

Scenarios, State Machines and Purpose-Driven Testing

Thomas A. Alspaugh, Debra J. Richardson and Thomas A. Standish
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{alspaugh, djr, standish}@ics.uci.edu

Abstract

Testing is a necessary but frequently expensive activity that is needed to ensure software quality. For large, complex systems, testing based on covering all control flow or all data flow paths is intractable. But focusing on tests that are purpose-driven, namely on tests that are derived from system requirements and that test whether requirements goals are met, significantly reduces the size of a “complete” test suite for the system while simultaneously increasing confidence that the system performs as expected.

Scenarios and state machines provide a useful framework for modeling and analysis of purpose-driven testing. Scenarios are sequences of events that represent purposeful uses of a system (or of its components, to any desired degree of detail). State machines, in the form of recursive transition diagrams, can model the successive refinement of requirements goals into architectures and implementations, and testing them using purpose-driven scenario-based tests provides early validation of that refinement. Formulating sets of scenarios that capture and represent a complete-enough set of requirements ensures that a test suite covering them explores all important regions of a system’s state space. The scenario-based tests will predict with high confidence which system goals have been met, and, certainly, which have not. This position paper sketches elements of our approach to purpose-driven testing using scenarios and state machines.

1 Introduction

We believe purpose-driven testing can be more efficient and more tractable than traditional testing methods based on exploring all control or data flow paths through code. This paper describes an approach to purpose-driven testing that uses scenarios and state machines to model purpose-driven tests.

One of the main challenges in testing is to explore enough of the system’s state space to gain sufficient confidence in the “correctness” of the system, without

having to perform so many tests that testing becomes intractable. The software testing research community has developed many approaches for testing based on the control flow, data flow and dependency relationships in the system’s code. The most basic are the myriad of *code-based coverage metrics* [CPRZ89] and *test generation techniques* [RC85]. Approaches based on the code for a system typically suffer from intractability for large systems, due to the many distinct paths through the code whose effects are not significantly different. Exploring all flows of control or data results in much low-yield testing – low-yield in the sense that it rarely results in greater confidence and few, if any, additional errors are detected.

We have come to believe that scenario networks (organized sets of scenarios) are of enormous benefit in identifying the interesting “paths” for testing. A scenario network defines the set of alternative ways to satisfy the relevant system requirements goals. By using the scenarios to build test sequences – scenario-driven testing – the more interesting paths are selected for exploration. If the scenario network represents a complete-enough set of requirements, then we may have high confidence that all goals, and hence the important regions of the system’s state space, have been explored. Of course, we are not claiming to guarantee correctness by goal/scenario coverage, but rather that scenario-driven testing will be able to predict, efficiently and with high confidence, which system goals have been met and which have not, and is able to do so more effectively than code-based testing approaches.

A useful starting point is to formulate a theory that covers many kinds of software systems and yet is based on just a few simple concepts, in order to attain the required precision in a comprehensible way. For this purpose, we have chosen to formulate the (level-0) core of our theory based on simple formulations of scenarios as sequences of events, and on state machines in the form of recursive transition diagrams (RTDs) that can call one another recursively [Woo70].

The remainder of this paper is organized as follows. Section 2 gives a brief overview of five essential elements of our theory – requirements, goals, representation domains, scenarios, and recursive transition diagrams (RTDs). Section 3 explores two small examples – an Automated Teller Machine (ATM) for a bank, and finding the least common multiple of two positive in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

4th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools, 21 May 2005, St. Louis, MO, USA.
Copyright © 2005 ACM ISBN 1-59593-130-9...\$5.00.

tegers. Section 4 comments on our plans for future work.

2 Brief Overview: Requirements, Goals, Domain Analysis, Scenarios, and RTDs

We assume the software system we are testing is intended to meet a set of requirements in the form of a (directed acyclic) goal graph G and scenarios operationalizing those goals. The top-level goals in G (those with no predecessors) are stakeholder goals, and the lowest-level leaf goals (those with no successors) express functional requirements that are to be satisfied by implemented software components. The immediate descendants g_1, g_2, \dots, g_n of a goal g are called instrumental sub-goals of g . The achievement of such instrumental sub-goals may *support* the achievement of g , or they may be combined in some relationship $R(g_1, g_2, \dots, g_n)$ that *satisfies* g . For example, R may be a logical relationship (such as $g_1 \wedge g_2 \vee \sim g_3$), or a goal prerequisite relationship (such as $(g_1 \text{ before } g_3) \wedge (g_2 \text{ before } g_3)$), or a plan (such as *repeat* g_1 *until* g_2). To make G 's general description even more complicated, yet more accurate, sometimes goals in G that are not immediate descendants of a common parent goal can be involved in relationships such as tradeoffs and goal conflicts.

We assume that a domain analysis has been performed that establishes a representational medium D in which the software can be implemented. Domain D contains objects that have properties, states that contain objects related to one another by relations, and operations on groups of related objects that transform initial states into successor states. States also have properties that are expressed as pre-conditions and post-conditions for state transformation operations. A goal is a desired state of affairs. A given software component p (a procedure or function) achieves a goal g if (as in program verification theory) p transforms an initial state satisfying a pre-condition into a final state that satisfies a post-condition corresponding to goal g .

Consider, now, a family of recursive transition diagrams (RTDs) defined as follows. An RTD is a finite state machine m , such that:

- m has a single start state and a single final state.
- The states of m other than the start state and the final state are called *internal* nodes.
- m 's graph is a *hammock graph*, meaning that every internal node n of m lies on a directed path from the start state to the final state.
- If a node p of m has a single successor node q , the edge e of m 's graph representing the state transition from p to q is labeled with a goal $g \in D$ or an operation $o \in D$.

- If a node p of m has more than one successor node (q_1, q_2, \dots, q_n) then the edges from p to q_i ($1 \leq i \leq n$) are labeled with predicates in D that are mutually exclusive and exhaustive.
- The machine m can have a (procedure or function) name, it can take parameters, and it can return results.
- The machine m can be labeled with a goal that it achieves or the name of an operation that it performs in representation domain D .

Two particular kinds of RTDs of interest are as follows: (1) RTDs whose edges (i.e., state transitions) are labeled with *goals* are called *goal automata* (or sometimes simply *plans*); (2) RTDs whose edges are labeled with operations that are implemented as procedures (or are themselves actual programming language statements) are called *functions* or *procedures* (depending respectively on whether these RTDs do, or do not, return values when they are called).

Roughly speaking, goal automata represent proto-programs expressing plans that get refined stepwise into actual running unit programs. Stepwise refinement occurs when the goals in a plan are refined into more detailed lower-level plans (or ultimately into functions or procedures) that implement them. In the process of stepwise refinement, we sometimes apply program equivalence-preserving transformations that map given plans/programs into new plans/programs whose structure is more advantageous from the perspective of clarity or efficiency.

A scenario is a sequence of events that corresponds to the purposeful use of a system. One type of scenario – a *requirements scenario* – expresses an example of how a system can be used to fulfill one of its required purposes. Requirements scenarios are expressions of desired or expected behavior that a system is required to exhibit, and, as such, constitute another way to express system requirements (along with requirements goal graphs).

Another type of scenario – an *event-trace scenario* – is collected by tracing the behavior of a running system under specific initial conditions. Such a scenario expresses an actual sample of running system behavior. The event-trace scenario may be annotated with goals retained when higher-level plans were refined into lower-level implementations. These retained goals facilitate requirements goal tracing and the matching of requirements scenarios to event-trace scenarios during testing, in which the actual and expected behavior of scenarios sharing the same goal are compared.

Thus, an event-trace scenario, collected during a *test run*, can be compared to a requirements scenario sharing the same goal, to determine if actual system

behavior matches expected system behavior. If the match succeeds, the test is said to *pass*; otherwise the test is said to *fail*.

To compare actual and expected behavior and determine whether a scenario-based test passes or fails, it is necessary to use either *oracles* or pre-stored expected event data and result data. Oracles provide an independent computable method to determine the correct expected outcomes in a requirements scenario, under the input conditions given in a corresponding event-trace scenario. This enables one to compare the actual outcomes in the event-trace scenario with the correct expected outcomes in the requirements scenario to decide whether a test passes or fails. When comparing the fine-grained details in an event-trace scenario with the corresponding details in a requirements scenario, oracles can be used to express and measure pre- and post-conditions surrounding individual state-transition operations in a plan or a program.

Scenarios can exist at several levels of abstraction. Abstract scenarios correspond to path traces through goal automata. Concrete scenarios correspond to traces through function and procedure RTDs that contain only implemented programming language operations (and that no longer contain any unrefined sub-goals, even though they may contain goal annotations derived during stepwise refinement).

In performing integration testing using higher-level scenarios that express examples of how system components interact, we may choose not to exercise the test oracles that test the correctness of trusted components that have already been verified or unit tested, or of built-in operators (such as hardware multiplication or mathematical library subroutines), even though it is possible though improbable that such trusted components have imperfections. (Even though such trusted components occasionally fail to live up to the trust we place in them, we need to avoid always retesting lower-level units completely in the context of integration testing, for reasons of cost effectiveness.)

In fact, when we choose not to test the behavior of the unit components during the integration test of a higher-level plan, we are cleanly partitioning the unit testing of the components from the integration testing of the plans that use such units. Thus scenario-based tests can be simplified by such partitioning for purposes of test efficiency.

3 Two Brief Examples

Our two brief examples involve an ATM (Automated Teller Machine) used by a bank, and computing the *least common multiple* of two positive integers. The first example allows us to express high-level goals and introduce our approach in that context. The second

example is simple enough to allow us to show greater detail of our approach.

Example 1: ATM. Fig. 1 shows part of a bank's requirements goal graph that refines top-level stakeholder goals into goals affecting ATM requirements.

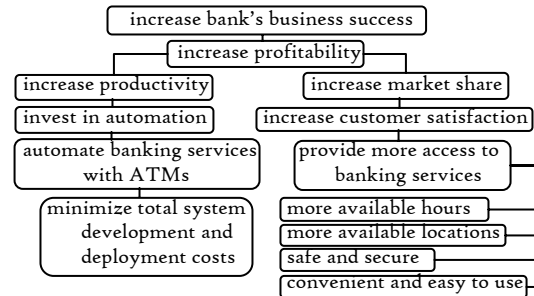


Figure 1. Part of a Bank's Goal Graph

Figure 2 shows an ATM goal automaton that satisfies the ATM leaf-goals in the complete requirements graph (not shown here). Figure 3 gives a goal-annotated ATM event-trace scenario that results from using an implemented ATM to withdraw some cash.

In fact, other sub-goal automata (not shown here) correspond to the sub-goals that label the state transitions in Fig. 2 to authenticate users and provide each of the ATM services. For instance, the sub-goal automaton for withdrawing cash dispenses cash provided the amount entered by the user is within limits prescribed by the bank's business rules (see goal 2.2.3 in Fig. 3).

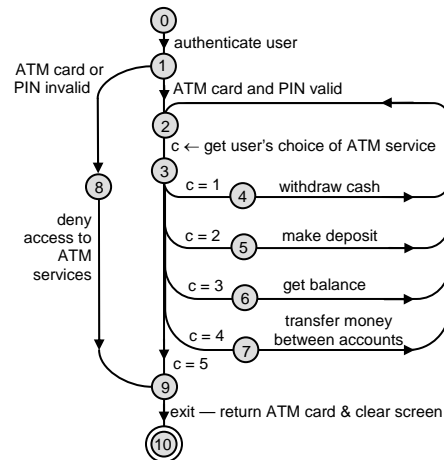


Figure 2. ATM Goal Automaton

From ATM goal automata such as these, we can use path tracing to enumerate goals in ATM requirements scenarios, and it is straightforward to compare the actual ATM session behavior recorded in the event-trace scenario in Figure 3 with the corresponding requirements scenarios obtained by such goal tracing.

A yet more powerful way to express such requirements scenarios is to use ScenarioML, a scenario description language devised by author Thomas Alspaugh. ScenarioML can actually describe whole families of scenarios as scenario schemata, and can even express interesting temporal relations between event intervals using Allen’s interval algebra [Als05].

Top-Level Goal: Conduct an ATM User Session

(note1: numbered items in outline below are ATM requirements goals)
 (note2: "*" indicates observed event collected in event trace scenario)

1. Authenticate User
 - 1.1 Get user to insert ATM card
 - * ATM Card inserted
 - 1.2 Check ATM Card is valid
 - * ATM card determined to be valid
 - 1.3 Get user to enter PIN
 - * user enters PIN and presses "Enter" key
 - 1.4 Check that entered PIN is valid
 - * entered PIN matches expected PIN for ATM card
 - * Authentication Successful
2. Grant User Access to ATM Services
 - 2.1 Get user to choose a transaction type (1: withdraw cash, 2: make deposit, 3: get balance, 4: transfer money between accounts, or 5: terminate session)
 - * user chooses (1) to withdraw cash
 - 2.2 Allow User to Withdraw Cash
 - 2.2.1 Get user to choose account to withdraw from (1: checking, 2: savings)
 - * user chooses (1) to withdraw from checking account
 - 2.2.2 Get user to enter amount, A, to withdraw
 - * user enters A = \$180
 - 2.2.3 Check that A is within limits (1: enough in user's account, 2: within daily withdrawal limit, 3: enough cash in machine, 4: even multiple of twenty dollars)
 - * user's requested amount is within limits (1-4)
 - 2.2.4 Dispense requested cash
 - * \$180 in cash dispensed
 - 2.2.5 Deduct dispensed amount A from: (1: cash available in machine, 2: user's account, 3: user's remaining daily withdrawal limit)
 - * three deduction transactions (1-3) accomplished
 - 2.2.6 Ask if user wants a printed receipt (1: yes, 2:no)
 - * user does not request printed receipt (2)
 - * Cash withdrawal transaction successful
 - 2.3 Get user to choose a transaction type (1: withdraw cash, 2: make deposit, 3: get balance, 4: transfer money between accounts, or 5: terminate session)
 - * user chooses (5) to terminate session
3. Terminate ATM User Session
 - 3.1 Return user's ATM card
 - 3.1.1 Prompt user to expect ATM card ejection
 - * ATM card ejected half-way
 - 3.1.2 Prompt user to withdraw card completely
 - * sensor indicates ATM card completely withdrawn
 - 3.2 Clear user account information from screen
 - * screen cleared
 - 3.3 Enter ready state to begin new ATM session
 - * ready state entered
 - * ATM user session ended successfully

Figure 3. Goal-annotated ATM Event-Trace Scenario

Example 2: lcm. Our second example involves finding $lcm(a, b)$, the *least common multiple* of two positive integers a and b . In books on elementary number theory [Eyn87] we discover that $lcm(a, b)$ can be computed using the formula: $lcm(a, b) * gcd(a, b) = a * b$, where $gcd(a, b)$ is the greatest common divisor of a and b . We can find $gcd(a, b)$ using Euclid’s algorithm.

Thus, our top-level plan automaton for $lcm(a, b)$ corresponds to a program plan using sub-goals such as

```
<< find the lcm of a and b >>
lcm(a, b) {
  << find the product p of a and b >>
  << find g, the gcd of a and b >>
  << return p / g as the lcm(a, b) >>
}
```

where the goals g in this plan are enclosed in double angle brackets $\ll g \gg$.

The corresponding top-level goal automaton is given in Figure 4.

find lcm(a, b) where a and b are positive integers

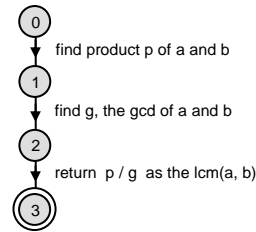


Figure 4. Goal Automaton for lcm(a,b)

Using stepwise refinement, we can refine this goal automaton into a goal-annotated implementation:

```
<< find the lcm of a and b >>
int lcm(int a, int b) {
  int p,g;
  << find the product p of a and b >>
  p = a * b;
  << find g the gcd of a and b >>
  g = gcd(a,b);
  << return p / g as the lcm(a,b) >>
  return p/g;
}
```

A goal automaton that gives the plan for Euclid’s method for finding $gcd(m, n)$ is given in Figure 5.

find gcd(m, n) for positive integers m and n

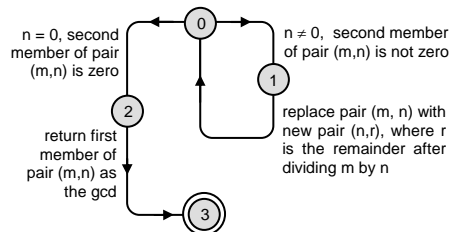


Figure 5. Goal Automaton for gcd(a,b)

This goal automaton can be refined stepwise into a goal-annotated implementation for $gcd(a, b)$:

```

<< find gcd(m,n) for positive integers m and n>>
int gcd(int m, int n) {
  int r;
  << while second member of pair (m,n) is not zero >>
  while ( n != 0 ) {
    << replace the pair (m,n) with the pair (n, m mod n) >>
    r = m % n;
    m = n;
    n = r;
  }
  << return first member of pair (m, n) as the gcd >>
  return m;
}

```

Executing $lcm(6, 4)$, leads to the goal-annotated event-trace scenario shown in Figure 6.

```

Top-level goal: find lcm(6, 4)
(note1: numbered items in outline below are goals)
(note2: "*" indicates observed event collected in event trace scenario)
1. find lcm of 6 and 4
  1.1 find the product p of 6 and 4
    * p = 24
  1.2 find g, the gcd of 6 and 4
    1.2.1 second member of pair (6, 4) is not zero
      * 4 ≠ 0
    1.2.2 replace the pair (m, n) with (n, m mod n)
      * (m, n) replaced by (4, 2)
    1.2.3 second member of pair (4, 2) is not zero
      * 2 ≠ 0
    1.2.4 replace the pair (m, n) with (n, m mod n)
      * (m, n) replaced by (2, 0)
    1.2.5 second member of pair (2, 0) is zero
      * 0 = 0
    1.2.6 return first member of pair (2, 0) as the gcd
      * 2 returned as value of gcd(6, 4)
    1.2.7 assign value 2 = gcd(6, 4) to be value of g
      * g = 2
  1.3 return p / g as the lcm of 6 and 4
    * return 24 / 2
* 12 returned as the value of lcm(6, 4)

```

Figure 6. Event-Trace Scenario for lcm(6,4)

We could use the event-trace scenario in Fig. 6 as a *test case* by comparing it with a requirements scenario consisting of goals extracted from the plan in the goal automaton for $lcm(a, b)$ given in Fig. 4.

```

<< find the lcm of positive integers a and b >>
1. << find the product p of a and b >>
2. << find g, the gcd of a and b >>
3. << return p/g as the lcm(a,b)>>

```

To use the above plan as a requirements scenario in such a test, we need to decide whether we will check the attainment of each of the three sub-goals by using either oracles or pre-stored expected results, or whether we will choose to trust the components that implement these sub-goals. We may choose to trust them because, say, we believe: (i) that the product p of

a and b is reliably computed by the hardware, (ii) that the $gcd(a, b)$ has been both verified and unit tested for correctness, and (iii) that we believe a theorem in a number theory book [Eyn87] that claims $lcm(a, b) = a*b / gcd(a, b)$. In this case, we may choose to ignore the nested trace of details in Fig. 6 that compute the $gcd(6,4)$, and to use an oracle only to test whether the final result $lcm(6,4) = 12$ is correct. For example, an oracle to test whether or not $m = lcm(a, b)$ could first test whether m is a common multiple of a and b (true iff $a | m \wedge b | m$) and could then test multiples of a of increasing size to see if any such multiple is both divisible by b and smaller than m , stopping when we reach a multiple of a equal to m .

The point is that when doing integration testing of a plan that uses already tested or verified components, we can choose to trust but not test the components, or we can choose (selectively) to test the operation of the components in the context of the integration test.

4 Plans for Future Work

We believe, but have not yet confirmed, that our approach to purpose-driven scenario-based testing can result in more efficient testing than traditional testing methods, which are based on exploring control or data flow paths. To confirm our belief and demonstrate how effective our approach is, we must gather quantitative data from experiments comparing our purpose-driven, scenario-based testing approach to traditional testing methods. Of particular interest is whether our approach can scale up to large system testing without encountering the intractability barriers characteristic of the traditional methods. We plan, therefore, to identify suitable medium and large-scale systems to test providing case studies comparing our approach to traditional methods.

5 References

- [Als05] T. A. Alspaugh, ScenarioML: Making Temporally Expressive Scenarios. Submitted to *13th IEEE Joint International Conference on Requirements Engineering (RE'05)*. Feb. 2005.
- [CPRZ89] L. Clarke, A. Podgurski, D. J. Richardson, S. Zeil, A formal evaluation of data flow path selection criteria, *IEEE Trans. Softw. Engr.* 15(11), p.1318-1332, Nov. 1989.
- [Eyn87] Charles Vanden Eynden. *Elementary Number Theory*, McGraw-Hill, Inc., New York, 1987, p. 15.
- [RC85] D. J. Richardson and L. Clarke, Testing techniques based on symbolic evaluation, in *Software: Requirements, Specifications, and Testing*, T. Anderson, ed., p.93-110, Blackwell Scientific Publications, June 1985.
- [Woo70] W. A. Woods, Transition Network Grammars for Natural Language Analysis, *Comm. ACM*, 13(10), Oct. 1970.