AUTOMATED SUPPORT FOR TEST-DRIVEN SPECIFICATION

Edward L. Jones
Florida A&M University
Tallahassee, Florida
USA
ejones@cis.famu.edu

ABSTRACT
This paper discusses an approach to test-driven specification, whereby specification is aided by test case design, and vice-versa. Decision tables are used as a lightweight specification language. We introduce an automated tool set to convert static decision tables into an executable form that supports interaction between test case design and specification refinement processes. We demonstrate the use of the toolset to accomplish three complementary functions: (1) to analyze the specification for completeness and consistency; (2) to assess the functional adequacy of a test set; and (3) to produce a test oracle. The contribution of this approach is a lightweight technique that exploits the duality between specification and testing early in the software lifecycle.

KEY WORDS
Specification-driven testing, decision table, model-based testing, functional coverage.

1. Introduction
Software testing has long been viewed a dual process to software development, but performed as an adjunct to development. The V-model [1] was among the first lifecycle models that attempted to call attention to the relationship between the creative activities of successive stages in the development lifecycle and the respective testing activities that occur during the “testing thread” of the lifecycle. The V-model draws attention to the need to plan for the testing activities that will be performed following the completion of the coding stage.

The W-model of software testing [2], better elaborates the relationship between development activity and testing by associating test activity to be applied to artifacts produced by each stage in development thread. A theme of the W-model is that upon the development of an artifact, an immediate verification process is applied. In that vein, this paper addresses the problem of testing the specification of software function. The premise of this paper is that specification and testing are two sides of the same coin – stating the intent, and proving that the software meets the intent. Each activity is challenging, but solving one contributes to solving the other.

Major issues of specification-based testing include specification correctness, the oracle problem, and test set adequacy. In the author’s opinion, specification is the single most difficult problem in software engineering for two reasons: it is the basis for developing software and also the basis for testing. Both ends of the development process are affected by the quality of the specification.

The second issue is the oracle problem: given a test set, determine the correct expected response for each set of stimuli. Producing an oracle requires that the specification be interpreted for each test datum to compute the intended response. For this to be possible, the specification must satisfy properties such as correctness, completeness, consistency and clarity. Even when these properties are satisfied, in most instances the engine for mapping stimuli to responses requires a manual, human involvement at the cost of expended effort and human error. Too often, the matter of the test oracle is deferred to the stage where testing results are examined; in the worst case, uncontested test results are accepted as the standard answer.

The third testing issue concern is measuring the effectiveness of the testing process. Current measures include the use of adequacy criteria such as control-flow or data-flow coverage measures [3]. Although some advantages accrue from the use of such measures, such as having quantitative goals and progress indicators that can provide guidance for when to stop testing, no single technique or cocktail of techniques has emerged.

The thesis of this paper is that the power of the decision table and other specification models is not realized unless the model can be given “life”, i.e., be made dynamic. The software developer needs to interact with the model. We propose that the model be made executable and capable of emulating the cause-effect/stimulus-response intents encoded in the model. The executable model, when presented with representative inputs (stimuli), produces responses specified in the model. We have developed a dynamic analyzer tool that identifies anomalies in the model, while computing a coverage measure (adequacy
criterion) for a given set of test stimuli. Other approaches to automating the creation, analysis and transformation of decision tables into computer programs have been reported. Our approach is unique in its objectives—to unite specification and testing—and in our desire to provide open source tools to a broad community of users.

1.2 Related Work

Many approaches to specification have been proposed, ranging from highly formal methods too complex for the average software developer, written prose, to graphical techniques capable of representing perspectives but rarely the whole. The benefits of formal methods are generally the ability to automate the checking process to ensure that the model is correctly formed and to identify classes of anomalies in the system being specified. Advances are being made in generating test cases and test oracles from formal models.

In this paper we use the decision table, a simple formalism, as the specification language. Decision tables have an intuitive appeal—this is not to say that they are simple or easy to master—and are suitable for reasoning about the cause-effect behaviour of software. Studies have demonstrated that decision tables are an effective specification notation, especially as specifications grow in size and complexity [4].

In the 1970’s, decision tables were recognized as mathematical formalisms suitable for use as a high level programming language [5,6]. Work on automatically converting decision tables into (from) computer programs has been reported [7,8]. A number of commercial websites advertise decision table tools for managers; some, such as [9] provide online tools to convert decision tables into computer programs.

Vanthienen and others have done much to advance the use of decision tables in knowledge engineering, artificial intelligence, and decision support systems [10,11,12,13]. Vanthienen’s Prologa tool automates the creation, verification, and optimization of decision tables. Because decision tables are models, they can be malformed or otherwise exhibit anomalies such as redundancy, ambiguity or incompleteness. Prologa offers extensive checking features, and provides a logic-based input language to define the logical combinations that produce a specific response. Prologa and the other decision table tools place certain restrictions on the size and style of decision table supported and assumptions about how to handle the combinatorial explosion of rules.

Textbooks on software testing routinely introduce decision tables as a specification technique that can be used as the basis for test case design [14]. This work attempts to bridge specification with testing, and assumes an open source decision table editor for creating decision tables. We do not assume DT optimization, but require the limited entry form where each decision rule is expressed as elementary Boolean values (‘Y’, ‘N’, ‘-’).

2. TEST DRIVEN SPECIFICATION

2.1 An Example

In this section, we introduce an example that shows the initial step in the evolution of a decision table to specify the software intent given in the narrative specification that follows. In a later section, we will demonstrate how the toolset supports the evolution process for the specification and an adequate test set for testing the specification.

Specification for Employee Pay: Calculate employee pay, including overtime paid at 1.5 times the hourly rate of hourly employees for qualifying hours. The normal work week is 40 hours; the maximum number of paid hours is 80. Salaried employees earn over $30 per hour, and are paid overtime only for hours in excess of 60 hours. Salaried employees working less than 40 hours are paid for the normal work week. Hourly employees earn less than $30 per hour, and are paid for all hours worked up to the maximum; they are paid overtime for hours in excess of 40.

The first step towards representing this narrative specification is to identify stimuli and responses. From the specification, we can deduce that the stimuli are the hours worked and the hourly rate of pay. The responses are the amount of regular (non-overtime) pay and the amount of overtime pay. The black-box stimulus-response behavior is depicted in Figure 1.

In this section, we introduce an example that shows the initial step in the evolution of a decision table to specify the software intent given in the narrative specification that follows. In a later section, we will demonstrate how the toolset supports the evolution process for the specification and an adequate test set for testing the specification.

The next step is to determine the cause-effect relationships between the stimuli and responses. The decision table in Figure 2 captures the properties of the stimuli that select the formula to use to compute the responses. The term, rule, when used without qualification (decision or action), refers to an entire column, and combines the decision rule with its corresponding action rule. The decision table in Figure 2 is known to be incomplete—it specifies only two rules—and is presented to establish the terminology of a decision table. In the remainder of this section we will show how the toolset is used to evolve concurrently the decision table and an adequate set of data for testing the decision table specification.

2.2 Decision Table as a Model for Testing

Decision-based test adequacy criteria have been proposed [14] which focus on decision elements, such as conditions or combinations of conditions. For the purposes of this

![Figure 1. Black-Box Schematic for Pay](image-url)
paper, we define test adequacy in terms of the decision table rules, justified by the observation that each rule partitions the function into equivalence classes—assuming that the table does not contain ambiguous rules. Equivalence partitioning is an accepted testing strategy that places more value on the nature of test cases than on the quantity of test cases. It is important that every rule is exercised. We define a black-box testing adequacy criterion, \textit{functional coverage}, as follows:

\[
\text{Functional Coverage} = \frac{\text{#rules satisfied}}{\text{#rules in decision table}}
\]

2.3 Automation
When testing complicated software with a static specification, it is difficult to determine manually whether each rule has been covered and if there are anomalies (redundant, ambiguous or missing rules) in the decision table specification. The Test Driven Specification Toolset (TDST) automates the following capabilities:

1. Evaluate a specified decision table condition, given specific test data;
2. Determine whether a specified decision rule is satisfied, given specific test data;
3. Select actions to be performed for a specified decision rule;
4. Perform a specified decision table action to compute responses.

5. Calculate the functional coverage, given a set of test data.
6. Generate expected results, given test data.

Components of TDST include the Decision Table Editor (DTE), and the Decision Table Analyzer Generator (DTAgen). DTE provides a GUI interface for creating and editing decision tables. DTE supports limited entry decision tables, the form shown in Figure 2, in which the decision rules take the appearance of truth tables. This form simplifies automated analysis. Most extant decision table tools, such as Prologa [Vanthienen], are proprietary, and do not give the user direct control over decision and action rules.

The generator program, DTAgen, converts the static decision table into a C++ class implementing an executable equivalent, and generates the Functional Test Analyzer (FTA) program, which measures the functional coverage of a given test set while detecting anomalies in the specification. Further details about the implementation of the toolset are found in Clark [15].

2.4 Concurrent Specification and Test Set Refinement
In this section we demonstrate the use of the toolset to guide the refinement of the decision table in Figure 3 and the creation of a set of test data capable of thoroughly testing the specification.

Figure 3 shows the result from testing the decision table in Figure 2 using the test set 3(b). The initial decision table is clearly incomplete, since it gives only two rules. The test coverage in 3(a) shows that the test data set satisfies the two rules, but that test cases two and four do not satisfy any rule. That is, the decision table is missing rules to handle these test cases. In Figure 3(c), the test generation report makes this explicit by tagging the test case as “MISSING” and tagging the expected values for regular pay and overtime pay as “MISSING”. Note that the tag “VALID” means that the test data satisfies exactly one rule. Figure 3 illustrates that achieving 100% coverage on an incomplete specification is possible, but that with sufficient test data, the toolset can alert the developer that more work is required on the specification.

<table>
<thead>
<tr>
<th>CONDITIONS</th>
<th>DECISION RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>hours &gt; 40</td>
<td>N N</td>
</tr>
<tr>
<td>rate &gt;= 30</td>
<td>Y N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTIONS</th>
<th>ACTION RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>overtime_pay = 0;</td>
<td>X X</td>
</tr>
<tr>
<td>regular_pay = hours * rate;</td>
<td>X</td>
</tr>
<tr>
<td>regular_pay = 40 * rate;</td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 2. Pay Decision Table Specification (Initial Version)

<table>
<thead>
<tr>
<th>(a) FUNCTIONAL TEST COVERAGE REPORT</th>
<th>(b) TEST SET</th>
<th>(c) TEST CASE GENERATION REPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rules --&gt;</td>
<td>hours</td>
<td>rate</td>
</tr>
<tr>
<td>Test -- --</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>Case 1 2</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>====== -- --</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>1 X</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RULES NOT SATISFIED:
FUNCTIONAL_COVERAGE = 2 / 2

Figure 3. Analysis reports from the toolset for initial specification and test set.
Figure 4 shows a subsequent version of the decision table in which additional rules have been added. Again, the given test set achieves functional coverage of 100%. However, two anomalies are identified in the test case generation report: ambiguous rules, and an erroneous rule. The fifth test data pair (70, 40) results in the firing of two rules (4, 5); the sixth data pair (90,50) satisfies three rules (4,5,6). Rule 4 is fired for test case four, but results in the computation of a negative (-$900) overtime pay. Closer examination shows that this rule is incorrect: salaried employees qualify for overtime pay only if they work more than 60 hours.

Figure 5. Analysis reports from the toolset for version 3 of the specification and test set.

Figure 5 shows a subsequent version of the decision table in which an attempt has been made to correct rule four in Figure 4. This attempt resulted in the removal of the rule (4) that test case four previously satisfied. This situation is shown by the “MISSING” tag on test case four in the test case generation report. Some progress was made in removing the ambiguous rules: test case six now fires only two rules instead of three.

The final specification with adequate test set is given in Figure 6. The sequence of refinements in this sections have illustrated the capability of the TDST toolset to assist the human user to identify opportunities for improving the specification to remove anomalies identified by test data, while also evolving the set of test data to ensure coverage of the specification.

### 3. CONCLUSIONS AND FUTURE WORK

There should be a link between specification, which occurs early in the software lifecycle, and testing, which occurs at the end. The toolset presented in this paper connects these ends of the lifecycle. The toolset can improve jointly the quality of test data and the quality of the specification used to create test cases. Creating test cases is labor intensive, tedious and error prone. The toolset can generate test cases automatically, providing the means of verifying the specification on the one hand, while accelerating the preparation for testing, on the other hand.

The emphasis of this paper has been the joining of concerns for specification and testing. The TDST toolset supports an iterative process in which test cases are constructed to reveal anomalies in the specification. Refinements to remove specification anomalies may in turn reveal inadequacies in the test data. Iteration terminates when no further anomalies are discovered in either the specification or the test data.

Although we feel that intellectual engagement is an important part of this process, there are circumstances where additional automation can speed the process and increase accuracy. We are currently working on using random generation of stimuli to replace or augment manually constructed test sets. Random generation is likely to produce imaginative but interesting test sets that stress the specification in ways that overcome bias (or rigor) on the part of the human developer, such as the common practice of considering only nominal behavior in the use of a software component.

Work is also underway to apply the toolset to model state-based specifications such as finite state machines, SCR charts, and pre- and post-conditions, and Mill’s black-box and state-box structures [16]. Although there are limitations to the applicability of decision tables, typical software contains many components whose behavior can be specified as a decision table [14]. The intent of this effort is to provide a unifying toolset to support predicate-based specifications, with the goal of enabling concurrent specification and testing.

Acknowledgments
The author thanks the reviewers for feedback for improving this paper. This work was partially supported by NSF Minority Institutions Infrastructure Grant #0424556, and by a grant from the Medtronic Foundation.

References


[9] Online service to convert DT to code in selected language, Decision Table Source Code Generation - Version Build 4.00, http://dtable.proxjonline.com/


