers are not only welcomed, but actively sought.

Background and Motivation

Some reactive systems, such as telephone switches and other control systems, seem to be most naturally specified informally by giving a set of behavior scenarios, consisting of interleaved sequences of applied stimuli and verified system responses. Here is such a scenario from the domain of telephone switches:

Assumptions: X, Y, and Z are idle stations.

Stimulus: Y activates Call Forwarding to Z.
Response: Y receives a confirmation tone.

Stimulus: Place a call from X to Y.
Response: Y receives a redirect notification.
Response: Z rings.

Stimulus: Z answers the call.
Response: X and Z are connected.

A comprehensive collection of these scenarios forms both an informal system specification and a suite of system tests. There are, however, several major problems with incorporating this specification and testing technique into a software development process:

- *Too many scenarios.* For systems as complex as modern phone switches, for example, there are too many scenarios (typically many thousands) for the entire suite to be manually executed even once per software release (of which there may be many per year). Furthermore, if bugs are found during an execution of the scenarios, there is no time to go back and revalidate each bug fix.
- *Ambiguity.* Informal English descriptions can be ambiguous. For example, the scenario above does not specify the technique that Y must use to activate Call Forwarding, but some such techniques may not result in a confirmation tone. Thus, the outcome of the test execution can be dependent on how the ambiguity is resolved.
- *Inaccuracy.* Informal English descriptions can be incorrect. For example, the default administration of stations dictates that they do not receive redirect notifications unless this feature is explicitly activated. Thus, the scenario above will fail, unless the tester inserts the missing administration step.
- *Consistency Maintenance.* It is difficult for humans to maintain the mutual consistency of these scenario sets as the system evolves over time, and the developers come and go. For example, an early scenario may specify that a call to a busy station is denied, with the caller receiving a busy signal; subsequently, the system may be changed so that a later scenario specifies that a call to a busy station is redirected to an automated answering feature. This change causes the first scenario to fail in system test.
- *Testing Pragmatics.* In running a batch of system tests sequentially, it is crucial that one test leave the stations in a known "default" state, so that subsequent tests' initial assumptions are satisfied. The scenario above violates this felicity condition by leaving Call Forwarding activated and leaving X, Y, and Z offhook. Thus, a batch test run can fail, even

though the software is really correct, simply because of the ordering of the tests.

A First Try: KITSS

In the KITSS project (Nonnenmann and Eddy, 1992),[1] the goal is to ameliorate these problems by translating from English into a formal test script language which can be automatically executed on a system test harness. KITSS operates in the domain of telephone switch software, using a knowledge-based domain reasoner that both assists in the translation and audits the scripts for consistency with a domain model. KITSS has the potential to help with all of the problems noted above: the translation process disambiguates the input, using sophisticated reference resolution and pragmatic reasoning, as well as a library of domain plans. Next, the auditing phase maintains consistency with the domain model and detects inconsistencies resulting from inaccurate scenarios. Finally, a planner uses domain knowledge to repair incomplete plans and to restore the system state at the ends of scenarios.

While it is beyond the scope of this paper to analyze all of the successes and failures of the KITSS project, I believe there are several lessons of the project with direct impact on this proposal.

The reasoner's model cannot be static. One basic assumption of KITSS is that there is a highly complete and virtually *static* domain model that can be built once and changed only very slowly. If this were true, then the effort of building this model could be amortized over all applications of the system. Unfortunately, change seems to be the rule rather than the exception. In one case study of only the knowledge required to support the natural language semantics module, I calculated that I had to add (or change) roughly one *knowledge unit*[2] for every five sentences processed successfully. This was measured in adding the knowledge required to allow the system to correctly translate all sentences of 38 scenarios (roughly 400 sentences). Moreover, the frequency of knowledge addition was not "converging" as the test coverage grew. The simple reason is that broader coverage means new things to talk about and new ways of talking about things. While this is hardly a definitive study, it is nevertheless suggestive that the domain model will constantly undergo evolution, rather than remaining fixed.

Experience with other KITSS system modules, such as the automated reasoner and the natural language parser, indicates this phenomenon applies to them as well. Each time a system release includes a new feature, the reasoner's domain model must be extensively updated to allow for it. Even if no new feature is added in a release, it is typical that some aspect of the specified behavior is either changed or simply better defined. Commonly, unanticipated *feature interactions* need to be defined or repaired; for example, it may be necessary to change what happens if a Priority Call is placed to a station with Call Forwarding active, since such calls are not treated as normal calls. Of all KITSS modules, only the natural language parser (Jones and Eisner, 1992) has addressed the issue of automating the acquisition process.

The natural language semantics problem is too hard. At the start of the KITSS project, it was believed that the natural language used in writing the scripts was constrained enough to make possible automatic understanding. While this seems to be true for the *syntactic* aspects of the English used (Jones and Eisner, 1992), it appears that the *semantic* aspects are wildly unconstrained, with sentence styles varying from simple and action-centered to elliptic, imprecise, inaccurate, subjunctive, and even metaphorical. An example is

> Station B2 wants to talk privately with Station B1, so presses the Consult button.

This is elliptic, in that the second half of the compound sentence leaves out who is pressing the button. It is metaphorical in that stations cannot really have desires, and cannot really talk. To fully handle phenomena such as metaphor and ellipsis, a system must have a great deal of pragmatic, common sense knowledge. It is well known that the problem of common sense knowledge is extremely difficult. A result of this observation is that we cannot depend on perfection in the natural language component, so an interactive interface is required that makes it possible for the user to examine each translation and repair it as necessary.

The benefits of imperfect natural language processing may not justify the knowledge and processing costs. In another informal study, I used KITSS to translate 14 test cases. I did this in two ways: first, by having the natural language component try to translate the sentence and only repairing those sentences inaccurately or not translated; and second, by simply manually typing the translation essentially directly into the logical language used by the domain reasoner. The session which used the natural language component required 47 minutes, while the session without only required 49 minutes. The key reasons for this, I believe, are that (1) the translation is usually extremely easy to find for someone familiar with the logical language, and (2) the processing time in the domain reasoner was large enough that there was plenty of time for the human reasoner to think about the paraphrase in parallel with this processing.

[1]KITSS was reported on in last year's Workshop (Nonnenmann and Eddy, 1991) as well.

[2]A "knowledge unit" is a qualitative unit of effort defined with respect to the knowledge formalisms employed in the KITSS system. Its key properties are that it must be added manually to the system by a relative expert in the domain model, and each knowledge unit is of roughly the same complexity to add (as measured by the time to discover and add it).

A New Approach: ISAT

The subject of this proposal is a new tool called ISAT, for Interactive Specification Acquisition Tool. The first point of departure with KITSS is to acknowledge the model acquisition and maintenance problem as the overriding problem. This has impact throughout the tool's design, starting with a different role for the user. Whereas in KITSS the user's task is knowledge-assisted translation of scenarios, given a static system and domain model, the ISAT user's task is to synthesize a system model (specification) which is consistent with the scenarios. Translation of the scenarios into automated test scripts is a by-product of this process, rather than the primary goal. Thus, whereas the users of KITSS are system testers, the users of ISAT are the developers and designers of the system. Of course, the testers still benefit from fully automated and maintainable test scripts.

Note that there is a subtle difference between KITSS's domain model and ISAT's system model. The domain model in KITSS has evolved into a collection of constraints, plans, and inference rules. It is not, however, a predictive model of the switch, as this would require *completeness*. Such completeness is impossible to attain in a system with a fixed domain model. Thus, KITSS is capable only of checking certain aspects of scenario consistency, and uses plan recognition to fill in missing scenario steps.

By contrast, ISAT's system model *is* assumed to be predictive. It must be complete enough to predict all observations in all scenarios. Whenever there is an unpredicted observation or an inconsistency, the user must fix either the scenario or the model. By designing ISAT for maintainability, however, this is acceptable. Note that I will use the terms "specification" and "system model" interchangeably throughout to denote a predictive model of the stimulus/response behavior of the target software.

One might wonder why it should be possible to acquire such a specification from the user, since traditionally software specifications have been extremely difficult to produce. There are two answers to this question. First, the goal is to acquire a behavioral, input/output specification of the system similar to what one might find in a user's manual for the target system. Since, for example, there are those who claim to understand how to use their phones, we might expect this level of specification to be much simpler than a full specification of the switch software itself. A full specification would include far more detail than is normally seen by a user, such as constraints imposed by hardware resources. Second, in the ISAT project I am not requiring a complete and accurate specification up front. Instead, the specification is fundamentally an evolving entity which undergoes continuous, but controlled, change. By designing for maintainability, and supporting automatic code synthesis (see below), ISAT sidesteps the difficulties of full specification.

In view of the observations above about natural language processing, ISAT will not accept English input; rather, it will accept formal input only. Thus, each scenario must be manually transcribed into a formal stimulus/response language. Furthermore, the system model itself will be expressed in a formal rule language with a clear semantics.

I believe that this problem redefinition, even though it places a larger burden on the user, allows much more leverage on the testing and maintenance problems. The next section will discuss in detail the benefits which I believe should accrue from this change. Broadly speaking, ISAT (like KITSS) is an *apprentice* system, i.e. one which assists engineers in doing a task by automating the routine subtasks and tracking as many details as possible. Examples of this paradigm in the literature abound: the LEAP system (Mitchell, et al, 1985) was an apprentice VLSI design assistant, and the MIT Programmer's Apprentice Project (Rich and Waters, 1990) supported research on many different apprentice systems, such as KBEmacs, the Design Apprentice, and the Requirements Apprentice (Reubenstein, 1990).

Proposed Tool Functions

Through an extended interactive dialog, augmented by batch-mode processing of various subtasks, the system supports the user in constructing a predictive model of target system behavior that is complete enough to predict every response in every test scenario. With such a model, several important software development tasks can be carried out. The primary functions of the proposed ISAT apprentice system are given here and discussed in more detail below.

- *Scenario checking.* Verify that each response in a given scenario is predicted by the model, given the assumptions and stimuli in the scenario.
- *Model Maintenance.* Control and analyze a user's changes to the system model, performing impact analysis and regression testing to ensure that such changes are consistent with all known scenarios. Provide diagnostics and explanations when conflicts arise.
- *Automatic Programming.* At any time when the system model is known to be consistent with the scenarios, compile the model into an efficiently executable control module for the target system.
- *Generation of Automated Test Scripts.* Put out scripts in the low-level executable form necessary for execution on a test harness. This includes filling in missing steps necessary to leaving the system in the default state, error recovery, etc. (This is essentially the KITSS task.)
- *Test Suite Enhancement.* Fill in individual scenarios with additional response verifications that were left out of the input scripts, based on the predictions of the model. Possibly suggest additional scenarios

for testing known gaps in scenario coverage, such as model rules that are never fired or state variables that never change.

Scenario Checking

The fundamental mode of interaction in acquiring the model is for the user to present a scenario to the system, which the system then "simulates" using its current system model (represented in a simple pattern-action rule form based on a simple notion of state). The system then informs the user whether the behavior specified by the scenario is successfully predicted by the system's model. Exceptional conditions are raised when any of the following conditions arise:

- a response specified by the user *contradicts* some deduced consequence of the system's model,
- a response specified by the user, though not contradictory, fails to be predicted by the system's model (indicating possible incompleteness of the model),
- a stimulus applied by the user is deduced to be illegal in the system's current state
- the system model reaches an inconsistent internal state (which may not be observable as a system response in the scenario).

Whenever such an exception arises, it could be due either to an incomplete or inaccurate system model held by the system, or to an incomplete or inaccurate scenario presented by the user. Thus, the first important automated facility of the ISAT system is the ability to *fully explain* any deduced state condition. This explanation is presented in terms of the pattern-action rules constituting the system's model and the scenario stimuli and configurational information given by the user. The user can then use this explanation to isolate the difficulty, resulting in either changing the scenario definition or fixing the model.

Note that this explanation facility hinges on a key property of reactive system control software: each event results in a relatively small number of internal state changes. This allows us to construct a fully explained and complete execution trace of the system model on a given scenario input. This property does not hold true of other types of software, such as data processing software, compilers, etc. They typically have enormous traces, involving millions of internal states. It is practically impossible to build and query a complete trace of such a system.

Another function potentially performed by the checker is to compare the final state of a scenario with the assumed default initial state of all scenarios. ISAT can then point out when the state is left in a non-default state, and even assist in planning some steps for correcting it.

Model Maintenance

If the difficulty lies in the system's model, the user must decide how to repair the model. Usually, for any given model repair, the biggest difficulty lies in understanding how the proposed change will effect the correctness of the system on *other* scenarios. That is, does this fix break anything that worked before?

In ISAT, this is not a problem because of our insistence on complete explainability. In principle, each scenario can be re-checked; any that no longer complete successfully provide full explanations of why they fail, allowing the user to quickly locate the unintended interactions. In practice, we can speed up this process by orders of magnitude for small model changes by examining the justification structures built up in originally checking them; a small model change will not effect many scenarios, so this checking can quickly determine that the original structure still applies (this is analogous to the difference between deriving a proof and checking an existing proof). In my experiments with the current prototype of ISAT, this simplified impact analysis is roughly 40 times faster (for localized model changes) than a full recheck of all scenarios would be.

The above technique applies to changes in model rules; I anticipate there will be analogous protocols for dealing with other types of model changes, such as changing the types of model functions, adding and deleting state variables, etc.

Automatic Programming

Since the system model is predictive, it can serve as a controller for the target system, assuming the hardware and low-level primitives support the stimulus and response primitives directly. Specifically, I assume that the system substrate can be coded to supply typed interrupts when sensors indicate the presence of real stimuli (such as when a button is pressed, or a phone goes offhook). I also assume that the substrate supports actuators for each of the observable signals deduced by the system model (such as a status lamp lighting or a tone being emitted). Given this substrate, which may or may not exist in current day designs, we can synthesize a controller by essentially treating the sensor interrupts as stimulus statements in a scenario. Then, when the system model computes the next observable system state, the changed observables are sent as updates to the actuators.

The only difficulty with this direct approach is the efficiency of the controller: even if there are no hard real time constraints (which there may be), large systems like phone switches must handle many interrupts per second to be usable. Fortunately, I believe it should be quite possible to compile the system's model into an extremely efficient program. The run-time system need not track rule justifications, and rule chains can be precomputed at compile time. Thus, the event handlers for the system at run-time needn't do term matching, justification construction, or consistency computation.

Note that there are several benefits to this automatic programming approach.

- *Initial Coding is fast.* This is because the control part is generated from the system model automatically. Because of the extremely simple model of computation in the system and the limited domain, I hypothesize that this automatic programming problem can be fully automated.

- *Subsequent releases are relatively painless.* In many domains, new system releases tend to involve mostly changes to the high-level control, rather than the sensor-actuator substrate. Thus, future releases can be produced by first getting them correctly reflected in the ISAT system model and then automatically regenerating the controller, leaving the system substrate the same.
- *Bug fixing turnaround is fast.* Another benefit lies in debugging the actual system. If a user calls up with a bug report, it can be simulated in ISAT's model, to see if it is replicated there. If it is not, then the bug is localized to the sensor-actuator substrate. More likely, however, is that the bug is manifested in the model, where the full explanation facilities of ISAT allow quick localization, fixing, and regression checking. This can potentially lead to extremely rapid turnaround for bug fixing. Note that quick bug localization based on querying scenario traces depends on the special properties assumed of this class of reactive control systems.

Even though I believe the automatic programming task in this domain is tractable, challenges remain. For example, compiling arbitrary rule patterns into efficient code is still challenging. For example, when a rule's condition contains the pattern `(connected X ?y)`, the naive compilation performs a linear search among all stations to see which are connected to B1. It would be desirable to compile this into a hash-table based representation of the set of connected stations to X. There is much existing research on this and related compilation problems, however.

Generation of Automated Scripts

The challenge here is to translate from the high-level stimulus and response statements appearing in the scenarios into the particular low-level language used by the test harness. This will involve two steps. First, each high-level step, such as "Activate Call Forwarding from Station X to Station Y" must have one or more *compound action methods* defined telling the system how to translate it into a sequence of primitive stimuli understood by the system. I believe this is properly part of the system model acquired from the user. The final step is to do a more-or-less standard compilation step from the event primitives in the model into the language of the test harness. The event primitives should be designed to make this relatively easy.

Note that the checking phase of ISAT is presumed to have already made sure that each scenario leaves the system in a known default state.

Test Suite Enhancement

There are two different types of enhancement that ISAT can easily perform: first, each single scenario can be "filled out" with missing observations, to increase confidence in the proper working of the switch. It can do this based on the predictions of its model. For example, it might insert checks for dial tone after each off-hook operation. Of course, we must be careful not to bog down the scenarios in endless checking of details, since there are so many to execute on the test harness.

The second type of enhancement is to the coverage of the suite as a whole. If a model rule is never fired in any scenario, this probably indicates that one or more scenarios should be created to exercise it. (Otherwise, why would the user put it in?) Similarly, if a certain type of state variable either always has the same value or is never determined in all scenarios, scenarios should be created to see if it is possible to cause a change. At the very least, these types of conditions can be brought to the user's attention. More ambitiously, the system can do goal-based planning to try to bring about the conditions necessary to firing the rule or changing the variable's value.

Implementation Strategies

To date, I have implemented an initial prototype demonstrating some of ISAT's capabilities. In particular, the system can check scenarios and perform impact analysis for individual rule changes. It can also emit low-level streams of stimulus/response statements as the first step in producing automated scripts. I have used ISAT to build several different models of different combinations of telephone switch features. The most complex is a model that layers Priority Calling and Call Forwarding on top of Plain Old Telephone Service for multiple call-appearance phone stations. Since this model acquisition was done in parallel with ISAT implementation, it is premature to attempt to gauge how well I was supported in this by ISAT. I have used the model to successfully check 17 scenarios taken from actual pre-existing AT&T system test documents.

The formalism I've adopted is based on simple pattern-action rules used to form partial descriptions of state transitions.

```
   if (CALL-STATE (CA ?X ?N)) = :IDLE, and
      (SELECTED-CA ?X)        = (CA ?X ?N), and
      (HOOK! ?X :OFF)         = :TRUE,
 then (CALL-STATE (CA ?X ?N)) = :PRE-DIALING
```

This rule says that if call appearance number ?N at station ?X is idle and is the selected appearance of ?X and ?X goes offhook, then in the next system state the CALL-STATE of that appearance is ":PRE-DIALING". There are two types of model rules: state change rules, like the above, are used in a "forward" manner; that is, they are executed to quiescence after each stimulus event is applied. Demand rules, like

```
  if (CALL-STATE (CA ?X ?N)) = :DENIED, and
     (SELECTED-CA ?X)        = (CA ?X ?N), and
     (ONHOOK? ?X)            = :FALSE
then (RECEIVED-TONE ?X)      = :BUSY-TONE
```

are only used when a response event asks about one or more of the predicates in the rule's conclusion. Thus, the above rule will only be used when the scenario executes an observation of the received-tone of some station. Demand rules are necessary so that the system need not forward chain to deduce a large number of observations that won't be used in a given state. For example, there are quadratically many potential connections between stations, but by making the observable connected predicate only deduced on demand, the model can execute in linear time.

Note that, unlike state rules, the demand rules do not chain arbitrarily. They are used only one level deep. This makes the system's efficiency much more predictable, and I have found it no significant restriction in expressive power.

ISAT deals with the classical AI frame problem (how to consistently carry forward unchanged facts when a small aspect of the state changes) by distinguishing different declared ontological statuses of terms. Any term consisting of an application of a function declared :PERSISTENT by the user has its old value brought forward, unless an explicit contradiction is deduced by a rule firing. Any non-persistent terms must be rededuced in each state. Most such terms are deduced by demand rules, though, so they do not incur a large unnecessary cost.

Because of the extremely simple formalism and semantics, the current prototype is able to support several useful explanation functions. Of course, it can answer (WHY? <fact> <state>) by simply giving the explicitly maintained justification in terms of rule applications, external inputs, etc. Another extremely useful capability is the ability to answer (WHY-NOT? <fact> <state>). Obviously, in its most general form, this is too open-ended to be meaningful; but in the context of ISAT this is interpreted to mean the following. First, tell me any contradictory facts (optionally explaining them); then, tell me all the rules that could have deduced the fact and tell me why they didn't fire (by telling me which of their conditions are not satisfied). This has been very useful in building the models I have already built. An analogous facility is (METHODS? <action statement> <state>), which gives a description of which compound action methods apply in the state for the given compound action application, and which fail to apply and why.

Summary

The proposed tool, ISAT, is a software development tool environment for reactive control systems, such as telephone switch software. It is motivated by, and builds on lessons learned from, the KITSS project. By redefining the problem and meeting the model acquisition problem head-on, I believe many major benefits are achievable, such as specification acquisition and maintenance, automatic synthesis of control systems, test enhancement, and automated script compilation. An initial prototype of ISAT is currently under construction, with several of the main functions implemented. It has been tested on several scenarios from a "real world" application.

Acknowledgements

This proposal has grown out of work on the KITSS project at AT&T Bell Labs. I have learned a great deal from this project and the people involved with it: Van Kelly, Mark Jones, John Eddy, and Uwe Nonnenmann. Some of the ideas and opinions expressed in this paper are clearly derived from the conceptual foundation of KITSS. However, I make no claim as to whether the individual project members (except me) agree with the specific opinions expressed here.

References

Jones, M.A.; and Eisner, J.M. 1992. A probabilistic parser applied to software testing documents. In Proceedings of the Tenth National Conference on Artificial Intelligence. Cambridge, MA: MIT Press.

Mitchell, T.; Mahadevan, S.; and Steinberg, L. 1985. LEAP: A learning apprentice for VLSI Design, In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, vol 1, pp 573–580. Los Altos, CA: Morgan Kaufmann.

Nonnenmann, U.; and Eddy, J.K. 1992. KITSS – A functional software testing system using a hybrid domain model. In Proceedings of the Eighth IEEE Conference on Artificial Intelligence Applications. Montery, CA: IEEE.

Nonnenmann, U.; and Eddy, J.K. 1991. KITSS – Toward software design and testing integration. In Proceedings of the AAAI-91 workshop: Automating Software Design: Interactive Design. AAAI.

Reubenstein, H. 1990. *Automated Acquisition of Evolving Informal Descriptions*, Technical Report, AI-TR-1205. M.I.T. Artificial Intelligence Laboratory.

Rich, C.; and Waters, R. 1990. *The Programmer's Apprentice*, New York, NY: ACM Press.