Explanation-based Scenario Generation for Reactive System Models

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
hall@research.att.com

IEEE made me say that. The actual paper starts on the next page.

Explanation-based Scenario Generation for Reactive System Models

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
hall@research.att.com

Abstract

Reactive systems control many useful and complex real-world devices. Tool-supported specification modeling helps software engineers design such systems correctly. One such tool is a scenario generator, which constructs an input event sequence for the spec model that reaches a state satisfying given criteria. It can uncover counterexamples to desired safety properties, explain feature interactions in concrete terms to requirements analysts, and even provide online help to end users learning how to use a system. However, while exhaustive search algorithms work in limited domains, the problem is highly intractable for the functionally rich models that correspond naturally to complex systems engineers wish to design. This paper describes a novel heuristic approach to the problem that is applicable to a large class of infinite state reactive systems. The key idea is to piece together scenarios that achieve subgoals into a single scenario achieving the conjunction of the subgoals. The scenarios are mined from a library captured independently during requirements acquisition. Explanation-based generalization then abstracts them so they may be coinstantiated and interleaved. The approach is implemented, and I present the results of applying the tool to tasks arising from a case study of telephony feature interactions.

1 Introduction

Reactive systems control many useful and complex real-world devices, such as telephone switches, air and space craft, and software agents. Such feature-rich systems are difficult to design correctly, particularly when distinct functional features are designed by different people at different times over the lifecycle of a product family. *Specification modeling[11, 16]* allows engineers to apply relatively sophisticated validation tools such as simulation, coverage analysis, model checking[17, 5], or theorem proving[12, 20], to relatively abstract models of the system's behavior in order to find design errors before implementation. It is the abstractness of the models that makes many of the reasoning techniques tractable. The validated spec model can be used as a starting point for code generation, as documentation of the behavior of the system, and in support of maintenance and evolution[11].

A spec modeling tool suite benefits significantly from a *scenario generator*, which constructs an input event sequence for the spec model that reaches a state satisfying given criteria. Such a tool can uncover counterexamples to desired safety properties, explain feature interactions in concrete terms to requirements analysts, increase test coverage, and even function as documentation, showing end users how to achieve their goals while still learning how to use a system. However, while some model checkers[17, 4] are capable of generating scenarios for certain limited classes of reactive systems, such as finite state machines with small (or highly symmetric) state spaces, the problem is intractable for functionally rich models that arise as natural abstractions of systems engineers wish to design. For example, in addition to requiring search in an infinite state space, models incorporating arithmetic operators can require the scenario generator to find satisfying instances of arbitrary arithmetic constraints, which is undecidable.

This paper describes a novel heuristic approach, called SGEN^2 (for "scenario generation via generalization") which is applicable to a large class of infinite state reactive systems. The key idea is to instantiate and piece together abstracted scenarios that achieve subsets of the conjuncts of a goal predicate into a single scenario achieving the conjunction of the subsets. The scenarios are mined from a library of concrete scenarios captured independently during requirements acquisition. Critically, they are then abstracted via explanation-based generalization. The approach is

sound, but incomplete, so it will not succeed in finding scenarios in all cases of satisfiable goal predicates; however, it is intended to be fast, even in failing cases, so that it can be a practical interactive tool. Moreover, the approach's power can be increased by adding more scenarios to the library, so, as more requirements are uncovered and specified, the power of the tool grows naturally. Even an incomplete generator is quite useful. Typically, an engineer will discover (e.g. via static analysis or proof attempts) descriptions of states in which spec inconsistencies may arise, or correctness properties may be violated; the scenario generator is then run on these descriptions. Whenever the generator is successful, a definite design flaw has been found, so the engineer can focus attention there first. The other cases, which may not even be satisfiable, can be put off to later in the design process, after the known problems are fixed. Fixing these first problems may either alter or eliminate the other ones anyway. When the generator fails, putting out a scenario coming as close to the goal as possible can be helpful as well.

This paper can be summed up in three key ideas:

- Current limited-domain exhaustive search approaches (such as model checking[17]) to scenario generation are not enough; we need a usable scenario generator that accommodates more expressive logics and large state spaces, even though the problem is highly intractable;
- The heuristic SGEN^2 approach, based on mining and abstracting requirements knowledge using explanation-based generalization, applies to richly expressive logics and large state spaces;
- A moderate sized case study involving feature interactions in telephony gives initial empirical evidence that SGEN^2 is practical and useful.

After Section 2 defines terms and describes the tool suite in which SGEN^2 is implemented, the next three sections make these key points. I conclude with discussion of related work, limitations, and future work.

2 Background: Spec Modeling

This work is performed within the Interactive Specification Acquisition Tools (ISAT) framework. ISAT[11, 12, 13] is a prototype tool suite for reactive system design that is intended to support full-lifecycle spec modeling as well as code generation. A *reactive system* is a (not necessarily finite-) state machine that reacts to parameterized input *events* by changing its state and by performing *acts*, which can be thought as output events. ISAT is based on two hypotheses:

- Functional requirements are most reliably elicited from and validated by requirers as concrete, formal behavior scenarios; and
- Specifications must be executable and amenable to automatic analysis.

A designer constructs a reactive system *model* in the executable spec language, while a requirer specifies functional requirements as concrete scenarios. The latter are interleaved sequences of input events and act or state observations required to be true. Thus, crucially to SGEN^2, *a natural part of the design lifecycle is the acquisition of a library of validated concrete scenarios* describing the system's behavior.

2.1 Model Formalism and Backpropagation

An ISAT spec model consists of a *theory definition* together with a set of *event handlers*. The theory defines the types, functions, and semantic axioms of a pure (side-effect-free) computational logic, as well as the signatures of the state relations, events, and acts that make up the system. In order to support model simulation (execution), all primitive function declarations in the model's theory must include a total operational function capable of computing the value of the function on inputs in its domain and some non-error, type-compatible output value on inputs outside its declared domain. (ISAT model theories are somewhat similar to computational logic as described in [3].) Thus, ISAT supports arbitrary functional richness, bounded only by the user's willingness and ability to code implementations for the functions and provide the logical axioms supporting the other reasoning tools (see below). For example, models can operate on arbitrary data structures. I have used this richness to good advantage in my work on applying ISAT to the specification and implementation of the Email Channels system[13]: the ISAT model operates on message data structures, lists of users and messages, and even database relation objects.

Event handlers are expressed in a limited procedural language P-EBF ("procedural event-based formalism"), which is semantically related to the rule-based EBF I described in [11]. The details of P-EBF are not crucial to this paper, except that it supports a *predicate backpropagation* operator BACKPROP. Note that P-EBF need not be the input language seen by the designer; many domain-appropriate front-end formalisms (e.g. domain-specific languages or graphical

programming environments) may be compiled into P-EBF. Such formalisms are beyond the scope of the present paper, however.

Formally, the state of an ISAT model is represented as a collection of parameterized (partial) functional relations $r_j : D_1 \times \ldots \times D_n \mapsto T$, where T and each D_i are data domains (types). For example, the relation `CALL` : *Address* $\mapsto$ *Call* stores for an address (i.e., phone number) the object representing its ongoing call (if any). State values are referred to within P-EBF expressions via the `LOOKUP` operator; for example, `(LOOKUP CALL "1234")` returns the current call in which extension `"1234"` is involved, if any. A *state predicate* is a Boolean-typed ISAT expression. Predicates may be parameterized by typed formal parameters. Here is a state predicate of one address parameter, `usr`:

```
(and (member? usr (lookup known-addresses))
     (equal IDLE (lookup mode usr))
     (not (equal NO-CALL (lookup call usr))))
```

This predicate represents all states in which there is an idle address that nevertheless still has a valid call object. It is the negation of a desirable state invariant; thus, a generated scenario reaching such a state proves the existence of a design error.

The BackProp Operator. Formally, ISAT's BackProp takes six arguments, (P', a', s, e, s', M), and returns three values (P, E, a). P' is a state predicate and a' is a list of actual (concrete) parameters for P' such that P' is true when evaluated in model M's state s'; s is a state for model M such that applying the concrete input event e to M in s results in the new state s'. Pictorially, $M : s \stackrel{e}{\mapsto} s'$ and $P'(a')$ is true in s'. The return value E is an event schema for (variablization of) the concrete event e, defining fresh formal parameters. P is a state predicate taking the same arguments as P' plus the formals of E, and a is a list of actuals for P such that $P(a)$ is true in s. Moreover, we specify that

> BackProp$(P', a', s, e, s', M) = (P, E, a)$ if and only if for *all* states S and actual parameters $A = (A_{P'}, A_E)$ satisfying $P(S, A)$, applying $E(A_E)$ to the model M in S results in a state S' in which $P'(A_{P'}, S')$ holds.

To clarify, the formals of P are just the union of the formals of P' and those of the event schema E. Thus, the actual list A will have values both for the formals of E and the formals of P'. Intuitively, BackProp computes a sufficient (not necessarily necessary) condition on event E and the state prior to applying E, such that P' is true afterward.

BackProp applies explanation-based generalization [11, 8] to the P-EBF formalism. Others have described similar operators, such as Dijkstra's predicate transformers, or Igerashi et al's verification condition generators[18]. It is beyond the scope of this paper to explain the algorithm in detail, but here is an example. In state 1, user `"1234"` has `MODE IDLE`. The event `(OFFHOOK "1234")` results in state 2 in which the `MODE` of `"1234"` is `DIALING`. Then BackProp applied to the 1-parameter predicate `(EQUAL DIALING (LOOKUP MODE ?x))` returns the event schema `(OFFHOOK ?y)` and predicate `(AND (EQUAL :IDLE (LOOKUP MODE ?x)) (EQUAL ?x ?y))`. (The actuals lists bind both `?x` and `?y` to `"1234"`.) Intuitively, this means that if we offhook any idle user, that user will move to the dialing mode.

BackProp*. Note that if we have a succeeding scenario trace involving a sequence of input events, we can iteratively apply BackProp to get an entire *generalized scenario*, where the initial predicate will not depend on the state at all (because ISAT scenarios are defined never to succeed if they depend on uninitialized state values). The rest of the paper will refer to this operation as BackProp*; it takes in a model, a scenario trace, and a predicate to be backpropagated together with its satisfying actuals list, and returns this fully backpropagated predicate, its actuals list, and the list of event schemas making up the generalized scenario.

2.2 ISAT Tools Overview

ISAT exploits the two hypotheses above to provide a suite of analysis tools to help the designer produce a specification that meets the true needs of the requirer. ISAT includes the following tools:

- *Scenario simulation* takes a scenario and a model and executes the model to determine whether the scenario represents correct behavior of the model. Thus, requirements scenarios can be directly validated.
- *Coverage analysis* reports states never reached by, and statements of the model that are not executed by, any of the requirement scenarios. This helps the designer elicit adequate requirements from the requirer.
- *Layered theorem proving*[12, 20] is a technique for proving arbitrary correctness properties, such as state invariants and pseudo-state diagrams[12].
- *Conflict detection*[14] returns predicates describing states in which the model, if it reaches them,

will derive an inconsistent next state (potentially causing either a crash of the simulator or, worse, the implemented system). Inconsistencies can result from setting state relations to two inconsistent values or raising conflicting output events, such as playing both the ringback tone and the busy tone at the same time to the same phone.

Coverage analysis, conflict detection, and proof attempts produce state predicates to which we can apply a scenario generator in order to discover whether they represent reachable states of the model.

3 The Scenario Generation Problem

Formally, the *scenario generation problem* is to take a model M and state predicate P' and find a sequence L of concrete input events and a list of actual parameters $A_{P'}$ for P', such that executing L in M starting from the undefined initial state results in a state s' satisfying $P'(A_{P'}, s')$. For this work, I have concentrated on *conjunctive* state predicates, i.e. those whose expression consists of the logical AND of a collection of predicates. The method can be applied to disjunctions of conjunctive state predicates by applying it concurrently to each of the disjuncts, but that requires engineering for efficiency that is beyond the scope of this paper. Sections 1 and 2 discussed some ways a tool suite can benefit from solving this problem.

Why Rich Formalisms? Model checkers[17] and symbolic model checkers[5] guarantee that when they find a property not valid in a model, they return a concrete counterexample (scenario) illustrating the violation. Thus, we should explore under what circumstances these tools solve the scenario generation problem before inventing different ways to solve it.

Model checkers exhaustively search the state space of the system, testing the property in each state. Thus, they are limited by the size of the state space they can handle. Some model checkers exploit limited forms of state space symmetry to handle systems with larger spaces, but all eventually run into this "state explosion problem". And while symbolic model checkers have checked properties in impressively large (10^{120} [5], 10^{56} [2]) state spaces, it is not clear if the technique can be extended to handle nonboolean logics. For a survey of model checking and its relation to theorem proving for verification, see[6].

Should we simply avoid models with large state spaces? I believe the answer is "no." Several common types of design problems are only manifest in more complex (large or unbounded state space) models of a system. For example, complex systems are frequently designed in a modular fashion by designing functional "features" independently and then combining feature sets to meet customization or market needs. Telephone switching systems are a good example of this approach, yet many other systems are built this way. The problem is that even though individual features are valid in isolation, their combination may lead to undesirable interactions that lead to faulty behavior. The only way for a tool to discover these interactions is to model the feature combinations; it follows that the more features a system has, the more complex must its model be in order to detect interactions.

Another reason limited-space approaches are not the final answer is that it is difficult both to do enough abstraction to make the problem tractable and yet to retain enough detail to manifest the problems of interest. In particular, each property to be checked may require a different, hand-constructed model abstraction. And since designers don't know in advance which problems the system has, there could be a lot of wasted effort and/or false confidence in results. By dealing with more complex models, the abstraction can be relatively straight-forward, and a single one can be used for all properties.

Finally, another reason to prefer a single, easily produced abstraction that is clearly faithful to the system, is that there is the possibility of generating implementations directly from the models, either through code synthesis or by direct manual implementation. Often, abstractions that are necessary for tractability are missing too much detail to allow any direct mapping to implementations. For example, Alur et al[1] report on a model checking effort for a phone switch in which it was necessary to model queue data structures by 7 bit integers (representing the number of items in the queue). An implementation must supply all details of queue implementation, as well as any system behavior depending on the actual contents of the queues.

Why is scenario generation hard? As soon as our representation language allows event and state parameterization and functions, we have added an uncomputable constraint satisfaction problem to the problem of combinatorial search in large state spaces. For example, designers commonly need models with arithmetic, lists and other data structures, text manipulation functions such as pattern matching, etc. But then it is possible to define systems and properties that are only satisfied when the system reaches a state satisfying an arbitrary sentence of this rich theory. Proving such a state reachable is undecidable, by Gödel's Incompleteness Theorem; generating a sce-

nario that actually reaches it is even harder because of the combinatorial search.

Thus, in summary, we want to be able to apply scenario generation to complex modeling formalisms, and yet the problem goes from merely search to uncomputable. Our only hope in these cases is to find an approach that can solve the problem in usefully many cases, and not take too long doing it. We also require that whenever the tool returns a scenario, it actually satisfies the goal predicate (soundness). These are the goals of the SGEN^2 approach.

4 The SGEN^2 Approach

Let us term the overall conjunctive state predicate the "goal predicate" and the individual conjuncts making it up the "conjunct predicates" or simply the "conjuncts." There are two key insights behind the SGEN^2 algorithm. First, the library of requirement scenarios, while unlikely to have a scenario which reaches a state satisfying the goal predicate, nevertheless is likely to have scenarios that reach states satisfying *sets* of the conjuncts. Thus, we might find such scenarios and somehow paste them together into a single scenario that achieves the full conjunction. Now, typically two such scenarios will operate on different data items; for example, scenario 1 may achieve set 1 of the conjuncts for address `"1234"`, while scenario 2 achieves set 2 for address `"5678"`. Thus, these two concrete scenarios cannot be interleaved to form a scenario that achieves the union of the sets for a single address. However, the second key insight is that we can solve this subproblem by abstracting the two scenarios, using BACKPROP*, and finding a common instantiation of them (binding of their variables to data values) such that the union of the two predicate subsets *is* satisfied. Once such a common instantiation is found, a heuristic search merges the two event sequences into one, achieving the union of the conjunct sets. Appendix A gives a precise high-level pseudocode description of the SGEN^2 algorithm.

The following illustrative example is taken from the case study. Consider the goal predicate

```
(and (member ?y (lookup known-addresses))
     (lookup fpr-active ?y)
     (equal dialing (lookup mode ?x))
     (lookup tcs-active ?y)
     (member ?x (lookup tcs-screened-list ?y)))
```

This describes states in which known address `?y` has two features, "fpr" and "tcs" both active, with `?x` on its tcs-screened-list, and in which `?x` is `dialing`.

Initialization. SGEN^2 first mines its library and discovers scenario S_1:

```
(init)
(init-address "1234")
(activate-tcs "1234" "1234")
(offhook "1234")
```

which results in a state satisfying 4 of the 5 conjuncts:

```
(and (member ?y (lookup known-addresses))
     (equal dialing (lookup mode ?x))
     (lookup tcs-active ?y)
     (member ?x (lookup tcs-screened-list ?y)))
```

when we bind both `?x` and `?y` to `"1234"`. Since it is unlikely we will find another scenario that fortuitously achieves the rest of the goal for the constant `"1234"`, we apply BACKPROP* to the above predicate and the trace of scenario S_1 to get the generalized scenario G_1:

```
(init)
(init-address ?x)
(activate-tcs ?y ?x)
(offhook ?x)
```

subject to the backpropagated condition `(equal ?x ?y)`. SGEN^2 also records the actual bindings {`?x = "1234", ?y = "1234"`}

SGEN^2 **recursive step.** SGEN^2-REC continues by searching the mined library information for satisfiers of the remaining conjunct(s) of the goal. In this case, it discovers (among others) that the scenario S_2

```
(activate-fpr "5678" 0 10 "1357")
```

achieves the remaining conjunct `(lookup fpr-active ?y)` when `?y` is bound to `"5678"`. Note that since S_1 and S_2 operate on different constants, they cannot be directly interleaved to get a scenario reaching the desired conjunction. Applying BACKPROP* to the remaining conjunct and the trace of S_2, we get the generalized scenario G_2:

```
(activate-fpr ?y ?t1 ?t2 ?w)
```

subject to no constraints (other than implicit type constraints), with actual bindings: {`?y = "5678", ?t1 = 0, ?t2 = 10, ?w = "1357"`}.

SGEN^2-REC then calls the COINSTANTIATE routine which attempts to find a common instantiation of G_1 and G_2 obeying both sets of constraints. In this case, since the constraint set for G_2 is empty, COINSTANTIATE quickly finds that the common instantiation $I =$ {`?x = "1234", ?y = "1234", ?t1 = 0, ?t2 = 10, ?w = "1357"`} satisfies both sets.

SGEN^2-REC finally calls MERGESCENARIOS on the two scenarios $G_1(I)$ and $G_2(I)$, which denote the instances of G_1 and G_2 obtained by applying

I. MERGESCENARIOS also takes the two predicates $P_1(I)$ and $P_2(I)$ which are satisfied by $G_1(I)$ and $G_2(I)$ respectively, so that it can check whether its result satisfies both simultaneously. In the case above, MERGESCENARIOS finds the following interleaving which does, indeed, satisfy the conjunct sets.

```
(init)
(init-address "1234")
(activate-tcs "1234" "1234")
(activate-fpr "1234" 1 10 "1357")
(offhook "1234")
```

If at this point there were still unsatisfied conjuncts of the goal, SGEN^2-REC would call BACKPROP* to generalize this result scenario and then recur to search for yet another scenario to satisfy the next subset. If COINSTANTIATE or MERGESCENARIOS fails, then SGEN^2 and SGEN^2-REC move on to the next candidates in the search (cf Appendix A).

4.1 Library Mining

The first step of SGEN^2 is to search the library of execution traces of requirement scenarios for states in which sets of conjuncts are satisfied. The subroutine MINELIBRARY accomplishes this as follows. For each scenario in the requirements library, it first generates an execution trace by calling the simulator. It then extracts from the trace sets of data values (grouped by type) appearing in the trace. Then, for each possible well-typed assignment of these data values to the formal parameters of the goal predicate, it searches the states of the execution trace for those in which a conjunct first becomes true (for that parameter assignment). It creates a *predicate group satisfier (pgs)* for that state, which records the assignment and which set of conjuncts are satisfied. This set of satisfied conjuncts is termed the *satset* of the pgs. MINELIBRARY returns the entire collection of PGSs found in this way in all traces. It sorts the list in decreasing order of the size of the satset so that SGEN^2 will consider earlier those PGSs that satisfy the most predicates at once.

MINELIBRARY is linear in the total number of states in all traces in the library. More importantly, however, it is proportional to the number of parameter assignments, which is exponential in the number of goal predicate parameters. While the current implementation seems to work adequately fast on the case study examples (≤ 5 parameters each), it may be necessary to limit the number of assignments considered when the goal predicate has many parameters.

4.2 Coinstantiation

COINSTANTIATE heuristically attacks the (in general) uncomputable problem of coinstantiation by simply trying out all possible well-typed assignments of constants to the parameters of G_1 and G_2, where the constant pool is simply the union of all constants in the actual-bindings of the PGSs from which G_1 and G_2 were generalized. This has proven effective in the case study, and takes negligible time (see statistics below). If necessary, COINSTANTIATE can be made to consider larger constant pools, such as those in *all* scenarios.

4.3 Scenario Merging

MERGESCENARIOS takes in two scenario/predicate pairs, where each scenario results in a state satisfying its predicate. The goal is to return an interleaving of the two scenarios that satisfies both predicates. MERGESCENARIOS does not attempt to check all possible interleavings, as this would require checking exponentially many (in the sum of the lengths of the two input scenarios) interleavings in the worst case. And note that the worst case occurs any time no interleaving exists, so it is fairly common. Designate the input scenario/predicate pairs as the "left" scenario and predicate and the "right" scenario and predicate. Our approach is to sequentially select the front event off of either the left or right scenario and add it to the end of the result scenario until both left and right are empty. Doing this in all possible ways, waiting until left and right are empty before checking the predicates, would result in the exponential worst case mentioned above.

Instead, MERGESCENARIOS heuristically limits the search as follows. Each time it selects an event e_l from the left scenario, it checks to see whether, if the result scenario were extended from that point with the remainder of the right scenario, the right predicate would be satisfied. If not, e_l is vetoed; otherwise, it proceeds to the next choice. (By induction, one can show that if instead we extended the result with the remainder of the *left* scenario, the left predicate would also be satisfied.) The dual check is done when the event is selected from the right. When the front events on left and right are *identical*, the algorithm also attempts the third option of adding one event and discarding the other.

Note that since there can be interleavings that satisfy both predicates at the end but which contain intermediate points at which the check would fail, this approach is less powerful than brute force search; however, in the case study, MERGESCENARIOS only failed

Total Attempts	Scenario Generated	Satisfiable/ No scenario	Not Satisfiable
63	24	36	3

Table 1: SGEN^2 success on case study

once when a brute force search would have succeeded, and yet was as much as 12 times faster (average: 2x).

5 Case Study

I ran SGEN^2 on 63 distinct scenario generation problems that arose in a larger case study of feature interactions in a telephone switch specification. (The study actually produced 66 problems, but three were duplicates, so were discarded for this paper.) The larger study is actually a tool contest associated with the 1998 Feature Interactions Workshop[7]. The system being modeled is a telephone switch implementing Plain Old Telephone Service (POTS), plus 12 functional features such as Call Forwarding (CF), Terminating Call Screening (TCS), FreePhone Routing (FPR), and nine others. This SGEN^2 case study was performed before four of the twelve were modeled, so only POTS and eight features are included here. In a related paper[14], I explain how I used the ISAT tool set to model these specs and to detect various types of feature interactions among them, many of which are predicates describing states in which undesired things may happen, such as feature inconsistencies becoming manifest (conflicts) or feature correctness properties being violated. In the absence of a scenario generator, it is left to the user to determine whether those state predicates describe reachable states of the model. Thus, these 63 problems provide a moderately complex test of the power and usefulness of a scenario generator, and are representative of the problems that may be encountered by such a tool. The full data is available at [15].

Results. The 63 predicates averaged 1.72 parameters and 5.98 conjuncts each. Table 1 shows the results of running the generator. "Scenario Generated" refers to trials in which SGEN^2 succeeded in finding a scenario; "Satisfiable/No Scenario" refers to the cases when it failed to find a scenario, even though the predicate is satisfiable; and "Not satisfiable" refers to those cases determined (through external means) to be unsatisfiable and, hence, there exists no scenario to generate.

	All	Scen.Gen Only	No Scen.Gen Only
#	63	24	39
Total	8938	603	8335
Library Mining	658	510	148
BackProp	1766	65	1701
Coinstantiation	0	0	0
Merge	6216	0	6216

Table 2: SGEN^2 aggregate run times (rounded to nearest second).

Table 2 shows run time statistics for the 63 trials. All times are measured on a 225 MHz Macintosh clone (144 MB memory) running the ISAT system under Macintosh Common Lisp 4.2. For this table, the "no.scen.gen only" condition includes *all* cases where the tool did not find a scenario, whether or not the goal predicate was satisfiable (since to the user these are equivalent when waiting for the tool to finish).

Discussion. Of the 60 cases in which it was possible to generate a scenario, SGEN^2 succeeded 40% of the time. Thus, the user can be sure that at least those cases illustrate real design errors and therefore concentrate first on fixing them. Note that one error can cause scenarios to fail (due to conflicts) that would otherwise succeed far enough to reach a second error state. Thus, fixing an error can cause SGEN^2 to succeed when it failed previously. I know of one definite case (and some others suspected) where this sort of error interference occurred in the case study.

When I first ran the study, a few cases failed because individual conjuncts were not covered by the scenario library. Of course, if there is no known way to satisfy a single conjunct, the goal predicate won't be satisfied either. Fortunately, it is relatively easy to discover a scenario covering a single conjunct, such as `(member ?x (lookup tcs-screened-list ?x))`. I easily created three scenarios to cover these cases, resulting in one more success and several failures. The results above reflect these additional scenarios.

Turning to time, we see that the average time per trial is 142 seconds overall, with succeeding cases taking 25 seconds on average (101 sec maximum) and failing cases requiring 214 seconds (1054 sec maximum). Note that the distribution of time is radically different between succeeding and failing cases, with MERGESCENARIOS dominating for failing cases and MINELIBRARY dominating for succeeding cases. COINSTANTIATE was never significant, suggesting that

there is room to improve its power (by checking larger constant pools, for example) without significantly harming the overall run time. On the other hand, we must be extremely careful in increasing the power of MERGESCENARIOS since that is the bottleneck in failing cases. These results are only intended to be suggestive of future algorithmic improvements; I believe they can be significantly reduced by a careful re-engineering effort. (The current ISAT system is an exploratory prototype.) Note also that these results depend on the model and scenario library as well.

In summary, it seems that at least for validation purposes an imperfect scenario generator can still be quite useful as long as it doesn't take too long. Of course, we can always hope for a better success average, and future work will go into improving the heuristics. However, it is desirable to keep the times relatively low in all cases, including failure cases, so the tool is still usable. Thus, we must engineer the power/speed tradeoff carefully.

6 Related Work

Having discussed model checkers above, I will only summarize here. Model checkers are useful solutions to the problem of scenario generation as long as one can effectively generate models in the limited formalism necessary to run the tool tractably. However, there is reason to believe that we need to handle the more complex formalisms addressed in this work for at least the reasons discussed in Section 3. In addition, we may wish to use scenario generation in ways beyond validation, such as online help systems. For comparison, it is amusing to estimate the state space size necessary to model the telephony case study specs in a finite state formalism. If we model all 12 features for n users, I estimate there are at least

$$S(n) > 2^{n^2+4n\log n+12n}$$

reachable states (logs are to base 2). If we consider call-waiting and similar features, we need at least 3 users, but if we add forwarding and other multi-user features, one can easily imagine properties referring to 6 or more users, leading to $S(6) > 2^{170} \approx 10^{51}$ states, which would challenge even the best model checkers.

Note that even infinite-state model checkers, such as that of Bultan et al[4], are highly restrictive. That system is restricted to state spaces that are the cross product of a boolean state space and one representing integer inequalities (higher dimensional polyhedra). While increasing model checking power by adding specialized constraint reasoners shows promise, it is not even clear that most reactive systems people design will be expressible within such restricted formalisms, due to the common occurrence of functions mixing arguments of several different types.

Another class of approaches to the problem that may seem applicable are AI-based planners, such as STRIPS[9] or Prodigy[19]. The problem with applying these systems, where the spec model provides the planning operators, is that planning operators must explicitly list their consequences; for example, STRIPS operators must have ADD and DELETE lists. Similarly, the macro operators learned by the EBG-based PRODIGY system must explicitly include the goal(s) they achieve. This is too limiting, because users of scenario generation may give any goal statement they wish in terms of functions defined in the logic. Any planning operator derived from the spec model potentially achieves too many (infinitely many, in fact) different goals; far too many to be stored explicitly even if we could bound the vocabulary. SGEN^2 avoids this problem by doing its abstraction and reasoning on the fly in MINELIBRARY and BACKPROP*. The only knowledge stored is the "raw" scenario traces, unadorned with any goal information.

There is also work in the traditional testing literature on generating test inputs to cover a given path in a program. For example, Gotlieb et al[10] describe a constraint-based approach, which essentially reduces to trying to find a satisfying assignment for a boolean functional expression which is, of course, uncomputable once we enrich the formalism to include (e.g.) arithmetic. However, the constraint based approach may prove useful in improving COINSTANTIATE and MINELIBRARY; it does not address the state space search needed to handle reactive systems.

Finally, there are other spec modeling tool suites providing some of the same (and many contrasting) tools as ISAT, such as the SCR tool suite[16]. Such environments may incorporate model checking, but none capable of dealing with rich formalisms have scenario generators, to my knowledge.

7 Limitations and Future Work

The most basic limitation of SGEN^2 is that it is fundamentally a hill-climbing algorithm. In particular, there are examples in the case study which are easily solved by merging two scenarios from the library, but which SGEN^2 cannot find. As an example, one scenario achieves a particular conjunct set halfway through its event sequence, but then has several more steps that are removed by BACKPROP* as irrelevant

to achieving that set. It turns out they are necessary, however, if one wishes to later merge it with a second scenario achieving the rest of the goal. (These "extra" steps are things like hanging up a phone after activating a feature, because a subsequent scenario must start from the idle state.)

SGEN^2's power comes from the richness of the scenario library; it is therefore likely to be more useful in development processes and environments that encourage the formalization of such scenarios. SGEN^2 provides, perhaps, a new argument in favor of integrating formal scenarios into the software process.

SGEN^2 is still in its early youth, and there are many ways it can be improved. For example, in its search, SGEN^2 only considers the first PGS having a given sat-set. A better, but more expensive, approach is to try a PGS if and only if its BACKPROP*-generalization is not isomorphic to one seen previously. The effect of this on run-time must be monitored, however. MERGESCENARIOS, being the time bottleneck on failing cases, may profit from more work on limiting its search. MINELIBRARY needs to search fewer cases when the predicate takes many parameters.

Of course, results from one case study are not conclusive, so future work should investigate SGEN^2's effectiveness on other domains and systems.

8 Conclusions

A scenario generation tool can be useful in a specification modeling tool suite, in focusing attention on design errors demonstrably present, in helping communicate errors in the requirements, and even in implementing online help systems. Exhaustive-search approaches, such as model checking, while useful, are not tractable in rich formalisms allowing more direct system models to be expressed. SGEN^2 is a heuristic approach to a highly uncomputable problem, based on the simple idea of piecing together partially satisfying scenarios from the requirements library, using explanation-based generalization to abstract them in order to be able to coinstantiate them. Results from the case study are encouraging; SGEN^2 seems to succeed often enough to be useful and yet be efficient enough to be engineered into an interactive tool. While the work needs further empirical validation, it seems promising and should be pursued.

References

[1] R.Alur, L.Jagadeesan, J.Kott, J.Von Olnhausen; Model checking of real-time systems: a telecommunications application; In *Proc. 19th Intl. Conf. Software Eng.*, 1997, ACM Press, 514–524.

[2] R.J. Anderson, P. Beame, et al; Model-checking large software specifications; In ACM SIGSOFT Software Eng. Notes 21(6), Nov. 1996, 156–166.

[3] R.S. Boyer & J Moore; *A computational logic handbook*; Academic Press, 1988.

[4] T. Bultan, R. Gerber, & C. League; Verifying systems with integer constraints and boolean predicates: a composite approach; In *Proc. 1998 Intl. Symp. Software Testing and Analysis*, ACM SIGSOFT Software Eng. Notes 23(2), 113–123, 1998.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, & L.J. Hwang; Symbolic model checking: 10^{20} states and beyond; *Info. and Comput. 98*, 142–170, 1992.

[6] E.M. Clarke, J.M. Wing, et al; Formal methods: state of the art and future directions; *ACM Comput. Surv. 28(4)*, December 1996, 626–643.

[7] N. Griffeth, T. Ohta, J-C. Gregoire, & R. Blumenthal; FIW'98 Feature Interaction Detection Tool Contest; `http://www.tts.lth.se/FIW98/contest.html`

[8] G. DeJong and R. Mooney; Explanation-Based learning: an alternative view; *Machine Learning* 1 (2) (1986) 145–176.

[9] R.E. Fikes, P.E. Hart, and N.J. Nilsson; Learning and executing generalized robot plans; *Artificial Intelligence* 3 (1972) 251–288.

[10] A. Gotlieb, G. Botella, & M. Rueher; Automatic test data generation using constraint solving techniques; In *Proc. 1998 Intl. Symp. Software Testing and Analysis*, ACM SIGSOFT Software Eng. Notes 23(2), 53–62, 1998.

[11] R.J. Hall; Systematic incremental validation of reactive systems via sound scenario generalization; *J. Automated Software Engineering 2(2)*, 131–166; Norwell, MA: Kluwer Academic, 1995.

[12] R.J. Hall; Reactive system validation using automated reasoning over a fragment library; in *Proc. 1997 IEEE Automated Software Engineering Conference (ASE'97)*. IEEE 1997.

[13] R.J. Hall; How to avoid unwanted email; *Comm. ACM 41(3)*, 88–95, March 1998.

[14] R.J. Hall; Feature combination and interaction detection via foreground/background models; to appear in *Proc. 1998 Intl. Workshop on Feature Interactions in Telecommunications Systems*, IOS Press.

[15] R.J. Hall; Complete case study data for this paper; `ftp://ftp.research.att.com/dist/hall/papers/isat/sgen2-case-study.txt` (1998).

[16] C.L. Heitmeyer, R.D. Jeffords, & B.G. Labaw; Automated consistency checking of requirements specifications; *ACM Trans. Software Eng. and Methodology 5(3)*, 1996, 231–261.

[17] G.J. Holzmann; *Design and validation of computer protocols*; Englewood Cliffs, NJ: Prentice Hall, 1991.

[18] S. Igerashi, R. London, & D. Luckham; Automatic program verification I: a logical basis and its implementation; *Acta Informatica 4*, 1974.

[19] S. Minton; Quantitative results concerning the utility of explanation-based learning; *Artificial Intelligence* 42 (1990), 363–392.

[20] C. Rich & Y. Feldman; Seven layers of knowledge representation and reasoning in support of software development; *IEEE Trans. on Software Eng. 18(6)*, 451–469, June 1992.

A SGEN^2 Pseudocode

Figure 1 gives a pseudocode description of the top level SGEN^2 algorithm. SGEN^2 takes in a goal predicate GP and returns (on success) a predicate group satisfier (pgs), a data type defined in Section 4.1. On success, the returned pgs will satisfy the entire GP. The function SGEN^2-REC is a recursive subroutine that keeps adding scenarios to the current (partial) result until all conjuncts are satisfied. Accordingly, it takes several arguments: the current list of mined PGSs, the abstraction of the current (partial) result scenario (CurE), its constraint predicate (CurPi), the conjunct set it achieves (CurPf), and the actual bindings (CurI) needed to instantiate the scenario.

Note that Satset(p) is a function that returns the set of conjuncts satisfied by a pgs. Function Reduce(PGSList, pred-set) removes from each entry in PGSList the conjuncts in pred-set, so they become PGSs relative to the original goal predicate without the conjuncts in pred-set. Reduce also sorts its output in the same way as MineLibrary. CreatePGS converts the merged concrete scenario into a PGS for CurPf $\cup$ Satset(p).

```
Parameter M : Model
Parameter L : Library

Function SGEN² (GP)
    PGSList := MineLibrary(GP, M, L)
    SatSets := ()
    For each p ∈PGSList, do
        If Satset(p) ∉ SatSets
            Add Satset(p) to SatSets
            [CurPi, CurE, CurI] := BackProp*(p)
            rPGSList := Reduce(PGSList, satset(p))
            If rPGSList is empty
                Return p ; (Success)
            If SGEN²-REC(rPGSList,
                         CurP,
                         Satset(p),
                         CurE,
                         CurI) succeeds
                Return its result. ; (Success)
    Fail

Function SGEN²-REC(PGSList, CurPi, CurPf, CurE, CurI)
    SatSets := ()
    Foreach p ∈PGSList, do
        If Satset(p) ∉ SatSets
            Add Satset(p) to SatSets
            [P, E, I] := BackProp*(p)
            II := CoInstantiate(P, CurP, I, CurI)
            If II≠ none
                mm := MergeScenarios(CurE(II),
                                     E(II),
                                     CurPf,
                                     Satset(p))
                If mm ≠ none
                    mmpgs := createPGS(mm,
                                       CurPf,
                                       Satset(p))
                    rPGSList := Reduce(PGSList,
                                       Satset(p))
                    If rPGSList is empty
                        Return mmpgs ; (Success)
                    [CurP, CurE,CurI] :=
                                BackProp*(mmpgs)
                    If SGEN²-REC(rPGSList,
                                 CurP,
                                 CurPf ∪ Satset(p),
                                 CurE,
                                 CurI) succeeds
                        Return its result. ; (Success)
    Fail
```

Figure 1: Pseudo code description of SGEN^2.