Architecture-Based Testing Using Goals and Plans

Kristina Winbladh, Thomas A. Alspaugh, Hadar Ziv, and Debra Richardson
Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425
{awinblad, alspaugh, ziv, djr}@uci.edu

ABSTRACT

This paper presents a specification-based testing approach that compares software specifications defined at different levels of abstraction, e.g. architecture and implementation, against specified system goals. We believe that a goal-driven approach that connects several development artifacts through verification of specified goals provides useful traceability links between those artifacts as well as an efficient testing technique. Our approach begins with a system goal graph in which high-level goals are step-wise refined into low-level functional goals that can be realized as code components. Each of the architectural components is associated with a plan that describes the component's functional behavior. Source code is annotated with goals from plans and events that achieve the goals; code is then precompiled to emit those goals and events 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 executes, the goals and events emitted are automatically tested against the plans and expected results. As components achieve their component-level plans, a higher-level plan recognizer, concerned with the integration of components, can verify correct system behavior over the interaction trace of a collection of lower-level plans. A small example illustrates the concept.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Specification-based testing, Architecture-based testing, Goal-driven software development

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ROSATEA'06 July 17, 2006, Portland, Maine USA
Copyright 2006 ACM 1-59593-459-6/06/07 ...$5.00.

1. INTRODUCTION

Software testing is an essential but difficult activity whose principal task is to confirm the extent to which a system under development meets its stated requirements. We follow a specification-based testing approach that supports efficient use of testing resources by focusing on the most important behavior, i.e. that which the stakeholders specified. In previous work [16, 17], our approach centered on code-level testing against specified goals and plans, and proved to be particularly useful for identifying false positives and domain knowledge errors. This paper proposes an extension to that framework by connecting goals to other software artifacts, i.e. software architectures, to verify that these too adhere to specified system goals.

We start with a goal graph for a system and the plans associated with its functional goals. Each component in the architecture is associated with a plan that describes its functionality in terms of the subgoals it is supposed to achieve and the order in which they together achieve the overall goal of the functional unit. The implementation of each code component is annotated with goals from corresponding plans and with events whose intention is to transform the system state to satisfy a goal's set of pre- and postconditions. The goal- and event-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 [1]) 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. Higher-level system goals are expressed as plans that describe how components interact to provide system-wide functionality. As the system runs, component-level behavior is verified against the functional plans, and the sequence of achieved functional plans is verified against high-level plans concerned with the overall system behavior.

We believe there are several benefits to this approach. First, different software artifacts such as requirements, architecture, specifications, and implementation are representations of the same system at different levels of abstraction, they can also present different views of the system by focusing on certain aspects [8]. By connecting the software architecture to the requirements and implementation through goal

notations and testing, we provide a semi-automated strategy for identifying mismatches between them. Second, by linking development artifacts we provide a better means for traceability between these different development activities. Third, we believe that goals and plans are particularly useful as the connective media because they are an intuitive way to describe the system's intentions at any level of abstraction.

The extension to our previous work includes verifying that an architecture satisfies the overall system goals. This is done by associating each component in the architecture with a plan that describes its functional behavior. The implementation of these components are tested against these plans to verify low-level functional behavior. Overall system behavior, i.e. how components in the architecture together achieve a goal, is given by a system level plan. Goals that components achieve at the lower level are combined to determine whether higher-level plans have been satisfied.

The remainder of the paper is organized as follows: In Section 2 we describe the basic idea of our approach and in Section 3 we illustrate the approach an example. In Section 4 we discuss some of the related work in this area. Finally in Section 5 we conclude the paper by proposing future work.

2. THE BASIC IDEA

Our approach compares actual system behavior to expected system behavior expressed as goals that the system should satisfy. We determine expected system behavior from the higher-level goals, the lower-level goals they are refined into, and the relationships between those goals. Figure 1 shows an overview of our testing approach. At the upper levels of

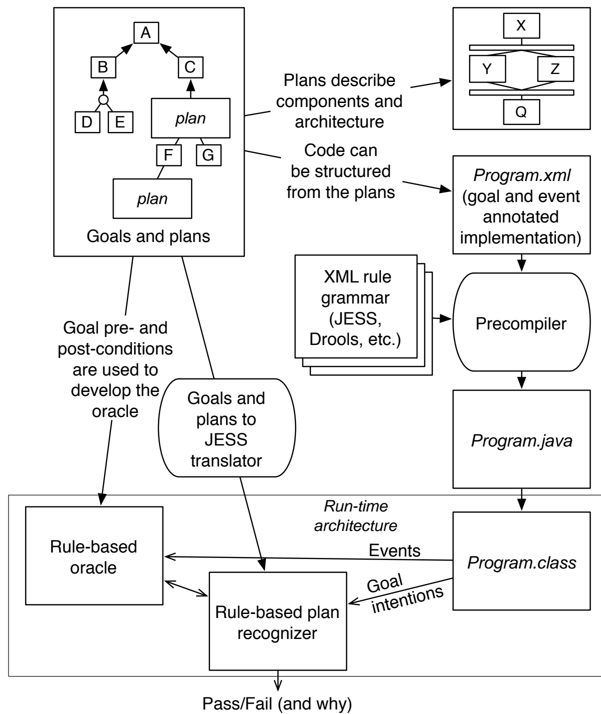

Figure 1: Process diagram

the goal refinement graph, the relationships are AND/OR refinements. At the lower levels, functional goals are refined by plans that describe 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 of the lowest-level goals. 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.

There has been extensive research on goal-directed requirements engineering and software engineering. Some effort is directed toward establishing the relation between goals and scenarios and how this relation can be used for requirements validation [14]. Scenarios, sequences of events of appropriate use of a system, are concerned with operational use of the system and goals are concerned with intentional use, i.e. they are declarative statements that complement the operational nature of scenarios. In terms of validation, goals present a meaningful description of the system to the system's stakeholders, because they capture the reasons why the system behaves as described in the scenarios. Other goal-driven software methods, such as *Tropos* [4], have used goals as the driving force from early requirements, to architecture design, to detailed design. We aim to extend the use of goals further by including them in the implementation, execution, and testing of the program, so that the operational use is verified against the intentional.

An essential feature of our approach is annotating the system's code with goals from the plans and events from the scenarios. We precompile the code, transforming the annotations into additional code that will emit statements of intention to satisfy each goal and events that the oracle uses to verify that each goal was in fact satisfied, at the appropriate points in the execution. The plans are automatically translated from a markup representation into a rule-based plan recognizer that runs with the program and matches its emitted goals with the goals expected by the plans. Satisfaction of the emitted goals is verified by an oracle. The oracle verifies if emitted events in fact change the state of the system according to a goal's set of pre- and postconditions. The oracle is (at present) rule-based and 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. After successfully testing implementation code against functional goal plans, we want to lift our approach one level of abstraction and test the architectural description against higher-level plans that describe how components work together to achieve a system goal. The higher-level plans, similarly to the low-level ones, describe goal sequences to be met by the system. These goal sequences, unlike the lower-level ones, do not consist of lowest-level goals and are not concerned with the internal workings of the component. The high-level plans instead describe how a sequence of components satisfies a system-level goal. As the lower-level plans are satisfied, the sequence in which they are satisfied is verified by the plan recognizer for the higher-level plan.

We believe that goals, goal graphs, plans, and scenarios are

an especially advantageous foundation on which this approach can connect several development artifacts through verification of specified goals and provide useful traceability links between those artifacts. 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. ILLUSTRATING EXAMPLE

We now illustrate our testing approach using a basic chat program that consists of one chat server, one time server, and two chat clients. In particular, we will show how a mismatch between an architecture and a system-level plan derived from the required system behavior could be detected. We constructed two alternative architectures for our chat program shown in Fig. 2. The architecture on the left (LA) accurately reflects the system-level plan while the architecture on the right (RA) does not. We implemented both RA and LA. Our method showed that the implementation of LA correctly followed the system-level plan while the implementation of RA did not.

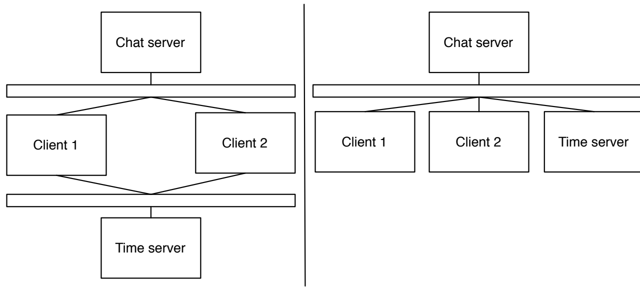

Figure 2: Left: correct, right: incorrect

The overall system behavior of our chat program can be summarized by a client typing and sending a message to the server, the server broadcasting that message to both clients, and each client requesting the time from the time server when they got the message. This system behavior is represented by the system-level plan shown in Fig. 3 and reflected in the architecture LA. The architecture RA, on the other hand, reflects a different overall system behavior: after a client has sent a message to the server the server requests the time from the time server and then broadcasts the message to the two clients with the time that the server got the message.

The system-level plan in Fig. 3 consists of a sequence of goals where each goal represents the successful completion of the execution plan for one of the components in the architecture (referred to as a component-level plan). The goal sequence indicates the order these components are expected to interact. The component-level plans describe the actual behavior of the components. For example, Fig. 4 shows the Client component-level plan. The system-level plan and the component-level plans are translated into a set of JESS rules that make up the plan recognizer. A fragment of the plan recognizer corresponding to the system-level plan is shown in Fig. 5. The rules in the plan recognizer corresponding to the Client component plan are shown in Fig. 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<plan goal="System strategy for one message">
    <iteration>
        <sequence>
            <goal name="Client plan matched"/>
            <goal name="Server plan matched"/>
            <goal name="Client plan matched"/>
            <goal name="Client plan matched"/>
            <goal name="Time server plan matched"/>
            <goal name="Client plan matched"/>
            <goal name="Client plan matched"/>
            <goal name="Time server plan matched"/>
            <goal name="Client plan matched"/>
            <goal name="Client plan matched"/>
        </sequence>
    </iteration>
</plan>
```

Figure 3: System plan for chat program

```
<?xml version="1.0" encoding="UTF-8"?>
<plan goal="Provide chat client services">
    <iteration>
        <alternation>
            <sequence>
                <goal name="Get message from user"/>
                <goal name="Send message to server"/>
            </sequence>
            <sequence>
                <goal name="Get message from server"/>
                <goal name="Send request to time server"/>
            </sequence>
            <sequence>
                <goal name="Get time from time server"/>
                <goal name="Display message"/>
            </sequence>
        </alternation>
    </iteration>
</plan>
```

Figure 4: Correct client component plan

The implementation of each component is annotated with GoalML statements that represent the goals from their respective plans. This annotated version of the implementation is precompiled by translating the GoalML statements into Java code that asserts JESS facts representing the goals into the JESS working memory at the appropriate point in the execution. Fig 7 shows annotated code of the Client component in the upper box and the precompiled version in the bottom box. As the program executes, goals are emitted from the program into the rule engine and compared to the component-level plan. The actual monitoring of goals by the plan recognizer is affected by the JESS pattern matching facility that matches assertions to condition patterns in the left hand sides of the rules representing the component-level plans. Once all the rules corresponding to a component's plan have been matched, the fact representing the plan's completion is added to the rule engine's working memory. The facts about component plan completion are in turn matched against rules that monitor the system-level plan.

It will be helpful at this point to get a feel for how the plan recognizer works by considering an example. Suppose that the user tries to send a message. When the user enters

```
(deffacts initialize-arch-plan
   (goal (name "Client plan matched") (status expected))
   )

(defrule arch-plan-1
   ?g1 <- (goal (name "Client plan matched")(status matched))
   =>
   (assert (goal (name "Server plan matched") (status expected)
                 (condition ignore)))
   (assert (goal (name "arch-plan-1")))
   (retract ?g1)
   (printout t  "arch-plan-1"  crlf)
    )
```

Figure 5: Part of System plan expressed in JESS

```
(defrule client-plan-4
   (goal (name "Send message to server")(status matched))
   =>
   (assert (goal (name "Client plan matched") (status matched)
                 (condition ignore)))
   (printout t  "Client plan was matched "  crlf))
```

Figure 6: Part of Client plan expressed in JESS

a message and presses the send button, the first method called is `getMessageFromUser()`. There is a goal annotation in this method that will cause a Get message from user goal to be asserted into the JESS working memory when the method is called. This goal matches the condition of the first rule of the client plan. The action of this rule is to assert a fact that says to expect a goal Send message to server. The next method called is `sendMessage()` (see the sendMessage() method in Fig. 7 for the precompiled version of the Java code that actually creates and asserts the JESS fact representing the goal). That method has a goal annotation that will cause a Send message goal to be asserted into the working memory. This goal will be approved and accepted since it is expected. It then matches the condition of rule `client-plan-4` (see Fig 6). The action of that rule is to assert the fact representing the goal Client plan was matched. This new fact will match the condition of the system-level recognizer rule `arch-plan-1` (see Fig. 5). The result of this rule firing will be to assert that the next expected goal is Server plan matched.

The significance of the example in the previous paragraph is that it shows how expectation based goal monitoring and matching seamlessly moves from detailed component-level plans to architectural system-level plans.

Now that we have concretely demonstrated our basic technique, it only remains to show how it is used to actually find a problem. Fig 8 shows part of the trace resulting from running the implementation of architecture RA and comparing the emitted goals against the system-level plan that is based on the architecture LA as we described at the beginning of this section. As Fig. 8 demonstrates, the Client plan matched and Server plan matched goals were achieved but a Time server plan matched goal was asserted instead of Client plan which was the next expected system-level goal according to the plan (see Fig. 3).

4. RELATED WORK

Work on refinement of goals into operationalizable requirements, requirements that can be operationalized into code

```
public void sendMessage(String text){
        </code>
        <goal name="Send message to server"/>
        <code>
        server.getMessage(alias + ": " + text);
        entryField.setText("");
    }
```

```
    public void sendMessage(String text){
try{
    engine.executeCommand("(assert
              (goal (name \"Send message to server\")
                    (status received)))", context);
    engine.executeCommand("(run)");
    }catch(JessException je){
        System.out.println(je);}

        server.getMessage(alias + ": " + text);
        entryField.setText("");
    }
```

Figure 7: Top: annotated code, bottom: precompiled code

```
Goal: "Get message from user" was received when expected.
Goal: "Send message to server" was received when expected.
Client plan was matched
Goal: "Client plan matched" was received when expected.
Goal: "Get message from client" was received when expected.
Goal: "Send request to timeserver" was received when expected.
Server plan was matched
Goal: "Server plan matched" was received when expected.
Goal: "Get request" was received when expected.
Goal: "Send time" was received when expected.
Time server plan was matched
Goal:"Time server plan matched" was received but not expected
```

Figure 8: Goal-trace with an error

components, has been carried out for at least a decade, including goals reduced into both functional requirements (e.g., use cases [5]) as well as non-functional ones. Of particular significance is the work on goal refinement and mapping goals to requirements scenarios by van Lamsweerde *et al.* [7, 9, 15], and Mylopoulos *et al.* [12].

Coppit and Haddox-Schatz have used specification-based assertions as a means for testing [6]. Their method translates formal specifications to program assertions to be inserted in the implementation. During program execution these assertions are 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 code. We verify the implementation against these plans at the same time as we are testing intermediate and final results against specification-based oracles. Since 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 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.

Higher level goals should correspond to higher level testing activities such as integration and regression testing. We plan to explore ways in which our approach complements or relates to other research in this area, in particular architecture-

based testing. Architecture-based testing typically means testing that source code matches a specified architecture. This is often called conformance testing, i.e. checking that an implementation fulfills its specification [11]. Several systematic 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 an approach from specifications in the C2 architectural style through Abstract Labeled Transition Systems (ALTS) down to code-level execution of the architecture-based tests [3, 11]. This is part of a larger research project in architecture-based regression testing [10].

5. CONCLUSIONS AND FUTURE WORK

We have presented a specification-based testing approach that allows testing of lifecycle artifacts of different abstraction levels to be verified against system goals. This is a continuation of previous work in which we showed how our approach is successful in finding false positives and domain knowledge errors in code with regard to the specified goals and plans. The new extension allows integration testing of architectural components with regard to system-level plans that realize system-level goals. We demonstrated this approach on a small concrete and implemented example with artifacts of manageable size to show in this paper.

We plan to implement tool support for our approach and evaluate it on a more significant case study, such as an adaptive e-commerce site. Testing the complex nature of asynchronous computation between server and client is not entirely straightforward neither is testing adaptive strategies. Our approach would allow verification of the program against the specified adaptation strategies if these were represented as plans. We will also perform a cost-benefit analysis of our approach with regard to current testing approaches. Future work also includes architectural simulation so that the high-level integration plans can be verified prior to code implementation. We have previously verified low-level plans by expressing and simulating components' behaviors in goal-annotated scenarios. This could become an important feature of our approach since early lifecycle testing of design artifacts allows errors to be resolved before they become too expensive. We also believe that extending our testing approach to additional artifacts, such as software architectures and requirements, and with traceability among those artifacts, more efficient, purposeful testing can be accomplished [2, 13]. 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.

6. REFERENCES

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

[2] 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.

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

[4] J. Castro, M. Kolp, and J. Mylopoulos. Towards requirements-driven information systems engineering: the tropos project. *Inf. Syst*, 27(6):365–389, 2002.

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

[6] 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.

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

[8] P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] S. Uchitel, R. Chatley, J. Kramer, and J. Magee. Fluent-based animation: Exploiting the relation between goals and scenarios for requirements validation. In *RE*, pages 208–217, 2004.

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

[16] K. Winbladh, T. A. Alspaugh, H. Ziv, and D. J. Richardson. An automated approach for goal-driven, specification-based testing. In *ASE'06 - In Submission*, 2006. `http://www.ics.uci.edu/~awinblad/publications.html/`.

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