

An Automated Approach for Specification-based Testing Using Business Goals and Plans

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

Abstract

This paper presents a specification-based testing approach and implementation architecture that addresses several known challenges including false positives and domain knowledge errors. Our approach begins with a system goal graph consisting of high level business goals which are refined to operational goals, and plans that describe how to strategically achieve these goals. Along the goal refinement process business rules can be derived and/or used for constructing plans. Source code is annotated with goals and events and precompiled to emit those 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 oracles. 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.

Most businesses today rely on software in some form or another, it is thus becoming more important for software to integrate with business rules and policies to support a business's goals. In order to build software that adheres to business rules, these need to be described in requirements and other specifications and verified in the final system. *Business rules* and *policies* can be derived from business goals; they describe the operations, definitions, and constraints that apply to an organization in achieving its goals. Business rules are the foundation on which strategies and tactics are built [14]. The rules simply tell an organization what it can do while the strategies and tactics tell it how to do it. Analogously, our specification based testing approach describes the intention of the system in terms of goals and describes strategies for achieving goals either through AND/OR-refinements or plans. We use 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.

There are three major contributions of this work: First we can test whether an implementation follows an appropriate plan to get the correct results. It is usually impossible to do exhaustive testing, so efficient testing should thus focus on whether the system achieves its most important goals. Testing against plans and goals also 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 does so by following a plan that can be expected to produce correct results in all similar cases. We can also detect domain knowledge errors — cases when the system follows an intended plan but achieves incorrect results. Second, we provide automated support for this approach that substantially reduces the extra labor. Third, 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 Mylopoulos *et al.* [12, 18], we define a *goal* to be the purpose toward which an effort is directed. High-level goals are step-wise refined to derive functional requirements goals, which can be realized as code components. The functional goals are further refined by plans, so that each functional goal is the root of a plan. This refinement of high-level goals to detailed plans is similar to the NFR framework [18] and Cockburn’s three leveled use-cases [9] in that the process goes from the requirements level to the design level. A *plan* is an abstract description of how to satisfy some goal by satisfying its subgoals (lowest level goals) in some order. The plans may contain sequences, iterations, and alternations of lowest-level goals. A lowest-level goal is characterised by pre- and postconditions. Each of these conditions represents some state. State transitions from pre- to postcondition can be associated with an abstract operation and identified with a single piece of code which is its implementation. An *event* is an instance of an abstract operation. A lowest-level goal can thus be satisfied by a single event or a sequence of events that alter the system state from the goal’s precondition to its postcondition.

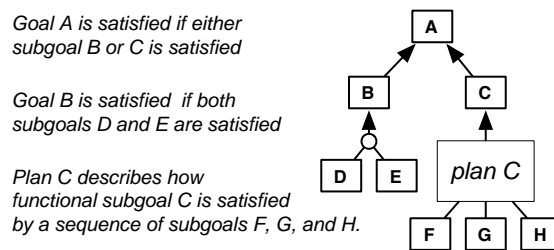


Figure 1. Goal refinement graph

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. In Section 5 we summarize related work in this area. Finally in Section 6 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 as well as events identified during design. At higher levels, we use the plans, or the AND/OR 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, 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.

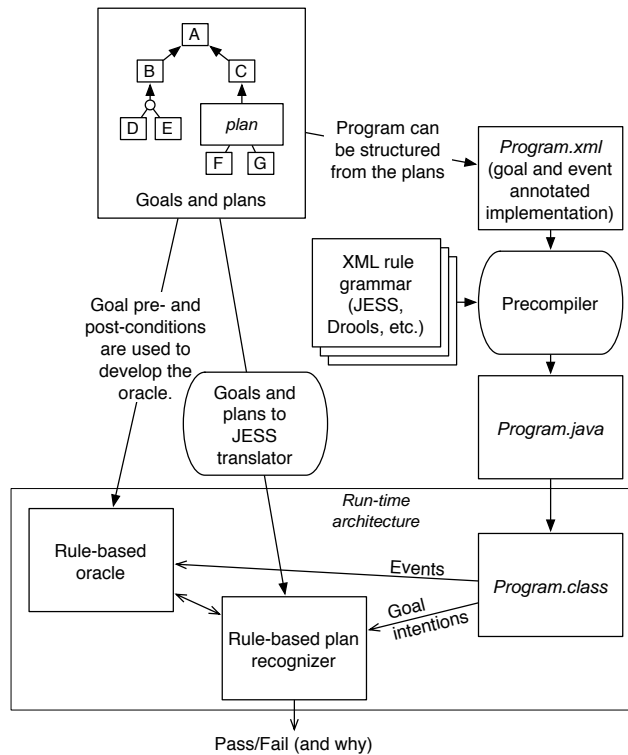


Figure 2. Process diagram

Our hypothesis is that goals, goal graphs and plans are an especially advantageous foundation for our approach. 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. Furthermore, plans 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. For example, consider the 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 shown in Figure 3. At the 'expert' level of play, the machine is required to identify fatal

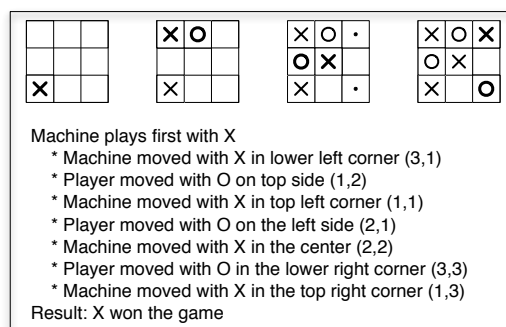


Figure 3. Raw event trace from a game

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. Since a raw event trace only shows events, we cannot distinguish whether our program is playing at the expert or intermediate level, i.e. if a move that led to a fork was a forced or random. 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 use statistics to argue the degree of confidence. Our approach is to produce a goal-annotated event trace that provides more direct evidence.

A goal annotated event trace shows both events and intentions. If the program was playing at the ‘expert’ level, the goal annotated ‘expert’ trace would show that the program recognized the player’s fatal second move and made a forcing move that led to a fork and a win, by revealing its intentions and actions that produced that result (see Figure 4 Left). Comparison with the raw event trace (Figure 3) shows that the events in the goal-annotated event trace are consistent with the raw event trace.

If the program was instead playing at the ‘intermediate’ level the goal annotated ‘intermediate’ trace would reveal that the move that led to the fork was random and not an intentional response to the player’s second move (see Figure 4 Right). Comparison with the raw event trace (Figure 3) shows that the events in the ‘intermediate’ goal-annotated event trace also are consistent with the raw event trace. Cases where the program produces correct results but does not follow the intended plan, i.e. indications of false positives, could thus be detected more easily in a goal-annotated event trace, because the underlying intentions of the results are different.

<p>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</p>	<p>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</p>
--	--

Figure 4. Left: goal-annotated expert trace, Right: goal-annotated intermediate trace

A certain amount of domain knowledge about the program being tested is necessary in order to perform meaningful tests. The Tic-Tac-Toe program contains 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. 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 identified as fatal. In our Tic-Tac-Toe study, we deliberately introduced such errors that our tests detected by finding games in which the events did not satisfy the pre- and postconditions of the current goal of the plan.

4. Prototype Application with ATM

In this section we describe an exploratory case study in which we 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 [5]. It provides a wider range of goal levels than the Tic-Tac-Toe example, as well as more interesting business goals. 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) and tool support on 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.

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

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 sequences, alternations, permutations, and iterations 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.

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 5 illustrates part of the goal graph for a bank.

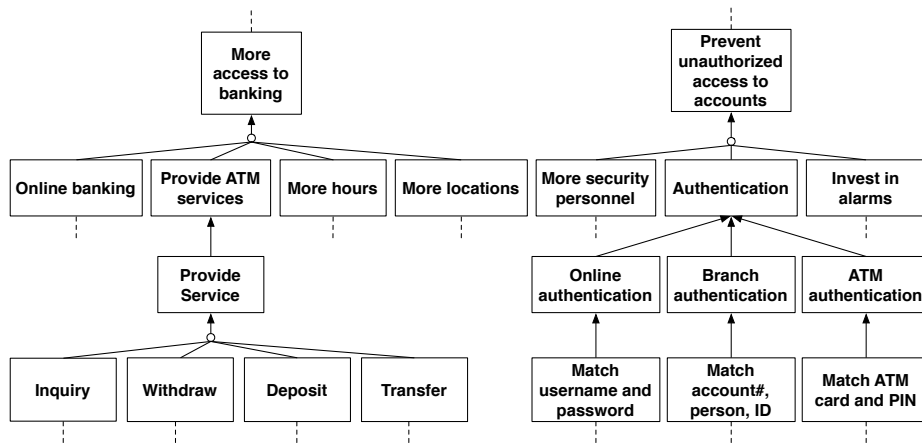


Figure 5. Portion of goal refinement graph

The graph includes high level business goals such as **Prevent unauthorized access to accounts**, low level functional goals such as **Withdraw** that map to a single component, and some goals, such as **ATM authentication** that impact many components of the system. Some parts of the system thus satisfy several goals.

4.3. ATM Session Plan

Figure 6 shows the GoalML plan for the functional goal **Provide ATM services**, again reconstructed from the requirements and standard banking business rules and goals.

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 6 illustrates valid interactions with the ATM.

```

<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 unauthenticated card">
        <goal name="Get PIN"/>
        <goal name="Get PIN"/>
        <goal name="Permanently retain ATM card"/>
      </sequence>
    </alternation>
  </iteration>
  <goal name="End session"/>
</plan>

```

Figure 6. ATM session plan

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 7 illustrates a sample of a GoalML-annotated piece of code (and its precompiled version).

4.5. Precompilation

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 8 illustrates a JESS assertion and a Drools assertion generated by the precompiler.

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 7).

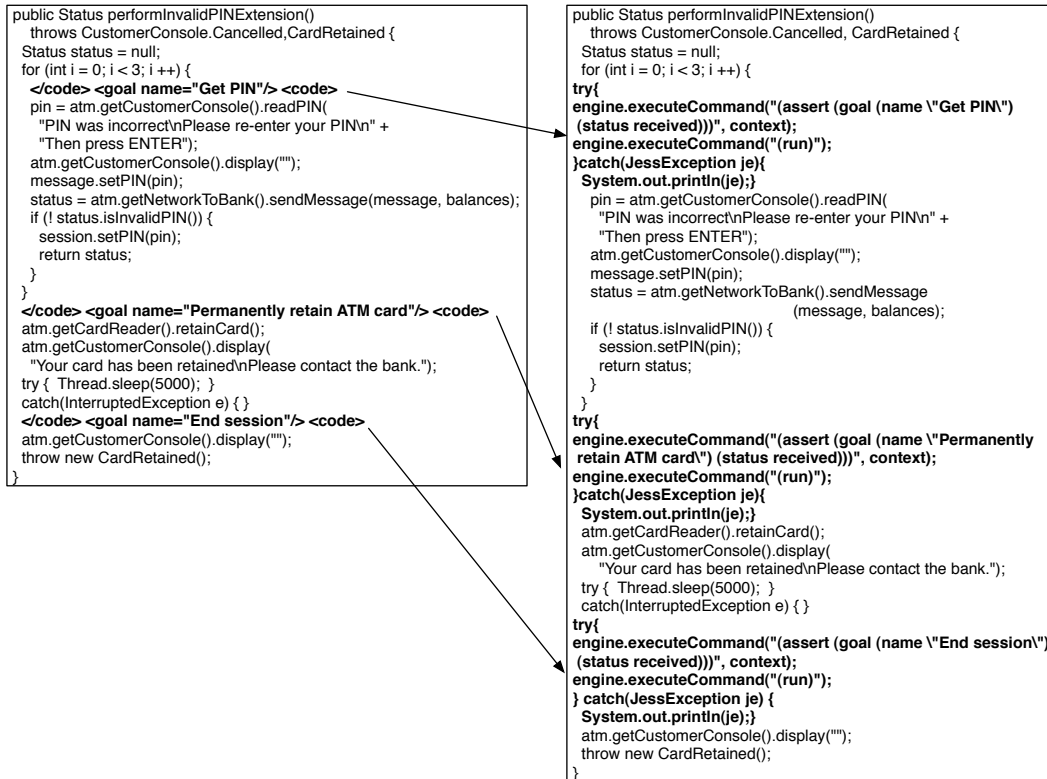


Figure 7. Goal-annotated and precompiled code samples

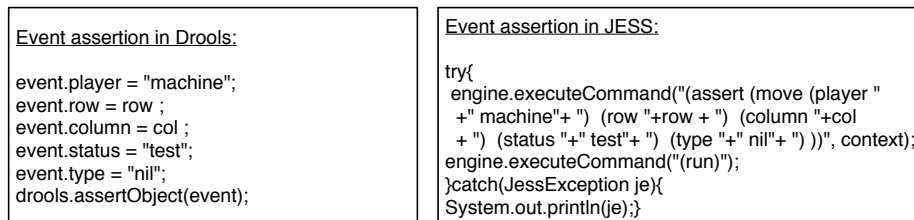


Figure 8. Precompiler output: Drools and JESS

4.6. 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.

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 9 illustrates one of the plan recognizer's rules. This rule recognizes when **Send 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 9. Sample plan rule

4.7. 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 [1] 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.

4.8. Executing Program and Recognizer

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 run-time. 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.

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 10 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 6). 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.

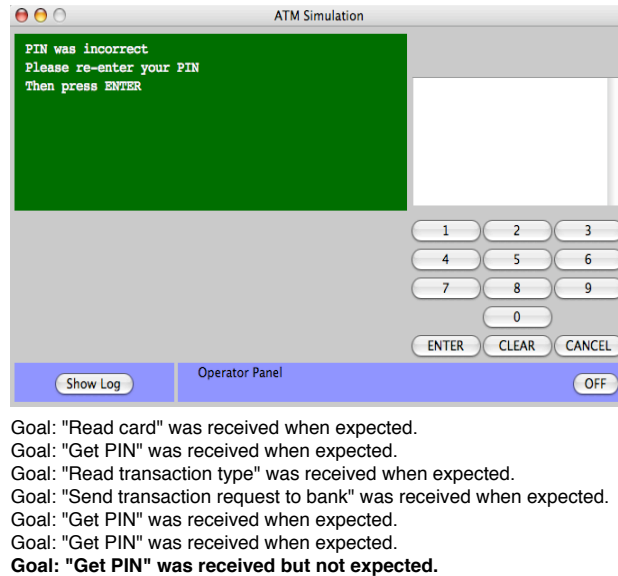


Figure 10. Goal trace with mismatch

4.9. 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 and fails to adhere to stakeholders' goals and business rules.

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 ambiguous how the design implements the invalid PIN requirement, which is that an invalid PIN may not be entered more than three times.

5. 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) [2], Software Tomography [6], residual testing [20, 22], Gamma Technology [21], and The Perpetual Testing project [23], 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.

Programming with assertions for discovering program errors has been a topic under investigation for a long time [10, 24]. 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. This issue is also addressed

by JML, an annotation language that supports runtime assertion checking and testing, as well as other uses [7]. Bandera verifies properties that refer to entities in the program [11] rather than annotating the source code. Although these techniques are effective for a number of types of verification, neither one focuses on verifying that the intended goals of the system are met.

Coppit and Haddox-Schatz have done some work in the area of specification-based assertions as a means for testing [10]. 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 [8]) as well as non-functional ones. Of particular significance is the work on goal refinement by van Lamsweerde *et al.* [12, 15, 25], and Mylopoulos *et al.* [18], specifically the work on mapping goals to requirements scenarios. There has also been substantial work in the area of business rules and goals. Kardasis and Loucopoulos proposed a roadmap for the elicitation of business rules based on the analysis of different stakeholders' enterprise goals. The main concept is that business rules are either the effect of enterprise goal operationalization, or the reason behind the way goals are operationalized. Based on that, business rules are collected by examining different stakeholders' views on their organization's objectives, and on the relation between these objectives and existing business constraints [14]. Although previous goal-driven and business rule-driven software development approaches refine and model goals they have not yet taken goals further than the design phase and have thus not been able to systematically verify the final system against the stated goals.

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 [26]. 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.

6. Lessons Learned and Future Work

Our specification-based approach compares goals and plans against program source code to improve testing efficiency by focusing testing resources on the most important behaviors – the stakeholder goals. We evaluated our approach on a Tic-Tac-Toe game and a larger, publicly-available ATM simulation. 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. Our approach takes advantage of the benefits of goal- and business rule modeling and induces coherency throughout the development process by extending the use of goals and rules to late software lifecycle activities such as testing. As a result, our approach focuses on developing software that reflects the policies and tactics that the stakeholders decided.

One limitation with our current oracle is that it knows nothing about the state of the world other than what the annotations tell it. In the future we plan oracles that determines the state of the world directly. This would reduce the number of annotations in the code, since event annotations would no longer be needed, and directly determine contributions the environment

makes toward goal satisfactions.

Our future goal is to evaluate the efficiency of our testing approach by conducting a more extensive case study in which we measure the time it takes to write plans and annotate code compared to other preparatory work for testing. We also want to evaluate how effective this approach is compared to other kinds of testing.

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 [17]. Architecture-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 formal 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 [4, 17]. This is part of a larger research project in architecture-based regression testing [16].

We therefore plan to extend our work to include not only goals, plans, 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 [3, 19]. Our ultimate goal is to provide forward mapping of business goals and rules described in requirements to other software artifacts and to provide automated traceability from those artifacts back to requirements.

References

- [1] Eclipse. <http://www.eclipse.org>.
- [2] Expectation-driven event monitoring (EDEM), 2002. <http://www.ics.uci.edu/~dhilbert/edem/>.
- [3] 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.
- [4] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In *SFM*, pages 122–147, 2003.
- [5] R. C. Bjork. ATM Simulation, 2002. <http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/>.
- [6] J. Bowering, A. Orso, and M. J. Harrold. Monitoring deployed software using software tomography. In *PASTE'02*, pages 2–9, 2002.
- [7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.
- [8] A. Cockburn. Structuring use cases with goals. *J. of Object-Oriented Programming*, Sept./Oct. 1997.
- [9] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., 2000.
- [10] 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.
- [11] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and R. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd Int. Conf. on Softw. Eng.*, 2000.
- [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] P. Kardasis and P. Loucopoulos. A roadmap for the elicitation of business rules in information systems projects. *Business Process Management Journal*, 11(4):316–348, 2005.
- [15] E. Letier and A. van Lamsweerde. Deriving operational software specifications from system goals. In *FSE'02*, pages 119–128, 2002.
- [16] 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.
- [17] H. Muccini, M. S. Dias, and D. J. Richardson. Systematic testing of software architectures in the C2 style. In *FASE*, pages 295–309, 2004.
- [18] 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.
- [19] 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.
- [20] 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.

- [21] A. Orso, D. Liang, M. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *ISSTA'02*, pages 65–69, 2002.
- [22] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE'99*, pages 277–284, 1999.
- [23] D. Richardson. Perpetual testing, 2002. <http://www.ics.uci.edu/~djr/edcs/>.
- [24] D. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.
- [25] A. van Lamsweerde and L. Willemet. Inferring declarative requirements specifications from operational scenarios. *IEEE Transactions on Software Engineering*, 24(12):1089–1114, Dec. 1998.
- [26] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.
- [27] M. Young and R. Taylor. Rethinking the taxonomy of fault detection techniques. In *ICSE'89*, pages 53–62, 1989.