FROM RULE-BASED TO AUTOMATA-BASED TESTING

Kousha Etessami
Bell Labs
Murray Hill, NJ 07974
kousha@research.bell-labs.com

Mihalis Yannakakis
Bell Labs
Murray Hill, NJ 07974
mihalis@research.bell-labs.com

Abstract

Rule based languages have been used extensively to provide a declarative description of causal relationships between events and conditions in a wide variety of systems. On the other hand, automata (finite state machine) based models provide a detailed operational specification of how the system state evolves over time. While rules are a convenient declarative mechanism, a finite automaton is more flexible and far more easily analyzed than a collection of rules with potentially complicated semantics.

In this paper, we address this dichotomy in the context of large scale testing. We describe a (probabilistic) rule based system, which has been developed and used over many years as the testing model of a large telephone switch. One important drawback of the rule-based system is that it is inflexible in terms of the test generation mechanisms it supports, essentially only allowing the generation of random tests.

We have designed and implemented a translation from the rule-based language to a probabilistic finite automaton model, and applied it to the switch model. Our translation makes use of several automata-theoretic algorithms to keep the size of the resulting automata manageable.

The automaton model allows much more flexible and targeted test generation. We have designed and implemented an assortment of targeted test generation algorithms which are applied to the probabilistic automaton.

Keywords: System testing, rule based systems, finite automata.

1. INTRODUCTION

For large complex software systems, ad-hoc testing methods can not be trusted to assure adequate quality. In order to test a system rigorously and systematically, it is necessary to have some kind of model which reflects, if not the structure of the system, at least the legitimate behavior expected of the system. A notable example of a large complex software system is a telephone switch along with the apparatus surrounding it. The code for these systems typically reaches well into millions of lines, with many interacting components.

Finite state machines (of various types) have often been used to model communication protocols and features for testing and verification purposes (Beizer, 1990; Brinksma, 1988; Lee and Yannakakis, 1996b; Holzmann, 1990; Vardi and Wolper, 1986), and commercial testing tools based on them are already in the marketplace. A different kind of modeling mechanism, which has seen widespread use in other areas, is a *rule-based* model. Sets of "if ... then ..." rules are used to specify that if a certain combination of facts/events/actions hold/occur, then a certain other combination of facts/events/actions must also hold/occur. Depending on the context, rules are interpreted either in a passive manner as constraints (properties) of the system, or in an active manner, as directions for action. Rules are used for example in active databases (see, e.g., Widom and S. Ceri, 1996, for a description of a number of such systems), in expert systems and knowledge bases (Ullman, 1988), in workflows (Hull et al., 1999), in policy description languages in various contexts (firewalls, routing, network management), requirement properties (for example, telephony feature properties (Holzmann and Smith, 1999)). Rules can provide a high level description of the causal relationships that must hold in a system in a declarative and localized manner. This often makes them very convenient for users, who only need to isolate individual causal relationships, rather than describe the global behavior of the system in a monolithic fashion, as would be done by, say, a global finite state machine (automaton) model.

However, the descriptive convenience gained by using rule bases comes at a cost: the global behavior of the interacting rules is difficult to predict or to analyze automatically. In the context of testing, this translates to the fact that it is difficult to generate flexible and targeted tests with desired properties based on the constraints specified by the rules.

This is where our work enters the picture, by providing a bridge between the rule-based and automata-based worlds for large scale testing, applied to a real world commercial system. The work was motivated by the testing of a large telephone switch. The switch was developed and has been evolving since the early 1980's, has millions of lines of code, and supports over 400 features. In order to test the switch, in the early 90's the testers developed a rule-based modeling language and tool called TGAP (for Test Generation and Analysis Platform). The TGAP system and the test model for the switch have continued to evolve and grow over the years. The TGAP model is a high level, functional model of the whole system which reflects the expected behavior of the system from the point of view of various participants in communication scenarios, as well as the underlying provisioning information (types of trunks, devices etc.). Participants include, for example, a caller, a callee, an operator, and so on, as well as administrative and setup operations.

The specification mechanisms of TGAP, in rough outline, consist of (hierarchical, probabilistic) *graphs* for all the participants, and of a large set of rules (in the form of "if .. then .." conditions) that prescribe precisely how behaviors of one user restrict those of another in valid phone scenarios. The detailed syntax and semantics of the TGAP model will be provided in the next section. These graphs plus rules together define the set of all legal scenarios in the system. The TGAP tool uses the probabilities provided to take a random walk on the graphs, eliminating illegal scenarios and backtracking until it finds a legal scenario, which it then outputs. This process is repeated as often as necessary in order to obtain a sufficient number of test scenarios. The scenarios are then translated into a home-grown test-execution language, via another tool, the *Converter*, that we do not address in this paper; the outcome of the Converter describes in detail a sequence of actions that are to be performed on an actual phone system and their expected outcomes.

As it stands, the testing platform suffers from several weaknesses stemming from its rule based design. The first of these is that, the only way to access the information in the test model is to generate a random scenario via the random walk outlined above. This limits significantly the tester's ability to perform *targeted* tests that exercise particular features of the system. Since exhaustive testing is impossible given the enormous number of possible scenarios (in fact, infinitely many when cycles are allowed in the graphs), testers need to be selective, using their knowledge to focus on important and/or problematic components. Automated analysis of the behavioral model can also help in designing tests that meet particular coverage criteria.

We describe a translation from the rule-based system to a probabilistic finite state automaton model. Once we established that such a translation is possible even in principle, it became clear that because of the large number of interacting rules in the TGAP model for the switch, there was a great potential for state explosion. The biggest challenge we faced in our translation was to keep the size of the resulting automata manageable. We designed a new incremental scheme in which we alternate between building up the automaton and doing minimization. Along the way, we made interesting use of several automata theoretic algorithms. The result is that the translation has already been used to successfully translate the model for the switch into a probabilistic finite state machine that is feasible, only 230,000 states and 480,000 transitions.

Once an automaton for the test scenarios became available, it enabled many forms of analysis and targeted test generation. We developed a host of algorithms to apply to the probabilistic automaton for a number of tasks such as: Generate the n most probable scenarios, to optimize coverage for a given number of scenarios. Compute statistics on the profile of the model and its coverage, for example count the number of possible scenarios Restrict the model to a subsystem or feature of interest specified in terms of keywords or patterns, and compute targeted scenarios that optimize desired test criteria. These algorithms are critical in order for testers to be able to optimize and target their tests to get adequate coverage and achieve desired testing goals.

2. SYNTAX AND SEMANTICS OF TGAP

As described in the introduction, a TGAP model consists of hierarchical *graphs* and *rules*. There is a set of users, such as caller, callee, operator, administrators etc. Each user has an associated user type, and for each type a hierarchical graph captures the possible behaviors for that type of user. The nodes of the graph are annotated with *keywords* that describe at a high level the events at the user. Rules dictate how behaviors of one user in a call constrain the behaviors of other users. They also dictate how call provisioning constraints, such as the kind of phone device a user has, limit the features accessible to that user.

The aim of graphs + rules is to define a set of valid *scenarios*. Informally, a scenario is a description of the precise sequence of events that occurs at each user in one use of the system, along with peripheral information such as what phone devices each user is using. Formally, a scenario is represented as a sequence of keywords from an alphabet Σ, separated by the special delimiter keyword "$\longrightarrow$", i.e., a string from $(\longrightarrow (\Sigma^+))^+$. The sequence immediately after each "$\longrightarrow$" represents the behavior witnessed by one user in the scenario, and is called a *transaction*. Note that in this formalism, the executions of different users are listed sequentially one after the other, rather than in the usual interleaved time order. One consequence of this is a succinctness in representation: a scenario represents really a potentially exponential set of possible interleavings of the individual user transactions. As we mentioned in the introduction, a TGAP scenario is processed by a Converter, which interleaves the transactions in some order (and fills in the details).

Now we describe the syntax of graphs and rules, and we then put the two together to describe how their combined semantics defines a set of valid scenarios.

Hierarchical graphs

Given an alphabet Σ of *keywords*, a TGAP (hierarchical) *graph* over Σ is a tuple $H = (G_1, G_2, \ldots, G_n)$ of *pure graphs*, where each pure graph, $G_i \doteq \langle W_i = V_i \cup \{g^i_1, \ldots, g^i_{m_i}\}, E_i, l_i, b_i, p_i, s_i\rangle$, consists of

1 A set of nodes W_i, partitioned into ordinary nodes V_i, and some special graph nodes $\{g^i_1, \ldots, g^i_{m_i}\}$.

2 An edge relation $E_i \subseteq V_i \times W_i$, forms a directed graph on $W_i \times W_i$, where the special graph nodes $\{g^i_1, \ldots, g^i_{m_i}\}$ are sinks (i.e., have out-degree 0). There is a single source node, s_i (i.e., a node with no incoming edge), which is the start node of the graph G_i.

3 A labeling function $l : V_i \to \Sigma$ that assigns a keyword to each ordinary node.

4 A labeling $b_i : \{g^i_1, \ldots, g^i_{m_i}\} \mapsto \{i+1, \ldots, n\}$ that assigns to each special graph node in G_i an index $j > i$ for another pure graph G_j. Reuse is allowed, i.e., many special nodes may be mapped to the same graph G_j.

5 A probability function $p_i : E_i \to [0, 1]$ that assigns a probability to each edge such that for each non-sink $v \in V_i$

$$\sum_{\{(v,w) | (v,w) \in E_i\}} p((v, w)) = 1 \tag{1}$$

Now, each graph can be converted to an ordinary directed graph using the following *flattening* process:

Let $G'_n := G_n$.
For $i = (n-1)$ down to 1 do:
For each edge $(v, g^i_k) \in E_i$, let $j = b_i(g^i_k)$, and substitute $(v, g^i_k) \in E_i$ by the edge (v, s'_j), where s'_j is the start node of G'_j. Let G'_i be the resulting flattened graph.

Note that the final result, G'_1, is a labeled flat graph with consistent node probabilities, meaning equation 1 holds for each non-sink node v.

Note also that in this case the flattening to a directed graph does not blow up the size, i.e., the size of the final flat graph G_1' is linear in the size of the hierarchical graph (the sum of the sizes of the pure graphs G_i). This is due to the limited use that is made of the hierarchy, since only sinks can be expanded with other graphs. Conciseness over graphs would be gained if internal nodes, rather than sinks, were also allowed to be expanded (see, e.g., (Alur and Yannakakis, 1998)). The main reason for using hierarchy in the TGAP platform is that it allows the graphs to be presented with GUIs in a clean partitioned way. We remark that in the tool's terminology, the hierarchical graph is termed a *tree* (although cycles are allowed in the tool and 'tree' is defined in the man pages as a 'cyclic directed graph'), partly because for the main switch system which has been modeled the hierarchical graph is actually a hierarchical tree without cycles. (For trees, hierarchy even only at sinks gives an exponential succinctness over flat trees, though they are equivalent in expressiveness to directed acyclic graphs.) We take some advantage of the acyclicity in our implementation, but our algorithms remain valid or can easily be modified to the more general cyclic setting.

Given a flattened graph, a sequence of labels along a path from start to sink node will be called a *transaction*. *Scenarios* will be built up out of transactions from different user graphs as prescribed by the rules.

Rules

TGAP rules are quite complicated. Rather than provide the formal syntax of all rules, we will describe the important details of all essential rules. The rules specify how to build a scenario: how to start a scenario, i.e., with which user and what conditions on the first transaction; how to continue a partial scenario, i.e., conditions on the next user and the next transaction depending on the history; and when to terminate a scenario, i.e., conditions on whether a scenario is complete or has to continue. There are five essential types of rules:

1. START_USER: This rule describes which user starts the scenario, with given probabilities. There is only one such rule. It takes the form: START_USER { user_1 prob_1 … user_k prob_k }

2. FIRST_TRANS: These rules describe constraints, given by right-patterns, on what may appear in the first transaction of a scenario. The format of the rule is: FIRST_TRANS right-pattern

3. MORE_TRANS: The format is: left-pattern MORE_TRANS
The meaning of the rule is "Any partial scenario which satisfies the left-pattern is not complete, i.e., more transactions are required in this scenario."

4. NEXT_USER: left-pattern NEXT_USER { user_1 prob_1 ... user_k prob_k }. This rule means that, for those partial scenarios satisfying the left hand side, and requiring more transactions, the next transaction must come from the given user types user_i, with the given probabilities prob_i, respectively.

5. NEXT_TRANS: left-pattern NEXT_TRANS right-pattern

The meaning of these rules is that, if a partial scenario satisfies the left side, and requires more transactions, then the next transaction must satisfy the right-pattern. When a partial scenario satisfies the left side of several NEXT_TRANS rules, all of these rules apply, in effect creating a conjunction of the right hand sides.

We will not formally define the format of left-patterns and right-patterns. They are particular forms of regular expressions, augmented with extra expressive features such as specifying the number of transactions, negations, and so on.

There is another rule which allows us to define aliases to denote any of a number of keywords: aliasname ALIAS keyword_1 ... keyword_k. Thus, whenever the word aliasname appears, it is to be interpreted as a disjunction $\bigvee_i \text{keyword}_i$. Aliases can be defined in terms of prior aliases.

How graphs and rules are put together

As we have seen, a TGAP model is given by a sequence $\mathcal{H} = \langle H_1, \ldots, H_r \rangle$ of hierarchical graphs, one corresponding to each user type, as well as a sequence $\mathcal{R} = \langle R_1, \ldots R_d \rangle$ of rules. For the rest of this section, view the hierarchical graphs in *flattened* form, H'_i. In order to simplify our treatment here, we will first ignore probabilities, and only consider a scenario valid if it can be generated by the model $(\mathcal{H}, \mathcal{R})$ with non-zero probability.

Given $(\mathcal{H}, \mathcal{R})$, a string $w \in (\rightarrow \Sigma^+)^+$ consisting of a sequence of transactions $w_1 \ldots w_m$, where each $w_i \in (\rightarrow \Sigma^+)$, is called a *valid scenario* if it satisfies the following criteria:

1. The first transaction w_1 in w must exist as a source-sink path with non-zero probability in the graph H_i, such that the user type user_i is selected with non-zero probability in the START_USER rule.
2. w_1 must also satisfy the right-patterns given by all FIRST_TRANS rules.
3. For $i < m$, $w_1 \ldots w_i$ must satisfy some MORE_TRANS rule's left-pattern.
4. For $i < m$, w_{i+1} must exist as a source-sink path in a graph H_i such that user_i is selected with non-zero probability in the *first* NEXT_USER rule in $\mathcal{R}$ whose left-pattern is satisfied by $w_1 \ldots w_i$.

5. For $i < m$, all the NEXT_TRANS rules whose left-patterns are satisfied by $w_1 \ldots w_i$, must have right-patterns that are satisfied by w_{i+1}.
6. There must be no MORE_TRANS rule whose left-pattern is satisfied by w.

What is needed in order to complete the above description is to describe how probabilities affect the generation of partial scenarios along the way. Although this is not difficult to do, the details of such a description would take us too far afield, so we only mention that one can proceed in a depth-first fashion to construct the scenario, picking children according to edge probabilities, and backtracking when a rule is violated.

3. (GRAPHS + RULES) → PROBABILISTIC AUTOMATA

Although the graphs + rules model is complex and has many details, it is no more powerful in expressive power than a probabilistic finite state machine (automaton) model. The following theorem establishes this for finite automata, again ignoring probabilities for now for simplicity.

Theorem 1 *For a TGAP model $\mathcal{M} = (\mathcal{H}, \mathcal{R})$, the set of valid scenarios consistent with $\mathcal{M}$ is a regular language. That is, we can construct a finite automaton $A_{\mathcal{M}}$ that accepts exactly the scenarios generated by $\mathcal{M}$.*

Proof. We only sketch the proof. The basic idea is that, since left-patterns and right-patterns in rules are given by variants of regular expressions, each rule can be represented by one or two deterministic finite automata, depending on whether the rule has both a left and a right side, and the entire model can be represented as a kind of "product automaton" of all the rule automata along with the user graphs.

In more detail, consider for example a NEXT_TRANS rule

$R =$ *left-pattern* NEXT_TRANS *right-pattern*

The rule can be represented by two coupled automata A_{left} and A_{right}, where A_{left} monitors the input, and goes to a special state when the *left-pattern* is satisfied. A_{right}, on the other hand, monitors the state of A_{left} and begins processing *right-pattern* as soon as A_{left} enters the special state.

The automaton $A_{\mathcal{M}}$, is built up from these component automata as follows. A state of $A_{\mathcal{M}}$ is given by a tuple $(q, i, a_1, a_2, \ldots)$, consisting of the current node q in some user graph, the index i of the current transaction, and the state a_j of each rule automaton. The transitions of $A_{\mathcal{M}}$ reflect the processing of the input, in lock step (synchronously), by all the rule automata, as we advance down some user graph. Transitions on the delimiter symbol $\rightarrow$ depend on being at a terminal node of a user graph and appropriate accepting states of the relevant rule automata. Acceptance is determined when we have reached a sink node on a user graph, and all rule automata are in an "agreeable" state (including, MORE_TRANS automata, which must indicate that no more transactions are required). ■

The difficulty with the above proof is that, even ignoring probabilities, it results in a hopelessly huge automaton for any reasonable model. To give a sense of the problem, consider the model for the large switch which has been our focus in this project. It has 29 users (there are 11 user types), most user graphs contain up to 100,000 nodes after flattening, and roughly 500 rules, some consisting of patterns with complicated regular expressions (including negation). Even assuming, very optimistically, that each rule could be modeled by an automaton of at most 10 states, the construction in the above proof would build a global automaton $A_{\mathcal{M}}$ with 10^{506} states!! So, yes, it is a *finite* automaton, but just barely so. It was a priori not at all clear what the size of such an equivalent automaton would be, and furthermore if it were small, whether and how one could construct it without blowing up the size in the process. This required a number of optimizations which we outline below.

Our translation produces a probabilistic automaton $A = \langle S, \delta, l, start \rangle$ where S is a set of states, $\delta \subseteq S \times [0,1] \times S$ is a transition relation with associated probabilities for each edge, such that for all v

$$\sum_{\{(v, p_{(v,w)}, w) | w \in S\}} p_{(v,w)} = 1.0$$

$l : S \to \Sigma \cup \{\longrightarrow\}$ is a labeling that assigns a keyword to each node. (For convenience, keywords label states rather than edges in these automata.)

Given the model $(\mathcal{H}, \mathcal{R})$ we obtain the probabilistic automaton $A_{\mathcal{M}}$ according to the following general outline:

1 For each rule, with general form "If exp1 then exp2", convert both exp1 and exp2 to (non-deterministic) automata A_{exp1} and A_{exp1}.

2 Incrementally apply each rule, in their prescribed sequential order, to the partial automaton built thus far, as follows

> intersect the current automaton with A_{exp1}, and on those paths that satisfy A_{exp1}, take the product with A_{exp2}, removing the paths not satisfying exp2.

3 minimize the result after each rule is applied.

This process involves different actions for different rule types, so what we've described only serves as a rough guideline. In essence, the algorithm consists of a process of repeated "producting" and minimization of an automaton. Each of the steps outlined requires us to take several measures to insure that the automata do not get too large. We sketch below some the measures. For most rules, Step 2 is further broken down into substeps depending on the rule and the patterns that appear in it. Since many of the rules have complicated expressions with many patterns, instead of constructing one automaton for the whole expression $exp1$ (or $exp2$), which can be quite large, we break the expression down into suitable subexpressions, and apply the producting followed by minimization routine for each subexpression.

Furthermore, when converting rule expressions to automata, we will need to address the problems we discussed arising from the use of negation and aliases in the expressions in rules. Two techniques we use for this purpose are: first we do not directly determinize the expression automata; we leave them nondeterministic and resolve nondeterminism on the fly while producting with the system automaton. Second, we do not explicitly produce transitions of the automata for all input symbols (there are just too many keywords). Rather we specify them implicitly, using formulas as labels for rule automata. Thus, rather than ordinary automata, we will convert exp1 and exp2 in a rule into *special automata*. Let $B(\Sigma)$ denote the set of boolean formulas over the alphabet Σ (in fact, DNF formulas suffice for our purposes, but we give the more general definition). A special automaton has the form $A_{exp_i} = \langle S', \delta, l, F, start' \rangle$ where, S' is the set of states, $\delta \subseteq S' \times S'$, $F \subseteq S'$ is a set of final states, but this time the labeling function $l : S' \rightarrow B(\Sigma)$, assigns to each state a *boolean formula* over the keywords, rather than one particular keyword.

The interpretation of such an automaton is as follows. A string $w = w_1 \ldots w_k$ over Σ is in the language $L(A_{exp_i})$ of the automaton iff there is a sequence of states $s_0, s_1, \ldots, s_k$, beginning with the start state $start'$ $= s_0$, such that keyword w_i satisfies $l(s_i)$, in the sense that if we view w_i as a assignment of truth values to Σ which assigns `true` to w_i itself and *false* to every other keyword, $l(s_i)$ is satisfied by this assignment.

Finally, we product A_{exp_i} with the current scenario automaton A. Again, we have to be very careful if we want to maintain feasibility, and we also have to maintain probabilities properly. A is very large, so we do not wish to incur the worst case $|A| \times |A_{exp_i}|$ blowup, which is typically unnecessary. To do this, our producting algorithm produces states "on-the-fly", in a way reminiscent of on-the-fly algorithms for state exploration in model checking (see, e.g., Holzmann, 1990). Second, because A_{exp_i} is non-deterministic, we need to do something akin to subset construction while our producting is proceeding, in order to obtain a final result in the deterministic form of A.

The basic outline of the algorithm for producting the system automaton with a special expression automaton, which proceeds via depth-first exploration of the product, is as follows:

automaton_producting_algorithm {
  Initialize stack $Z \leftarrow [(start, \{start'\})]$.
  (we assume $start$ and $start'$ are a consistent pair)
  `While` Z is not empty `do`
   remove (s,X) from atop stack Z
   `foreach` child t of s `do`
    find the set X' of next-states of X in A_{exp}
    which are *consistent* with t, meaning the label of t
    satisfies the formula labeling each $x' \in X'$.
     `if` (t, X') is not already in hash table `then`
      add (t, X') to hash table,
      place (t, X') atop stack Z,
      and create the edge $((s, X), p_{(s,t)}, (t, X'))$
}

Note that in the algorithm the probability of a new edge is inherited from the graphs, thus maintaining the sequence of weight on a given scenario. But the process no longer assures that probabilities will sum to 1, so we will need to do some rescaling. There is a subtle point here regarding the correct way to do this rescaling; we omit the details.

Finally, after each producting, we run a minimization algorithm on the resulting automaton A' in order to get rid of redundancies that might have been created in the process of producting. Note that our minimization algorithm will need to preserve probabilities. What we use is a bisimulation reduction for probabilistic automata (see, e.g., Baier et al., 2000). We describe the algorithm for arbitrary automata, although we use the acyclicity of the switch automaton to make the algorithm run more efficiently (in fact in linear time).

state_minimization_for_probabilistic_automata {
  Initialize: $\forall$ states s, $color(s) := l(s)$.
  (initially "color" the states according to the label of the state)
  `Repeat`
    $\forall$ states s,
     $new\_color(s) := (color(s), \{(c, p_c) \mid p_c = \Sigma_{\{t \mid color(t)=c\}} p_{(s,t)}\})$
    $color := new\_color$
    (rename colors, using sorting, to represent them concisely)
  `Until` the total number of colors does not increase
}

The resulting automaton has one state for each color class, and its transition relation can be read off from the final coloring.

4. TRANSLATION IMPLEMENTATION, AND THE SWITCH MODEL

A complete translator from TGAP models into probabilistic automata has been implemented. The implementation was carried out in the programming language ML, using roughly 7000 lines of code. We have used the translator to translate the large model for the switch. The dimensions of this model were as follows: There are 266 keywords, plus roughly 100 aliases. The largest flattened graph contains, approximately $100,000$ nodes (this gets reduced dramatically, to 4700 states, after minimization). There are approximately 500 rules.

Our translation produced an automaton having only $230,000$ states and 480,000 transitions. This is well within the realm of feasibility, as many of the test generation algorithms we apply to this automaton require only linear time. The one drawback to the translation is that it is currently very slow. However, since all of the analysis and test generation is performed directly on the resulting automaton, translation is an off-line cost that is only incurred once, performed prior to testing when the model is completed. Hence, it is a one-time penalty that is reasonable to pay.

5. TARGETED TEST GENERATION ALGORITHMS

The ruled-based TGAP system provided only random sampling as a test generation capability. The user requests a desired number N of scenarios, and the system attempts to generate N random scenarios using the user models, the associated probabilities in the operational profile, and the rules. We say only "attempts" because the system may not (and often does not) succeed in producing N scenarios, and may return a smaller number. The reason is that the system tries to construct the scenarios incrementally walking through the graph models and applying the rules. As a consequence it is possible that a partial scenario cannot be extended to a complete legal scenario, because of the rules; when the scenario fails to complete, the TGAP system backtracks and tries another route. After a fixed number of failures, the system terminates the search. Note that it is possible that there are not even N legal scenarios, for example if one adds rules in an effort to narrow down the scenarios in order to restrict to a subsystem of interest. However, there is no way to tell directly from the rules whether this is the case.

Moving to an automaton based model enables a number of testing capabilities with better test selection, analysis of coverage, targeting to subsystems of interest, and enforcement of desired test criteria. We have developed and implemented several algorithms and applied them to the switch automaton. We summarize below some of the tasks addressed, and our experience in the application to the switch model, without going into the details of the algorithms, which will be given elsewhere.

Counting scenarios. This is useful when a user asks for N scenarios, (1) to determine whether there exist that many scenarios, without searching in the dark, and (2) to determine which fraction of the total are accounted for by the N scenarios. Of course, the number of scenarios is finite only if the automaton is acyclic, as is the case with the switch automaton. The number of legal scenarios in this test model is more than 10^{20} (the precise number is 905,772,615,646,147,548,919); Obviously, it is impossible to generate that many items. However, note that we can count the scenarios without explicitly enumerating them. The algorithm runs in linear time in the size of the automaton and takes just a few seconds for the switch model.

Sampling scenarios. That is, generate N random scenarios according to the probabilities in the model. This is the only kind of testing that the TGAP system allows. Having the automaton makes this problem very easy; the algorithm runs much faster because it does not have to match and apply the rules online at runtime, and does not have to worry about dead ends, i.e. work on partial scenarios that don't lead anywhere. A random scenario corresponds to a random walk on the Markov chain represented by the automaton. Thus, one can generate thousands of random scenarios in seconds. Besides, producing the scenarios, we can also compute their associated probabilities, and thus calculate the fraction of the probability space that is covered by the produced test set.

Most probable scenarios. That is, generate the N scenarios with the highest probability. The rationale for this kind of test generation is to maximize the coverage of the operational profile (as measured by the probabilities) with a limited number of tests. Given the size of the model, obviously we need to pay attention to the efficiency. A naive approach takes time $O(N \cdot input)$. Our algorithm takes essentially linear time in the input (the size of the automaton) and the output (the N tests). The algorithm also runs fast in practice, comparable to the simple random sampling. For the switch automaton, on a time shared 250MHz MIPS processor, after reading the input and setting up the data structures (9 secs.) it generates the maximum probability scenarios in order at the rate of more than 3,000 scenarios per second. Again, besides the scenarios themselves we can also compute the associated probabilities to analyse the coverage, obtaining a profile of the probability distribution of the scenario space. The following table shows some data for the switch automaton; the second column gives the probability of the Nth test and the third column the coverage (cumulative probability) of the first N tests. We computed statistics for random testing, and noted that it requires 8-10 times as many tests to achieve the same coverage as the maximum probability method.

test	probability	coverage
100	0.00049	13 %
500	0.00013	22 %
1000	5.54e-05	26 %
5000	1.18e-05	35 %
10000	5.68e-06	39 %
50000	1.12e-06	48 %
100000	5.57e-07	52 %

Often one wants to target tests to specific subsystems or features of interest. The keywords are used for this purpose; some keywords may be irrelevant to the subsystem, while others represent activities of the feature or subsystem, thus, are relevant and are expected to appear often in scenarios of the subsystem. For example, to test functionalities related to call centers, testers compiled a list of special 'relevant' keywords that involve call center activities. The objective then is to generate relevant scenarios that exercise the parts of the system that have relevant keywords.

Most relevant scenarios. For a given list of relevant keywords, we can assign to each scenario (path through the automaton) a relevance score: the number of its nodes with relevant keywords. More generally, we can assign weights to the keywords according to their relative importance, and the relevance score is then the sum of the weights of the scenario's keywords. For a given number N of desired scenarios, we can generate the N scenarios with highest relevance score. The algorithm runs essentially in linear time in the input and output as in the case of probabilities and is also fast in practice.

Minimum cover. Find a small number of scenarios that cover all the relevant keywords, i.e., each relevant keyword appears at least once. This problem was addressed in Lee and Yannakakis, 1996a, where algorithms were proposed to find a near minimum set (finding the minimum set is NP-hard). For example, 28 tests suffice to cover all keywords of the whole switch automaton, and 5 tests cover the special keywords in the call center list. Similarly, we can find a small test set that covers all the nodes of the automaton or the original user graphs that are labeled with relevant keywords.

Balanced relevant cover. Suppose we can afford to run N tests (which may well be more than the minimum needed for simple coverage). We could find the N scenarios with highest relevance score but it is possible that all of them involve only a subset of the special keywords. Usually, it is desirable to cover all the relevant keywords, and furthermore to cover them multiple times in a roughly balanced manner. We have developed an algorithm for this purpose; it is based on the above algorithm for computing the most relevant scenarios but it uses a dynamic reweighting of the keywords through the computation to bias the generation towards a balanced coverage.

Pattern-based Restriction. A more refined way to characterize subsystems and features is by using regular expression patterns, similar to the rules. The patterns can be used to restrict the automaton, that is, we can combine the system automaton with the pattern automaton to produce a new automaton that characterizes the subsystem. Any of the above algorithms then can be applied to the resulting automaton to produce desired targeted tests for the subsystem.

6. CONCLUSION

We have described an efficient translation from a rule-based behavioral testing model to a probabilistic automaton. Automata are a more familiar and well-studied model which provides much more flexibility in test generation. The motivating application for this work was the testing of a major switch.

Our goal was to enable the application of state-of-the-art automata based testing algorithms while leveraging the existing rule-based model that had been developed over many years. Although feasibility of the translation from a rule-based to automata-based model was a big open question given the scale of the problem, the experiment turned out favorably, and we were able to generate a reasonable size equivalent automaton. We also developed several efficient algorithms for targeted test generation and coverage analysis which went well beyond anything that was possible with the rule-based system.

As we mentioned in the introduction, rule-based models are used widely to specify systems in a wide variety of domains. On the other hand, automata-based models are more convenient for verification and testing purposes; indeed most of the automated techniques in conformance testing and model checking are based on automata (finite state machines). Bridging the rule and automata paradigms, as we have done in this paper, will bring these testing techniques to bear on a variety of other domains.

References

Alur, R. and Yannakakis, M. (1998). Model checking of hierarchical state machines. In *Proc. 6th ACM Symp. on Foundations of Software Engineering*, pages 175–188.

Baier, C., Engelen, B., and Majster-Cederbaum, M. (2000). Deciding bisimlarity and similarity for probabilistic processes. *Journal of Computer and System Sciences*, 60(1):187–231.

Beizer, B. (1990). *Software Testing Techniques.* Thompson.

Brinksma, E. (1988). A theory for the derivation of tests. In *Proc. IFIP WG6.1 8th PSTV.*

Holzmann, G. J. (1990). *Design and Validation of Protocols.* Prentice-Hall.

Holzmann, G. J. and Smith, M. H. (1999). A practical method for the verification of event driven systems. In *Pro.c Intl. Conf. on Software Engineering*, pages 597–608.

Hull, R., Kumar, B., Llirbat, F., Simon, E., Su, J., and Zhou, G. (1999). Declarative workflows that support easy modification and dynamic browsing. In *Proc. WACC.*

Lee, D. and Yannakakis, M. (1996a). Optimization problems from feature testing of communication protocols. In *Proc. Intl. Conf. on Network Protocols*, pages 66–75.

Lee, D. and Yannakakis, M. (1996b). Principles and methods of testing finite state machines - a survey. *Proc. of IEEE*, 84(8):1090–1123.

Ullman, J. D. (1988). *Principles of Database and Knowledge-base systems.* Computer Science Press.

Vardi, M. Y. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In Kozen, D., editor, *First IEEE Symp. on Logic in Computer Science (LICS)*, pages 322–331.

Widom, J. and S. Ceri, E. (1996). *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann.