ISR Institute for Software Research
University of California, Irvine

An Automated Approach for Goal-driven, Specification-based Testing

Kristina Winbladh
University of California, Irvine
awinblad@ics.uci.edu

Debra R. Richardson
University of California, Irvine
djr@ics.uci.edu

Thomas A. Alspaugh
University of California, Irvine
alspaugh@ics.uci.edu

Hadar Ziv
University of California, Irvine
ziv@ics.uci.edu

May 2006

ISR Technical Report # UCI-ISR-06-8

Institute for Software Research
ICS2 110
University of California, Irvine
Irvine, CA 92697-3455
www.isr.uci.edu

An Automated Approach for Goal-driven, Specification-based Testing

Kristina Winbladh, Thomas A. Alspaugh,
Hadar Ziv, and Debra J. Richardson
Institute for Software Research
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
{awinblad, alspaugh, ziv, djr}@ics.uci.edu

ISR Technical Report #UCI-ISR-06-8
May 2006

Abstract: This paper presents a specification-based approach and implementation architecture that addresses several known challenges including false positives and domain knowledge errors. Our approach begins with a system goal graph and functional goal plans. Source code is annotated with goals from plans the program is attempting to achieve; code is then precompiled to emit annotations at run time. Plans are automatically translated into a rule-based recognizer. An oracle is produced from the pre- and post-conditions associated with the plan's goals. When the program is executed, goals and events are emitted and automatically tested against plans and expected results. This allows more efficient testing, including better recognition of false positives - correct results not matching plans - and domain knowledge errors - incorrect results from following intended plans. The concept is demonstrated for a small example and a larger publicly available case study in which we found a mismatch between stated requirements and actual program behavior.

An Automated Approach for Goal-driven, Specification-based Testing

Kristina Winbladh, Thomas A. Alspaugh,
Hadar Ziv, and Debra J. Richardson
Institute for Software Research
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
{awinblad, alspaugh, ziv, djr}@ics.uci.edu

ISR Technical Report #UCI-ISR-06-8
May 2006

Abstract

This paper presents a specification-based approach and implementation architecture that addresses several known challenges including false positives and domain knowledge errors. Our approach begins with a system goal graph and functional goal plans. Source code is annotated with goals from plans the program is attempting to achieve; code is then precompiled to emit annotations at run time. Plans are automatically translated into a rule-based recognizer. An oracle is produced from the pre- and post-conditions associated with the plan's goals. When the program is executed, goals and events are emitted and automatically tested against plans and expected results. This allows more efficient testing, including better recognition of false positives - correct results not matching plans - and domain knowledge errors - incorrect results from following intended plans. The concept is demonstrated for a small example and a larger publicly available case study in which we found a mismatch between stated requirements and actual program behavior.

1. Introduction

Effective software testing is essential for producing dependable software systems. Specification-based testing is a powerful testing technique whose vital purpose is to confirm the extent to which a system under development meets its specifications and requirements, and identify the ways and reasons it does not. Specification-based testing supports the efficient use of project resources by focusing on the most important behavior, i.e., that which the stakeholders

specified. This paper presents a specification-based testing approach and tool support that focus on finding mismatches between actual and expected system behavior. These mismatches address several known challenges in testing including false positives and domain knowledge errors.

We use plans and goals to further increase the efficiency of specification-based testing. Our approach incorporates oracles to verify satisfaction of intermediate and lower-level system goals and to match the satisfaction of these goals against the plans the system is intended to follow.

In a previous paper [26], we demonstrated this concept by implementing it for a specific example, a Tic-Tac-Toe program (Section 3 provides a brief summary of this work). This paper describes a more generalized and automated prototype for the approach. We start with a goal graph for a system and the plans associated with its functional goals, and annotate the source code with the goals from the plans that the program is attempting to achieve. The goal-annotated version of the implementation is precompiled into source code that emits those annotations at the appropriate time. The plans are automatically translated from GoalML, a markup language for expressing goals created by the first author, into a JESS (Java Expert System Shell [3]) rule-based recognizer. We manually produce an oracle from the pre-and post-conditions associated with the goals in the plan. When the program is run with the plan recognizer and the oracle, the goals and events emitted from the program are automatically tested against the plans and expected results. This allows us to test the actual system behavior against expected system behavior as expressed in the plans.

There are three major contributions of this work:

- We can test whether an implementation follows an appropriate plan to get the correct results. There are several advantages to this approach. Because it is usually impossible to do exhaustive testing or even enough testing, efficient testing should focus on whether the system achieves its most important goals. A second advantage is that testing against plans and goals helps detect false positives. A *false positive* occurs when the program is using an incorrect process, but happens to produce a correct result in a specific case. We verify that the system not only produces correct results but also does so by following a plan that can be expected to produce correct results in all similar cases. A final advantage is that we can more easily detect domain knowledge errors. When the system follows the intended plan but achieves incorrect results, this indicates a likely domain knowledge error.

- We offer a more generalized approach and automated support for this activity than described in our previous work [26], which demonstrated our approach with a specific prototype for a Tic-Tac-Toe program. Our approach has since been generalized and automated in a number of ways, and tested against a larger and publicly available example.

- We apply our testing approach on a higher level of abstraction, by inferring satisfaction of high-level goals from satisfaction of low-level goals, made

possible by the relationships in the goal refinement graph.

Similar to Dardenne and van Lamsweerde, and Myloupolous et al. [12, 17], we define a *goal* to be the purpose toward which an effort is directed. High-level goals can be traced down through a goal-refinement graph to derive functional goals, which can be realized as code components. The refinements can include OR-refinement in which satisfaction of any subgoal is sufficient to satisfy its parent, and AND-refinement in which satisfaction of all subgoals is necessary to satisy their parent. Refinements of lower-level functional goals can also include plans. A *plan* is an abstract description of how to satisfy some goal by satisfying subgoals in some order. This description may contain sequences, iterations, and alternations of subgoals, expressing the plan of action used to achieve the higher-level goal. A *scenario* is a sequence of events that may be used to describe how a lowest-level goal is achieved. Figure 1 illustrates a representation of a goal refinement graph exhibiting OR-refinement, AND-refinement, and a plan as a detailed refinement.

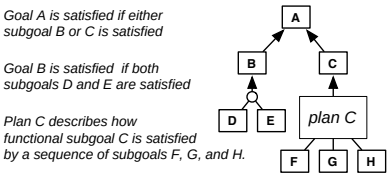


Figure 1: Goal refinement graph

The remainder of this paper is organized as follows: Section 2 provides an overview of our approach. In Section 3 we summarize our previous concept demonstration and in Section 4 we apply our approach and tools on a larger example, an ATM (automated teller machine) simulation. Section 5 provides a more detailed description of our approach and implementations. In Section 6 we summarize related work in this area. Finally in Section 7 we present lessons learned and discussion of future work.

2. Overview of the Approach

As in all testing, we compare actual system behavior to expected system behavior. Figure 2 shows an overview of our testing approach. We determine expected system behavior from the higher-level goals, the lower-level goals they are refined into, and the relationships between those goals. At the upper levels of the goal refinement graph, these relationships are AND/OR refinements. At the lower levels, functional goals are also refined using plans for how each goal is satisfied by a sequence of lowest-level goals. At the lowest goal level, we use oracles to determine whether each goal has been satisfied. The oracles are derived from the pre- and postconditions for each goal, and from the scenarios for the lowest-level goals. At higher levels, we use the plans, or the AND/OR

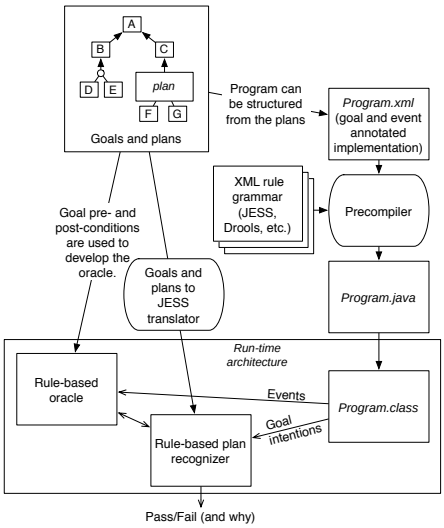


Figure 2: Process diagram

relationships where there are not plans, to determine whether a higher-level goal has been satisfied through the satisfaction of a group of its subgoals; we can also use oracles here as an additional confirmation.

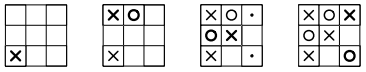
An essential feature of our approach is annotating the system's code with goals from the plans and events from the scenarios, so that during program execution these goals and events will be emitted. The goals and events show how the program attempts to satisfy the plans that achieve its higher-level goals. We precompile the code, transforming the annotations into additional code that will emit statements of intention to satisfy each goal and events the oracle uses to verify that each goal was in fact satisfied, at the appropriate points in the execution. We translate the plans into a plan recognizer that runs with the program and matches its emitted goals with the goals expected by the plans. It confirms whether an appropriate combination and sequence of goals was emitted. Satisfaction of goals is verified by an oracle, based on the events and the goals' pre- and postconditions. The oracle is (at present) combined with the plan recognizer. When we run the program with the plan recognizer and oracle, they find mismatches between actual and expected behavior, not just the specific results but also the process by which they are obtained.

We believe that goals, goal graphs, plans, and scenarios are an especially advantageous foundation on which to do this. A single form, goals, can be used at all levels of abstraction, providing a smooth transition from upper to lower levels that can be validated by stakeholders, both in terms of specific goals and in terms of the relations between them. Goal graphs with plans allow us to infer satisfaction from the lowest levels up to the highest. Finally, plans and scenarios support a substantial degree of automation.

3. Concept Demonstration with Tic-Tac-Toe

In many cases, raw event traces (without goal annotations) are not sufficient to directly detect false positive results. In this section we summarize our earlier proof-of-concept for our approach, in which we studied tests for a Tic-Tac-Toe program [26].

Figure 3 shows a raw event trace from a Tic-Tac-Toe program being tested to verify the skill level at which the machine is playing against a human player. At the 'expert' level of play, the machine is required to identify fatal second moves and exploit these with a forcing move that leads inevitably to a fork (a situation with two winning moves — see third board in Figure 3) and a win. The 'intermediate' level of play requires the machine to identify immediate opportunities to make a winning move or create a fork that leads to a win on the next move, but not to recognize such opportunities one move ahead. Both levels of play require that the program play with variety, randomly selecting among possible moves that satisfy its requirements.



Machine plays first with X
* Machine moved with X in lower left corner (3,1)
* Player moved with O on top side (1,2)
* Machine moved with X in top left corner (1,1)
* Player moved with O on the left side (2,1)
* Machine moved with X in the center (2,2)
* Player moved with O in the lower right corner (3,3)
* Machine moved with X in the top right corner (1,3)

Result: X won the game

Figure 3: Raw event trace from a game

Does the raw event trace in Figure 3 show whether the program is playing at the expert or intermediate level? By itself, it does not distinguish the two possibilities. One approach to testing this would be to perform a large number of test cases, verify that each is consistent with the desired behavior, and then make a statistical argument for the probability that the program is behaving as desired. Our approach is to produce a goal-annnotated event trace that provides more direct evidence.

Figure 4 shows a goal-annotated event trace for 'expert' level play. Comparison with the raw event trace (Figure 3) shows that the two are entirely consistent. The program recognized the player's fatal second move and made a forcing move that led to a fork and then a win, consistent with the requirements for this level of play.

Figure 5 shows a goal-annotated event trace for 'intermediate' level play. Comparison with the raw event trace (Figure 3) shows that these two are also entirely consistent. The program fortuitously chose a random move that happened to lead to a winning fork on the machine's next move. This is consistent with the requirements for this level of play.

This example illustrates how raw event traces may not distinguish false positive results that our approach does.

A certain amount of domain knowledge about the program being tested is necessary in order to perform meaningful tests. The Tic-Tac-Toe program con-

```
Machine plays first with X at the expert level
Goal: Make a random first move
   * Machine moved with X in lower left corner (3,1)
   * Player moved with O on top side (1,2)
Goal: Reply to opponent's fatal 2nd move by making a forcing
        move that will lead to a winning fork
   * Machine moved with X in top left corner (1,1)
   * Player moved with O on left side (2,1)
Goal: Create a winning fork
   * Machine moved with X in the center (2,2)
   * Player moved with O in the lower right corner (3,3)
Goal: Make a winning move
   * Machine moved with X in the top right corner (1,3)
Result: X won the game
```

Figure 4: Goal-annotated expert trace

```
Machine plays first with X at the intermediate level
Goal: Make a random first move
   * Machine moved with X in lower left corner (3,1)
   * Player moved with O on top side (1,2)
Goal: Make a random move
   * Machine moved with X in top left corner (1,1)
   * Player moved with O on left side (2,1)
Goal: Create a fork
   * Machine moved with X in the center (2,2)
   * Player moved with O in the lower right corner (3,3)
Goal: Make a winning move
    * Machine moved with X in the top right corner (1,3)
Result: X won the game
```

Figure 5: Goal-annotated intermediate trace

tains several subtle and interesting domain concepts. For example, the expert-level strategy makes use of knowledge about which second moves are safe and which are fatal.

Our approach identifies domain knowledge errors by detecting mismatches between expected and actual results even though the intended plan is being followed. For example, it is quite possible that the program's characterization of safe and fatal second moves is incorrect. If the expert-level strategy uses incorrect definitions of safe and fatal second moves when choosing its moves, the plan recognizer will detect event-traces of games in which the program believed it was making a safe second move, emitted a goal with that intention, and then an event (a move) that the oracle identifed as fatal. In our Tic-Tac-Toe study, we deliberately introduced such errors (confusing safe and fatal second moves) into the data tables on which expert-level play relies in our Tic-Tac-Toe program. Our tests detected these seeded domain knowledge errors by finding games in which the events did not satisfy the current goal of the plan, i.e., the goal to make a forcing move.

4. Prototype Application with ATM

In this section we describe an exploratory case study that applied our approach to an ATM simulation system. This existing, publicly available software system was developed in academia with the purpose of illustrating a complete object oriented development process [8]. It provides a wider range of goal levels than

the Tic-Tac-Toe example, as well as particularly interesting goals, both functional and non-functional. We illustrate the application of our approach to part of the ATM simulation system.

This case study demonstrates the application of most of our approach (goals, plans, goal refinement graph, and recognition of plans) to a larger, more complex subject system not developed by us. In contrast to the Tic-Tac-Toe study, we did not event-annotate the ATM source code or construct an oracle to verify those events; for well-understood systems such as ATMs, an oracle is of less interest and value because the concepts it verifies are more likely to be intuitively apparent to a user.

The case study also illustrates extensions of our approach and tool support since the concept demonstration [26]. We can now test satisfaction of higher-level goals as well as goals at the lowest levels, and the generalization of our precompiler to produce output for other rule-based languages (e.g. Drools [1]). In addition, our process has been further automated by autogeneration of plan recognizers from GoalML plans.

4.1. Introduction to the ATM Simulation

The ATM program simulates the functionality of a real ATM. Since there is no physical ATM or bank, an ATM card is simulated by entering the card number, the bank database is simulated by a module of the simulation, and receipts, dispensed cash, and deposits are simulated by images on the computer screen. An authorized customer is able to perform withdrawals, deposits, transfers, and balance inquiries.

The ATM communicates each transaction request to the bank component for authorization. If the bank component determines that the customer's PIN is invalid, the customer is required to re-enter the PIN correctly before a transaction can be completed. The system requirements state that if the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine.

The ATM simulation comes with the following artifacts: requirements, use cases, initial functional tests, analysis classes, CRC cards, a class diagram, state charts, interaction diagrams, detailed design, a package diagram, code, and maintenance ideas.

4.2. Goal Refinement

Since the ATM simulation project did not provide goals, we reconstructed a goal refinement graph from the requirements and standard banking business rules and goals. Figure 6 illustrates part of the goal graph for a bank.

The graph includes high level goals such as **Prevent unauthorized access to accounts**, low level functional goals such as **Withdraw** that map to a single component, and some goals that can be identified as concerns, such as **ATM authentication**. Concerns are goals that impact many components of the system; these are generally related to non-functional requirements. For example,

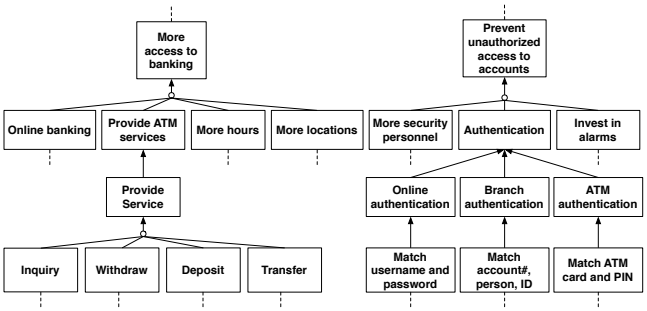


Figure 6: Portion of goal refinement graph

the functional goal **Provide Service** is refined into four subgoals, and maps to the class `Transaction` and its four subclasses (`Withdraw`, `Deposit`, `Transfer`, and `Inquiry`) respectively. Concerns such as **ATM Authentication** map onto many components, in this case all components that use authentication such as `Withdraw`, `Transfer`, `Deposit`, and `Inquiry`. Some parts of the system thus satisfy several goals.

4.3. ATM Session Plan

Figure 7 shows the GoalML plan for the functional goal **Provide ATM services**, again reconstructed from the requirements and standard banking business rules and goals.

```
<plan name="ATM Session" goal="Provide ATM services">
  <goal name="Read card"/>
  <goal name="Get PIN"/>
  <iteration goal="Try to provide services">
    <goal name="Read transaction type"/>
    <goal name="Send transaction request to bank"/>
    <alternation>
      <sequence goal="Do authenticated transaction">
        <alternation goal="Have valid PIN">
          <sequence/><!-- Do nothing if already have valid PIN -->
          <sequence>
            <goal name="Get PIN"/>
          </sequence>
          <sequence>
            <goal name="Get PIN"/>
            <goal name="Get PIN"/>
          </sequence>
        </alternation>
        <goal name="Do transaction"/>
        <goal name="Print receipt"/>
        <goal name="Ask if the customer wants another transaction"/>
        <alternation>
          <sequence/>
          <goal name="Eject ATM card"/>
        </alternation>
      </sequence>
      <sequence goal="Retain unautheticated card">
        <goal name="Get PIN"/>
        <goal name="Get PIN"/>
        <goal name="Permanently retain ATM card"/>
      </sequence>
    </alternation>
  </iteration>
  <goal name="End session"/>
</plan>
```

Figure 7: ATM session plan

The <goal> elements represent the lowest-level goals, and their sequence is described by the <iteration>, <sequence>, and <alternation> elements. For example, the alternation of the sequences **Do authenticated transaction** and **Retain unauthenticated card** and their internal alternations in Figure 7 illustrates valid interactions with the ATM.

4.4. Goal Annotating the Code

We used the goals from the **ATM session** plan to manually goal annotate the implementation code for three of the classes in the ATM system (`Transaction.java`, `Session.java`, and `ATMMain.java`). Figure 8 illustrates a sample of a GoalML-annotated piece of code.

```
public Status performInvalidPINExtension()
    throws CustomerConsole.Cancelled,CardRetained {
  Status status = null;
  for (int i = 0; i < 3; i ++) {
    </code> <goal name="Get PIN"/> <code>
    pin = atm.getCustomerConsole().readPIN(
      "PIN was incorrect\nPlease re-enter your PIN\n" +
      "Then press ENTER");
    atm.getCustomerConsole().display("");
    message.setPIN(pin);
    status = atm.getNetworkToBank().sendMessage(message, balances);
    if (! status.isInvalidPIN()) {
      session.setPIN(pin);
      return status;
    }
  }
  </code> <goal name="Permanently retain ATM card"/> <code>
  atm.getCardReader().retainCard();
  atm.getCustomerConsole().display(
    "Your card has been retained\nPlease contact the bank.");
  try {  Thread.sleep(5000);  }
  catch(InterruptedException e) { }
  </code> <goal name="End session"/> <code>
  atm.getCustomerConsole().display("");
  throw new CardRetained();
}
```

Figure 8: Goal-annotated code sample

4.5. Precompilation

Our precompiler translated the GoalML-annotated version of the program into a program that emits the goals it is intending to achieve, at the appropriate time during execution. The GoalML annotations were replaced by program code to emit the corresponding goals. In this case, the added program statements were JESS assertions in Java where each assertion adds a goal as a fact to the working memory of the plan recognizer (see Figure 9) .

4.6. Plan Recognizer

We created JESS rules that together recognize whether the **ATM session** plan is followed. The plan recognizer always knows which goals are expected next, and moves forward through the plan when one of these occurs. Figure 10 illustrates one of the plan recognizer's rules. This rule recognizes when **Send**

```
public Status performInvalidPINExtension()
    throws CustomerConsole.Cancelled, CardRetained {
  Status status = null;
  for (int i = 0; i < 3; i ++) {
try{
engine.executeCommand("(assert (goal (name \"Get PIN\")
 (status received)))", context);
engine.executeCommand("(run)");
}catch(JessException je){
  System.out.println(je);}
    pin = atm.getCustomerConsole().readPIN(
      "PIN was incorrect\nPlease re-enter your PIN\n" +
      "Then press ENTER");
    atm.getCustomerConsole().display("");
    message.setPIN(pin);
    status = atm.getNetworkToBank().sendMessage
                                        (message, balances);
    if (! status.isInvalidPIN()) {
      session.setPIN(pin);
      return status;
    }
  }
try{
engine.executeCommand("(assert (goal (name \"Permanently
 retain ATM card\") (status received)))", context);
engine.executeCommand("(run)");
}catch(JessException je){
  System.out.println(je);}
  atm.getCardReader().retainCard();
  atm.getCustomerConsole().display(
      "Your card has been retained\nPlease contact the bank.");
  try {  Thread.sleep(5000);  }
  catch(InterruptedException e) { }
try{
engine.executeCommand("(assert (goal (name \"End session\")
 (status received)))", context);
engine.executeCommand("(run)");
} catch(JessException je) {
  System.out.println(je);}
  atm.getCustomerConsole().display("");
  throw new CardRetained();
}
```

Figure 9: Code with assertion

transaction request to bank has been received as expected and is followed either by **Get PIN** or **Do transaction**.

The recognizer also contains rules that infer satisfaction of higher-level goals as a result of satisfaction of lower-level goals, by direct AND-OR relations or plans. For example, when the ATM simulation is run and the **ATM session** plan passes according to the recognizer, the recognizer determines and marks the goal **Provide ATM service** as satisfied. Once that functional goal is satisfied another rule determines that the goal **Provide ATM services** is also satisfied. This relation between the plan and the two higher-level goals is inferred by their refinement relations in the goal graph.

```
(defrule goal-sequence-2
  (goal (name "Send transaction request to bank") (status matched))
  =>
  (assert (goal (name "Get PIN") (status expected) (condition ignore)))
  (assert (goal (name "Do transaction")(status expected)
                (condition ignore))))
```

Figure 10: Sample plan rule

4.7. Executing Program and Recognizer

When the program generated by the precompiler executes, it creates an instance of the JESS engine containing the rules of the plan (i.e., the recognizer). As the program executes, goals are asserted into the rule-engine. The first goal asserted by the ATM simulation was **Read card**; this goal was expected by the plan recognizer. The goals **Get PIN**, **Read transaction type**, and **Send transaction request to bank** were all received in that sequence and matched against the plan. We ran a normal session using a valid ATM card and PIN, and the result was a sequence of goals accepted by the plan recognizer. Once the plan was matched, the higher-level goal **Provide service** was also matched which in turn inferred the satisfaction of the goal **Provide services**.

We then ran the simulation using a valid ATM card but an invalid PIN number. This gave a prefix of the correct goal sequence, but after we entered an incorrect PIN for the third time the plan recognizer detected a mismatch indicating that the program was not behaving according to the **ATM session** plan and thus not adhering to the requirements.

Figure 11 shows the goal trace with the unexpected fourth (total) **Get PIN** not expected by the plan; the expected goal at that point was **Permanently retain ATM card** (Figure 7),

It turns out that you must enter an incorrect PIN four times before the ATM card is retained by the simulation, and not three as the requirements stated.

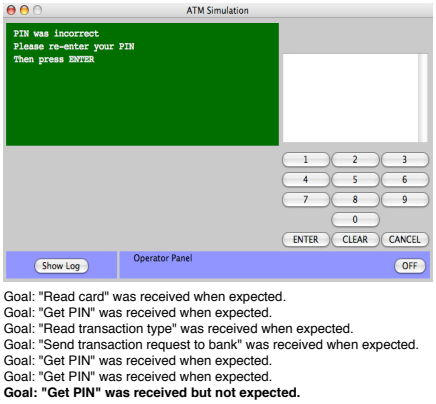


Figure 11: Goal trace with mismatch

4.8. Discussion

Our approach is specifically designed to catch errors that are manifested as mismatches between actual and expected system behavior. The error we found is of this type. We are unaware if this error was intentionally seeded or has been discovered before, but it was gratifying that our approach discovered an error in a published system.

An informal analysis of the error indicates that it can be traced back to problems in the design artifacts, and more specifically to the challenge of moving from one type of software representation to another. This is a likely place to introduce errors which then get propagated through the design and finally manifest themselves in code that does not match the requirements.

We believe that following goals and plans reduces the risk of introducing such errors, that applying the testing approach would have revealed the mismatch (presumably prior to "releasing" the system), and that our approach combined with automated traceability would have increased the likelihood of finding the error back in the use cases and statechart diagram.

First, the handling of PINs was distributed across two primary use cases and an extension use case. A correct understanding could be obtained only by assuming that the first description of entering a PIN in each use case referred to the same single action. Second, the relevant statechart leaves ambigous how the design implements the invalid PIN requirement, which is that an invalid PIN may not be entered more than three times.

5. Detailed Approach

5.1. GoalML Plans and Goals

The goal refinement graph and plans are expressed in GoalML. The fundamental element in GoalML is a `goal` with a name. Lowest-level goals also have `precondition` and `postcondition` elements that explain in terms of the domain how the goal can be satisfied. Plans are expressed by `sequence`s, `alternation`s, `permutation`s, and `iteration`s of goals and each other, in any combination. In goal refinement graphs, OR-refinement of a goal is expressed by an `alternation` of its subgoals, AND-refinement by `permutation`.

5.2. Precompiler

The precompiler translates a goal- and event-annotated code implementation to an executable component that emits goals and events during execution.

The precompiler can translate the goal tags into any user-specified code fragments. Templates for these fragments and supporting code are expressed in external XML files consulted by the precompiler. We currently have template sets for JESS and for Drools. Both JESS and Drools are rule-based languages with Java interfaces. Facts and rules are added to a knowledge base and when some fact in the knowledge base matches the right-hand side of a rule, the left-hand side of the rule executes (manipulating, adding, or deleting objects in the knowledge base). Even though the two languages are similar in their approaches they use completely different syntax. Their similarities and differences were taken into account when designing the XML template format, and when modifying the precompiler to take advantage of the external XML templates.

The templates are not restricted to rule-based output, and could for example simply print the goal names to a stream.

Figure 12 illustrates a JESS assertion and a Drools assertion generated by the precompiler.

Event assertion in Drools:

```
event.player = "machine";
event.row = row ;
event.column = col ;
event.status = "test";
event.type = "nil";
drools.assertObject(event);
```

Event assertion in JESS:

```
try{
 engine.executeCommand("(assert (move (player "
  +" machine"+ ")  (row "+row + ")  (column "+col
  + ")  (status "+" test"+ ")  (type "+" nil"+ ") ))", context);
engine.executeCommand("(run)");
}catch(JessException je){
System.out.println(je);}
```

Figure 12: Precompiler output: Drools and JESS

5.3. Plan Recognizer

The rule-based plan recognizer is automatically generated from the GoalML plans. A generator for JESS plan recognizers is currently implemented. The generator is designed for easy modification to produce output for other rule-based expert system engines.

The plan recognizer identifies higher-level goals as satisfied (if their plans or subgoals are achieved), or not. A higher-level goal defined by a plan, is satisfied if its plan is followed along a path of satisfied goals. A higher-level goal defined by OR- or AND-refinement is satisfied if one or all of its immediate subgoals are satisfied, respectively.

5.4. Oracle

The oracle is used to determine if lowest-level goals are satisfied. We check that the results produced by a system are correct by determining if the events, which individually or in combination embody the results, satisfy the system's goals. We do this by annotating the source code with events and goals. When the compiled system asserts an event, the oracle updates its view of the world accordingly. When the system asserts a goal, the plan recognizer identifies it as expected (matching a current plan) or unexpected. The oracle then identifies lowest-level goals as satisfied (if their postconditions are true in the current state of the world), or not. For each goal, we print that it was received when expected, or not, and (for lowest-level goals) that it was satisfied when received, or not.

For the Tic-Tac-Toe study, we annotated source code and implemented a JESS oracle along somewhat different lines. Here we asserted intended goals before the events that would satisfy them. Each goal was required to be satisfied

by a single event (rather than for example a sequence of events). A goal could be satisfied immediately or later after other events had been received and other goals satisfied. This opened the possibility that a goal could be satisfied later by events not intended for it. In the Tic-Tac-Toe study this possibility was avoided, we determined in retrospect, because each goal was associated with a particular move number. However, we determined that as a general approach it was more effective to assert events before goals, and check lowest-level goal satisfaction immediately.

It is not necessary to use JESS or another rule-based language to implement an oracle. However, we found that doing so gave significant advantages. Running the oracle with the system allowed us to access state and intermediate results while the system was executing, and without danger of impacting the system's state. Although we have not made use of this feature yet, JESS's Java interface allows us to assert entire objects without disturbing their state, which makes it possible to work more directly in the terms of the goals of the system. JESS is an Eclipse [2] plug-in, which allowed us to use it in our usual Java development environment. Finally, we found rules convenient for checking conditions, as the form of the rules could be matched to the form of the conditions.

5.5. Integration

Many of the activities discussed above are part of the pre-processing necessary for our approach to work. Once the executable components (the oracle, plan recognizer, and code with goal- and event-emitters) exist, they run concurrently, and the program is tested against the plan and expected results at its runtime. This provides useful low-level tests of code at a fine granularity, and it also provides meaningful insights about higher-level goals through inferring satisfaction of these from the satisfaction of the plans and low-level functional goals. This inference is possible because of the goal refinement graph and the plans that express the relationship between higher- and lower-level goals.

6. Related Work

When we execute goal-annotated components, they emit event traces of their running behavior containing events, data results, and goal-annotations of interest. Expectation-Driven Event Monitoring (EDEM) [4], Software Tomography [9], residual testing [19, 21], Gamma Technology [20], and The Perpetual Testing project [22], are research projects that also address this problem. However, these projects generally do not rely on goals, and specifically do not use goal annotations in event traces to indicate what subgoals the component was working on during execution.

The use of goal-annotated event traces emitted during code execution is not a completely novel idea, as programming with assertions for discovering program errors has been a topic under investigation for a long time [11, 23].

A challenge with previous techniques is that they do not integrate easily with existing programming environments, an issue we address by using XML-based annotations that are precompiled into a user specified format and by planning to make our environment an Eclipse plugin. Another challenge is that it is not well understood what kinds of assertions are the most effective at detecting software errors. We believe our solution presented in this paper provides an answer to that problem, as our main contribution to programming with assertions is to automatically test software against specifications and requirements.

Coppit and Haddox-Schatz have done some work in the area of specification-based assertions as a means for testing [11]. Their method involves translating formal specifications to program assertions to be inserted in the implementation. During program execution these assertions will be checked for violations and flags raised if such occur. Our method differs in that the specifications are expressed as plans for achieving requirements goals, and are separate from the implementation code. We are testing correctness of the implementation against these plans at the same time as we are testing intermediate and final results against specification-based oracles. Since the plans and oracles are separate from the implementation, we avoid the risk of impacting the program's state at all times. One benefit of our use of assertions is that we can make use of the run-time state of the program, just as the conventional idea of programming with assertions, while at the same time we maintain the separation of specification and implementation. This separation is also beneficial for debugging and understanding the errors, since plan structures are usually more readable than program assertions.

Work on refinement of goals into operationalizable requirements, requirements that can be operationalized into code components, has been carried out for at least a decade, including goals reduced into both functional requirements (e.g., use cases [10]) as well as non-functional ones. Of particular significance is the work on goal refinement by van Lamsweerde *et al.* [12, 14, 24], and Mylopoulos *et al.* [17], specifically the work on mapping goals to requirements scenarios.

We claim that our strategy helps to detect false positives. This was recognized as one of the most fundamental problems with software testing by Goodenough and Gerhart in [13], as the "... weakness of testing lies in concluding that from the successful execution of selected data, a program is correct for all data," and it is well known that many programs that have been tested, validated, and released to the field still contain errors [25]. Young and Taylor described this problem as being an over-optimistic inaccuracy of the test results [27]. We recognize that test data selection is highly relevant in overcoming this weakness, but we also claim that our solution based on testing actual system behavior against expected system behavior reduces over-optimistic results.

7. Lessons Learned and Future Work

Our specification-based approach compares goals and plans against program source code to improve testing efficiency. We evaluated our approach using a small example as a concept demonstration [26] and a study presented here on a larger, publicly-available system developed elsewhere. The examples exhibit several interesting results. First, in our Tic-Tac-Toe study, we were successful in finding both false positives and domain knowledge errors. In the ATM simulation case study, we identified a less-clearly characterized mismatch between actual and required system behavior. Second, we were able to significantly automate our approach, making it substantially quicker and more straightforward to use. Third, we were able to use the results from testing against low-level goals to automatically infer satisfaction of higher-level goals as well. Several of the pre-processing steps, such as the precompiler and generation of the plan recognizer, have been automated and generalized to be easier and faster to use and less dependent on a specific programming language.

In working with and using our approach we have learned that goal models are a particularly useful modeling notation. For example, the error we identified in the ATM simulation system can be traced — by manual analysis — to problems in the use cases and statecharts for that system. We also learned that crosscutting behaviors can be identified during the goal-refinement process as goals that map onto many components.

In a previous paper [26] we discussed the use of scenarios written in ScenarioML [5], a scenario markup language written by the second author, to describe the events that satisfy the pre- and post-conditions of lowest-level goals. These scenarios could be used to put event annotations in the code at appropriate places, so that the events could be used for testing by the oracle. In addition, ScenarioML provides greater expressiveness that GoalML can benefit from, and the automated support under development for ScenarioML complements that which we have provided for GoalML. Our future goal is to make use of scenarios, event annotations, and oracles in the ATM simulation.

We also wish to test against many plans concurrently to provide more efficient testing. We believe that this is possible with some modifications to the plan recognizer.

We believe that higher level goals should correspond to higher level testing activities such as integration and regression testing. This correspondence requires further study, since a higher level goal does not necessarily correspond to a single code module or architectural component. We therefore plan to explore ways in which our approach complements or otherwise relates to other research in this area, in particular architecture-based testing. Architecture-based testing typically means testing that program source code matches a specified architecture. This is often called conformance testing, i.e. checking that an implementation fulfills its specification [16]. Archictecture-based testing requires a systematic approach to code-level conformance-testing based on architecture specifications. Several approaches exist, taking as input an architecture specified using some architectural style such as C2, UML, or CHAM. Often, another for-

mal representation is derived from the specification (using, for example, Labeled Transition Systems (LTS), Finite State Machines (FSM), or Message Sequence Charts (MSC)). The relevant test artifacts (test scenarios, test suites, test plans, or test cases) are then generated from this formal representation. For example, Muccini et al. follow a systematic testing approach from specifications in the C2 architectural style through Abstract Labeled Transition Systems (ALTS) down to code-level execution of the architecture-based tests [7, 16]. This is part of a larger research project in architecture-based regression testing [15].

We therefore plan to extend our work to include not only goals, plans, scenarios and code, but also other development artifacts such as software architectures, other design representations, requirements, and so on. We hypothesize that with additional artifacts and with traceability among those artifacts, more efficient, purposeful testing can be accomplished [6, 18]. Our ultimate goal is to provide forward mapping of requirements to other software artifacts and to provide automated traceability from those artifacts back to requirements.

References

[1] Drools. `http://drools.org/`.

[2] Eclipse. `http://www.eclipse.org`.

[3] JESS (Java Expert System Shell). `http://herzberg.ca.sandia.gov/jess/`.

[4] Expectation-driven event monitoring (EDEM), 2002. `http://www.ics.uci.edu/~dhilbert/edem/`.

[5] T. Alspaugh. Temporally Expressive Scenarios in ScenarioML. Technical Report UCI-ISR-05-06, Institute for Software Research, University of California, Irvine, 2005.

[6] T. Alspaugh, D. Richardson, T. Standish, and H. Ziv. Scenario-driven specification-based testing against goals and requirements. In *REFSQ'05*, pages 187–202, 2005.

[7] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In *SFM*, pages 122–147, 2003.

[8] R. C. Bjork. ATM Simulation, 2002. `http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/`.

[9] J. Bowring, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *PASTE'02*, pages 2–9, 2002.

[10] A. Cockburn. Structuring use cases with goals. *J. of Object-Oriented Programming*, Sept./Oct. 1997.

[11] D. Coppit and J. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *29th IEEE/NASA Softw. Eng. Wkp.*, pages 305–314, 2005.

[12] A. Dardenne and A. van Lamsweerde. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):14–21, 1993.

[13] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[14] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *FSE'02*, pages 119–128, 2002.

[15] H. Muccini, M. Dias, and D. J. Richardson. Software architecture-based regression testing. *to appear in JSS, Special Edition on Architecting Dependable Systems*, 33(4):24–29, 2006.

[16] H. Muccini, M. S. Dias, and D. J. Richardson. Systematic testing of software architectures in the C2 style. In *FASE*, pages 295–309, 2004.

[17] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[18] L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv. Using scenarios to support traceability. In *Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'05)*, Nov. 2005.

[19] L. Naslavsky, R. Silva Filho, C. de Souza, M. Dias, D. Richardson, and D. Redmiles. Distributed expectation-driven residual testing. In *RAMSS'04 (ICSE)*, 2004.

[20] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *ISSTA'02*, pages 65–69, 2002.

[21] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE'99*, pages 277–284, 1999.

[22] D. Richardson. Perpetual testing, 2002. `http://www.ics.uci.edu/~djr/edcs/`.

[23] D. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

[24] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, Dec. 1998.

[25] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[26] K. Winbladh, T. A. Alspaugh, and D. J. Richardson. Specification-based testing using plans and goal annotations. In *ISSTA'06 - In Submission*, 2006. `http://www.ics.uci.edu/~awinblad/publications.html/`.

[27] M. Young and R. Taylor. Rethinking the taxonomy of fault detection techniques. In *ICSE'89*, pages 53–62, 1989.